STUDENT GUIDE

FACULTY OF INFORMATION TECHNOLOGY

PROGRAMMING 622

RICHFIELD

Registered with the Department of Higher Education as a Private Higher Education Institution under the Higher Education Act, 1997.Registration Certificate No. 2000/HE07/008

FACULTY OF INFORMATION TECHNOLOGY

LEARNER GUIDE

MODULE: PROGRAMMING 622

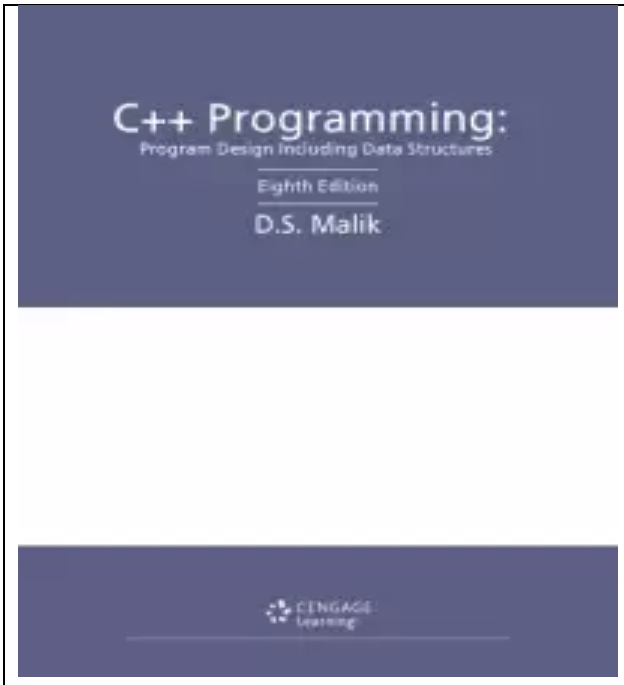
PREPARED ON BEHALF OF

RICHFIELD GRADUATE INSTITUTE OF TECHNOLOGY (PTY) LTD

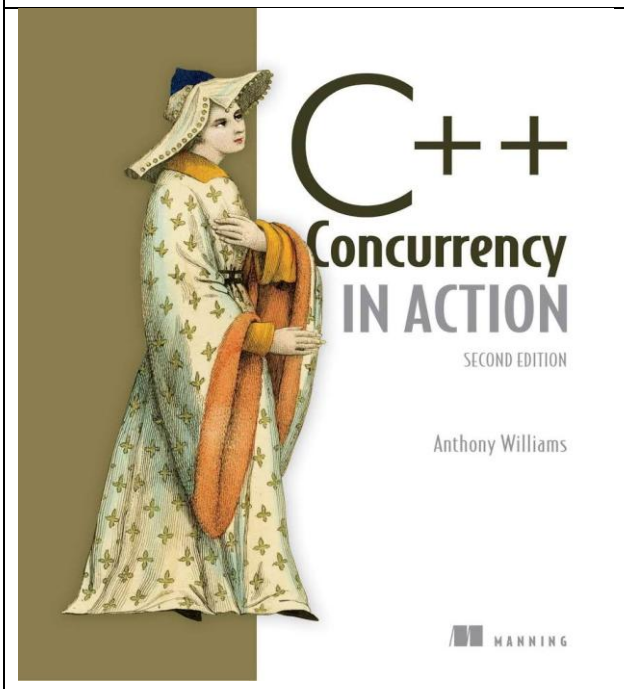
RICHFIELD GRADUATE INSTITUTE OF TECHNOLOGY (PTY) LTD

Registration Number: 2000/000757/07

All rights reserved; no part of this publication may be reproduced in any form or by any means, including photocopying machines, without the written permission of the Institution.



Malik, D.S 2018. C++ Programming: Program Design including Data Structures. Eighth Edition. United Kingdom: Cengage Learning



Williams, A. 2019 C++ Concurrency in Action PUBLISHER Simon and Schuster, 2019 ISBN 1638356351

Contents

1 TOPIC 1: SOFTWARE ENGINEERING PRINCIPLES AND C++ CLASSES 9

1.1 Software Life Cycle 9

1.2 Software Development Phase 10

1.3 Algorithm Analysis: The Big-O Notation 17

1.4 Classes 26

1.5 EXAMPLE 1-9 28

1.6 Data Abstraction, Classes, and Abstract Data Types 48

1.7 Identifying Classes, Objects, and Operations 54

Review Questions 58

2 TOPIC 2: STANDARD TEMPLATE LIBRARY (STL) 65

2.1 Components of the STL 65

2.1 Container Types 66

2.2 Sequence Containers 66

2.3 Sequence Container: vector 67

2.4 Containers and the Functions begin and end 74

2.5 The copy Algorithm 83

2.6 ostream Iterator and Function copy 87

2.7 Iterators 98

Review Questions 111

3 TOPIC 3: LINKED LISTS 118

3.1 Linked Lists 118

3.2 Item Insertion and Deletion 125

3.3 Linked List as an ADT 139

3.4 Linked List Iterators 142

3.5 Unordered Linked Lists 161

3.6 Ordered Linked Lists 177

3.7 Doubly Linked Lists 195

3.8 STL Sequence Container: list 215

3.9 Linked Lists with Header and Trailer Nodes 220

Review Questions 224

4 TOPIC 4: RECURSION 233

4.1 Recursive Definitions 233

4.2 Problem Solving Using Recursion 238

4.3 Recursion or Iteration? 264

Review Questions 281

5 TOPIC 5: STACKS 285

5.1 Stacks 286
5.2 Linked Implementation of Stacks 308
5.3 Stack as Derived from the class unorderedLinkedList 328
5.4 Application of Stacks: Postfix Expressions Calculator 330
5.5 Program Listing 345
5.6 Removing Recursion: Nonrecursive Algorithm to Print a Linked List Backward 348
5.7 STL class stack 352
Review Exercise 357
6 TOPIC 6: QUEUES 365
6.1 Queue Operations 366
6.2 Implementation of Queues as Arrays 369
6.3 Linked Implementation of Queues 385
6.4 Priority Queues 400
6.5 Server 411
Review Questions 435
7 TOPIC 7: SEARCHING AND HASHING ALGORITHMS 443
7.1 Search Algorithms 443
7.2 Sequential Search 445
7.3 Ordered Lists 449
7.4 Binary Search 450
7.5 Insertion into an Ordered List 457
7.6 Hashing 461
7.7 Rehashing 473
7.8 Chaining 483
8 TOPIC 8: SORTING ALGORITHMS 492
8.1 Sorting Algorithms 492
8.2 Insertion Sort: Array-Based Lists 502
8.3 Shellsort 516
8.4 Mergesort: Linked List-Based Lists 532
8.5 Heapsort: Array-Based Lists 547
8.6 Priority Queues (Revisited) 560
9 TOPIC 9: BINARY TREES AND B-TREES 567
9.1 Binary Trees 567
9.2 Binary Tree Traversal 576
9.3 Binary Search Trees 597
9.4 Binary Search Tree: Analysis 616
9.5 Nonrecursive Binary Tree Traversal Algorithms 619
9.6 Binary Tree Traversal and Functions as Parameters 626

9.7 AVL (Height-Balanced) Trees 633
9.8 AVL Tree Rotations 642
9.9 B-Trees 663
Review Questions 689
10 TOPIC 10:GRAPHS 696
10.1 Introduction 696
10.2 Graph Definitions and Notations 698
10.3 Operations on Graphs 704
10.4 Graphs as ADTs 705
10.5 Graph Traversals 712
10.6 Shortest Path Algorithm 720
10.7 Shortest Path 724
10.8 Minimum Spanning Tree 732
10.9 Topological Order 745
10.10 Breadth-First Topological Ordering 747
11 ASSIGNMENT.GUIDELINES. ATTENTION STUDENTS/STAFF 768

The Information Technology (IT) qualification at Richfield College is a dynamic and future-focused program designed to equip students with advanced technical, analytical, and problem-solving skills. At the core of the qualification is a commitment to academic excellence, industry alignment, and innovation, fostering graduates who are proficient in addressing modern technological challenges. This qualification strategically integrates theoretical knowledge with practical applications, preparing students for various roles in the IT sector. The IT program is structured to address the growing complexity of the evolving technological landscape.

The Higher certificate in Information Technology (HCIT) program is a foundational stepping-stone for students who wish to pursue further studies or enter the workforce. Graduates of this program are well-prepared to articulate to the Diploma in IT (DIT) or the Bachelor of Science in IT (BSc IT) qualifications, providing a seamless transition for those seeking to deepen their knowledge and skills in specialized IT areas. Additionally, the program equips students with the essential competencies for entry-level IT roles such as IT Support Technicians, Junior Web/ System Developers, IT Administrators, etc.

The Diploma in Information Technology (DIT) is a comprehensive and practical program designed to build a strong foundation in IT principles while equipping students with the hands-on skills required to meet industry demands. Focused on both theoretical knowledge and applied learning, this qualification prepares students for intermediate-level roles in IT and serves as a stepping-stone for further academic progression or specialization. Graduates of this program are well-prepared to articulate to the Bachelor of Science in IT (BSc IT) qualification. The curriculum covers programming, networking, database management, system analysis etc., ensuring graduates possess the competencies to solve real-world IT challenges effectively.

The Bachelor of Science in IT (BSc IT) program is structured to address the growing complexity of the evolving technological landscape. Through carefully curated modules, students gain a deep understanding of software development, database management, cloud computing, cybersecurity, IT management, artificial intelligence, machine learning, networking etc. Graduates of this program are well-prepared to articulate to the Bachelor of Science Honours in IT qualification. The curriculum is designed to bridge the gap between academic learning and real-world applications, thus fostering innovation and an entrepreneurial mindset. Students are encouraged to participate in research and

practical learning.

The programming focus within the IT qualification exemplifies academic innovation and professional alignment. By integrating a diverse range of programming languages with practical application, the curriculum prepares students to excel in the rapidly evolving tech industry. The program aligns with industry courses from globally recognized leading tech giants, such as Oracle, AWS, IBM, etc ensures that graduates possess the credentials to validate their expertise in software development and cloud-based technologies. This blend of foundational knowledge, practical experience, and industry-standard courses prepares students for immediate employment and establishes a strong basis for long-term career advancement in software development.

Programming 622 builds upon the foundational skills established in Programming 621 by delving into the world of data structures and algorithms using C++. This module equips students with the tools to efficiently organize and manage data through an exploration of fundamental data structures such as linked lists, stacks, queues, trees, graphs, and hash tables. In parallel, students will investigate key algorithms including sorting, searching, and recursion, while also learning to evaluate algorithmic efficiency and computational complexity. Through a series of hands-on exercises, practical projects, and real-world applications, Programming 622 challenges students to design, implement, and optimize solutions for complex problems. This comprehensive approach not only deepens their understanding of algorithmic thinking but also reinforces their proficiency in C++, preparing them for advanced programming challenges and innovative software development.

1 TOPIC 1: SOFTWARE ENGINEERING PRINCIPLES AND C++ CLASSES



LEARNING OUTCOMES:

Learn about software engineering principles

- Discover what an algorithm is and explore problem-solving techniques
- Become aware of structured design and object-oriented design programming methodologies
- Become aware of private, protected, and public members of a class
- Explore how classes are implemented
- Become aware of Unified Modeling Language (UML) notation
- Examine constructors and destructors
- Become aware of abstract data type (ADT)
- Explore how classes are used to implement ADT

1.1 Software Life Cycle

A program goes through many phases from the time it is first conceived until the time it is retired, called the life cycle of the program. The three fundamental stages through which a program goes are development, use, and maintenance. Usually a program is initially conceived by a software developer because a customer has some problem that needs to be solved and the customer is willing to pay money to have it solved. The new program is created in the software development stage. The next section describes this stage in some detail.

Once the program is considered complete, it is released for the user to use. Once users start using the program, they most certainly discover problems or have suggestions to improve it. The problems and/or ideas for improvements are conveyed to the software developer, and the program goes through the maintenance phase.

In the software maintenance process, the program is modified to fix the (identified) problems and/or to enhance it. If there are serious/numerous changes, typically, a new version of the program is created and released for use.

When a program is considered too expensive to maintain, the developer might decide to retire the program and no new version of the program will be released.

The software development phase is the first and perhaps most important phase of the software life cycle. A program that is well developed will be easy and less expensive to maintain. The next section describes this phase.

1.2 Software Development Phase

Software engineers typically break the software development process into the following four phases:

- Analysis
- Design
- Implementation
- Testing and debugging

The next few sections describe these four phases in some detail.

1.2.1 Analysis

Analyzing the problem is the first and most important step. This step requires you to do the following:

- Thoroughly understand the problem.
- Understand the problem requirements. Requirements can include whether the program requires interaction with the user, whether it manipulates data, whether it produces output, and what the output looks like. Suppose that you need to develop a program to make an automated teller machine (ATM) operational. In the analysis phase, you determine the functionality of the machine. Here, you determine the necessary operations performed by the machine, such as withdraw money, deposit money, transfer money, check account balance, and so on. During this phase, you also talk to potential customers who would use the machine. To make it user-friendly, you must understand their requirements and add any necessary operations. If the program manipulates

data, the programmer must know what the data is and how it is represented. That is, you need to look at sample data. If the program produces output, you should know how the results should be generated and formatted.

- If the problem is complex, divide the problem into subproblems, analyse each subproblem, and understand each subproblem's requirements.

1.2.2 Design

After you carefully analyze the problem, the next step is to design an algorithm to solve the problem. If you broke the problem into subproblems, you need to design an algorithm for each subproblem.

Algorithm: A step-by-step problem-solving process in which a solution is arrived at in a finite amount of time.

1.2.3 Structured Design

Dividing a problem into smaller subproblems is called structured design. The structured design approach is also known as top-down design, stepwise refinement, and modular programming. In structured design, the problem is divided into smaller problems. Each subproblem is then analyzed, and a solution is obtained to solve the subproblem. The solutions of all the subproblems are then combined to solve the overall problem. This process of implementing a structured design is called structured programming.

1.2.4 Object-Oriented Design

In object-oriented design (OOD), the first step in the problem-solving process is to identify the components called objects, which form the basis of the solution, and determine how these objects interact with one another. For example, suppose you want to write a program that automates the video rental process for a local video store. The two main objects in this problem are the video and the customer.

After identifying the objects, the next step is to specify for each object the relevant data and possible operations to be performed on that data. For example, for a video object, the data might include the movie name, starring actors, producer, production company, number of copies in stock, and so on. Some of the operations on a video object might include checking the name of the movie, reducing the number of copies in stock by one after a copy is rented, and incrementing the number of copies in stock by one after a customer returns a particular video.

This illustrates that each object consists of data and operations on that data. An object combines data and operations on the data into a single unit. In OOD, the final program is a collection of interacting objects. A programming language that implements OOD is called an object-oriented programming (OOP) language. You will learn about the many advantages of OOD in later chapters.

OOD has the following three basic principles:

- Encapsulation—The ability to combine data and operations in a single unit
- Inheritance—The ability to create new (data) types from existing (data) types
- Polymorphism—The ability to use the same expression to denote different operations

In C++, encapsulation is accomplished via the use of data types called classes. How classes are implemented in C++ is described later in this chapter. Chapter 2 discusses inheritance and polymorphism.

In object-oriented design, you decide what classes you need and their relevant data members and member functions. You then describe how classes interact with each other.

1.2.5 Implementation

In the implementation phase, you write and compile programming code to implement the classes and functions that were discovered in the design phase.

This book uses the OOD technique (in conjunction with structured programming) to solve a particular

problem. It contains many case studies—called Programming Examples—to solve real-world problems.

The final program consists of several functions, each accomplishing a specific goal. Some functions are part of the main program; others are used to implement various operations on objects. Clearly, functions interact with each other, taking advantage of each other's capabilities. To use a function, the user needs to know only how to use the function and what the function does. The user should not be concerned with the details of the function, that is, how the function is written. Let us illustrate this with the help of the following example.

Suppose that you want to write a function that converts a measurement given in inches into equivalent centimeters. The conversion formula is 1 inch = 2.54 centimeters. The following function accomplishes the job:

```
double inchesToCentimeters(double inches)
{
if (inches < 0)
{
        cerr << "The given measurement must be nonnegative." << endl; return
        -1.0;
}
else
        return 2.54 * inches;
    }
```

If you look at the body of the function, you can recognize that if the value of inches is less than 0, that is, negative, the function returns –1.0; otherwise, the function returns the equivalent length in centimeters. The user of this function does not need to know the specific details of how the algorithm that finds the equivalent length in centimeters is implemented. However, the user must know that in order to get the valid answer, the input must be a nonnegative number. If the input to this function is a negative number, the program returns –1.0. This information can be provided as part of the documentation of this function using specific statements, called preconditions and postconditions.

Precondition: A statement specifying the condition(s) that must be true before the function is called. Postcondition: A statement specifying what is true after the function call is completed. The precondition and postcondition for the function inchesToCentimeters can be specified as follows:

```
//Precondition: The value of inches must be nonnegative.
//Postcondition: If the value of inches is < 0, the function // returns -1.0; otherwise, the function returns the // equivalent length in centimeters. double inchesToCentimeters(double inches)
{
if (inches < 0)
{
        cerr << "The given measurement must be nonnegative." << endl; return
        -1.0;
}
else
        return 2.54 * inches;
    }
```

In certain situations, you could use C++'s assert statement to validate the input. For example, the preceding function can be written as follows:

```
//Precondition: The value of inches must be nonnegative.
//Postcondition: If the value of inches is < 0, the function // terminates; otherwise, the function
returns the // equivalent length in centimeters. double inchesToCentimeters(double inches)
{ assert(inches >= 0);
return 2.54 * inches;
}
```

However, if the assert statement fails, the entire program will terminate, which might be appropriate if the remainder of the program depends on the execution of the function. On the other hand, the user can check the value returned by the function, determine if the returned value is appropriate, and proceed accordingly. To use the assert function, you need to include the header file cassert in your program.

As you can see, the same function can be implemented differently by different programmers. Because the user of a function need not be concerned with the details of the function, the preconditions and postconditions are specified with the function prototype.

That is, the user is given the following information:

```
double inchesToCentimeters(double inches);
//Precondition: The value of inches must be nonnegative.
//Postcondition: If the value of inches is < 0, the function // returns -1.0; otherwise, the function
returns the // equivalent length in centimeters.
```

As another example, to use a function that searches a list for a specific item, the list must exist before the function is called. After the search is complete, the function returns true or false depending on whether the search was successful.

bool search(int list[], int listLength, int searchItem); //Precondition: The list must exist.
//Postcondition: The function returns true if searchItem is in // list; otherwise, the function return false.

1.2.6 Testing and Debugging

The term testing refers to testing the correctness of the program; that is, making sure that the program does what it is supposed to do. The term debugging refers to finding and fixing the errors, if they exist.

Once a function and/or an algorithm is written, the next step is to verify that it works properly. However, in a large and complex program, errors almost certainly exist. Therefore, to increase the reliability of the program, errors must be discovered and fixed before the program is released to the user.

You can certainly prove this by using some (perhaps mathematical) analysis of the correctness of a program. However, for large and complex programs, this technique alone might not be enough because errors can be made in the proof. Therefore, we also rely on testing to determine the quality of the program. The program is run through a series of specific tests, called test cases, in an attempt to find problems.

A test case consists of a set of inputs, user actions, or other initial conditions, and the expected output. Because a test case can be repeated several times, it must be properly documented. Typically a program manipulates a large set of data. It is, therefore, impractical (although possible) to create test cases for all possible inputs. For example, suppose that a program manipulates integers. Clearly, it is not possible to create a test case for each integer. You can categorize test cases into separate categories, called equivalence categories. An equivalence category is a set of input values that are likely to produce the same output. For example, suppose that you have a function that takes an integer as input and returns true if the integer is nonnegative, and false otherwise. In this case, you can form two equivalence categories—one consisting of negative numbers and the other consisting of nonnegative numbers.

There are two types of testing—black-box testing and white-box testing. In black-box testing, you do not know the internal working of the algorithm or function. You know only what the function does. Blackbox testing is based on inputs and outputs. The test cases for black-box testing are usually selected by creating equivalence categories. If a function works for one input in the equivalence category, it is expected to work for other inputs in the same category.

Suppose that the function isWithInRange returns a value true if an integer is greater than or equal to 0 and less than or equal to 100. In black-box testing, the function is tested on values that surround and fall on the boundaries, called boundary values, as well as general values from the equivalence categories. For the function isWithInRange, in black-box testing, the boundary values might be: -1, 0, 1, 99, 100, and 101; and so the test values might be -500, -1, 0, 1, 50, 99, 100, 101, and 500.

White-box testing relies on the internal structure and implementation of a function or algorithm. The objective is to ensure that every part of the function or algorithm is executed at least once. Suppose that you want to ensure whether an if statement works properly. The test cases must consist of at least one input for which the if statement evaluates to true and at least one case for which it evaluates to false. Loops and other structures can be tested similarly.

1.3 Algorithm Analysis: The Big-O Notation

Just as a problem is analysed before writing the algorithm and the computer program, after an algorithm is designed it should also be analysed. Usually, there are various ways to design a particular algorithm. Certain algorithms take very little computer time to execute, whereas others take a considerable amount of time.

Let us consider the following problem. The holiday season is approaching and a gift shop is expecting sales to be double or even triple the regular amount. They have hired extra delivery people to deliver the packages on time. The company calculates the shortest distance from the shop to a particular destination and hands the route to the driver. Suppose that 50 packages are to be delivered to 50 different houses. The shop, while making the route, finds that the 50 houses are one mile apart and are in the same area. (See Figure 1-1, in which each dot represents a house and the distance between houses is 1 mile.)



FIGURE 1-1 Gift shop and each dot representing a house

To deliver 50 packages to their destinations, one of the drivers picks up all 50 packages, drives one mile to the first house and delivers the first package. Then he drives another mile and delivers the second package, drives another mile and delivers the third package, and so on. Figure 1-2 illustrates this delivery scheme.



FIGURE 1-2 Package delivering scheme

It now follows that using this scheme, the distance driven by the driver to deliver the packages is:

1 + 1 + 1 + … + 1 = 50 miles

Therefore, the total distance travelled by the driver to deliver the packages and then getting back to the shop is:

50 + 50 = 100 miles

Another driver has a similar route to deliver another set of 50 packages. The driver looks at the route and delivers the packages as follows: The driver picks up the first package, drives one mile to the first house, delivers the package, and then comes back to the shop. Next, the driver picks up the second package, drives 2 miles, delivers the second package, and then returns to the shop. The driver then picks up the third package, drives 3 miles, delivers the package, and comes back to the shop. Figure 1-3 illustrates this delivery scheme.



FIGURE 1-3 Another package delivery scheme

The driver delivers only one package at a time. After delivering a package, the driver comes back to the shop to pick up and deliver the second package. Using this scheme, the total distance travelled by this driver to deliver the packages and then getting back to the store is:

$2 . (1 + 2 + 3 + . . . + 50) = 2550$ miles

Now suppose that there are n packages to be delivered to n houses, and each house is one mile apart from each other, as shown in Figure 1-1. If the packages are delivered using the first scheme, the following equation gives the total distance travelled:

$$1 + 1 + . . . + 1 + n = 2n \quad (1\text{-}1)$$

If the packages are delivered using the second method, the distance travelled is:

$$2 . (1 + 2 + 3 + . . . + n) = 2 . (n (n + 1)/2) = n^2 + n \quad (1\text{-}2)$$

In Equation (1-1), we say that the distance travelled is a function of n. Let us consider Equation (1-2). In this equation, for large values of n, we will find that the term consisting of n^2 will become the dominant term and the term containing n will be negligible. In this case, we say that the distance travelled is a function of n^2. Table 1-1 evaluates Equations (1-1) and (1-2) for certain values of n. (The table also shows the value of n^2.)

TABLE 1-1: Various values of n, 2n, n^2, and $n^2 + n$

n	$2n$	n^2	$n^2 + n$
1	2	1	2
10	20	100	110
100	200	10,000	10,100
1000	2000	1,000,000	1,001,000
10,000	20,000	100,000,000	100,010,000

While analysing a particular algorithm, we usually count the number of operations performed by the algorithm. We focus on the number of operations, not on the actual computer time to execute the algorithm. This is because a particular algorithm can be implemented on a variety of computers and the speed of the computer can affect the execution time. However, the number of operations performed by

the algorithm would be the same on each computer. Let us consider the following examples.

EXAMPLE 1-1

Consider the following algorithm. (Assume that all variables are properly declared.)

```
cout << "Enter two numbers";      //Line 1 cin >> num1 >> num2;
//Line 2 if (num1 >= num2)     //Line 3 max = num1;     //Line 4  else
//Line 5 max = num2;   //Line 6 cout << "The maximum number is:
" << max << endl;  //Line 7
```

Line 1 has one operation, <<; Line 2 has two operations; Line 3 has one operation, >=; Line 4 has one operation, =; Line 6 has one operation; and Line 7 has three operations. Either Line 4 or Line 6 executes. Therefore, the total number of operations executed in the preceding code is 1 + 2 + 1 + 1 + 3 = 8. In this algorithm, the number of operations executed is fixed.

EXAMPLE 1-2

Consider the following algorithm:

```
cout << "Enter positive integers ending with -1" << endl;     //Line 1
count = 0;                                                    //Line 2
sum = 0;                                                      //Line 3
cin >> num;                                                   //Line 4
while (num != -1)                                             //Line 5
{
        sum = sum + num;                                      //Line 6
        count++;                                              //Line 7
        cin >> num;                                           //Line 8
}
cout << "The sum of the numbers is: " << sum << endl;        //Line 9
if (count != 0)                                               //Line 10
        average = sum / count;                                //Line 11
else                                                          //Line 12
        average = 0;                                          //Line 13
cout << "The average is: " << average << endl;                //Line 14
```

This algorithm has five operations (Lines 1 through 4) before the while loop. Similarly, there are nine or eight operations after the while loop, depending on whether Line 11 or Line 13 executes.

Line 5 has one operation, and four operations within the while loop (Lines 6 through 8). Thus, Lines 5 through 8 have five operations. If the while loop executes 10 times, these five operations execute 10 times. One extra operation is also executed at Line 5 to terminate the loop. Therefore, the number of operations executed is 51 from Lines 5 through 8.

If the while loop executes 10 times, the total number of operations executed is:
10 . 5 + 1 + 5 + 9 or 10 . 5 + 1 + 5 + 8
that is,

10 . 5 + 15 or 10 . 5 + 14

We can generalize it to the case when the while loop executes n times. If the while loop executes n times, the number of operations executed is:

5n + 15 or 5n + 14
In these expressions, for very large values of n, the term 5n becomes the dominating term and the terms 15 and 14 become negligible.

Usually, in an algorithm, certain operations are dominant. For example, in the preceding algorithm, to add numbers, the dominant operation is in Line 6. Similarly, in a search algorithm, because the search item is compared with the items in the list, the dominant operations would be comparison, that is, the relational operation. Therefore, in the case of a search algorithm, we count the number of comparisons. For another example, suppose that we write a program to multiply matrices. The multiplication of matrices involves addition and multiplication. Because multiplication takes more computer time to execute, to analyse a matrix multiplication algorithm, we count the number of multiplications.

In addition to developing algorithms, we also provide a reasonable analysis of each algorithm. If there are various algorithms to accomplish a particular task, the algorithm analysis allows the programmer to choose between various options.

Suppose that an algorithm performs f(n) basic operations to accomplish a task, where n is the size of the problem. Suppose that you want to determine whether an item is in a list. Moreover, suppose that the size of the list is n. To determine whether the item is in the list, there are various algorithms, as you will see in Chapter 9. However, the basic method is to compare the item with the items in the list. Therefore, the performance of the algorithm depends on the number of comparisons.

Thus, in the case of a search, n is the size of the list and f(n) becomes the count function, that is, f(n) gives the number of comparisons done by the search algorithm. Suppose that, on a particular computer, it takes c units of computer time to execute one operation. Thus, the computer time it would take to execute f(n) operations is cf(n). Clearly, the constant c depends on the speed of the computer and, therefore, varies from computer to computer. However, f(n), the number of basic operations, is the same on each computer. If we know how the function f(n) grows as the size of the problem grows, we can determine the efficiency of the algorithm. Consider Table 1-2.

TABLE 1-2: Growth rates of various functions

n	$\log_2 n$	$n \log_2 n$	n^2	2^n
1	0	0	1	2
2	1	2	2	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65,536
32	5	160	1024	4,294,967,296

Table 1-2 shows how certain functions grow as the parameter n, that is, the problem size, grows. Suppose that the problem size is doubled. From Table 1-2, it follows that if the number of basic operations is a function of $f(n) = n^2$, the number of basic operations is quadrupled. If the number of basic operations is a function of $f(n) = 2^n$, the number of basic operations is squared. However, if the number of operations is a function of $f(n) = \log_2 n$, the change in the number of basic operations is insignificant. Suppose that a computer can execute 1 billion basic operations per second. Table 1-3 shows the time that the computer takes to execute f(n) basic operations.

TABLE 1-3: Time for f(n) instructions on a computer that executes 1 billion instructions per second

n	$f(n) = n$	$f(n) = \log_2 n$	$f(n) = n\log_2 n$	$f(n) = n^2$	$f(n) = 2^n$
10	0.01μs	0.003μs	0.033μs	0.1μs	1μs
20	0.02μs	0.004μs	0.086μs	0.4μs	1ms
30	0.03μs	0.005μs	0.147μs	0.9μs	1s
40	0.04μs	0.005μs	0.213μs	1.6μs	18.3min
50	0.05μs	0.006μs	0.282μs	2.5μs	13 days
100	0.10μs	0.007μs	0.664μs	10μs	4×10^{13} years
1000	1.00μs	0.010μs	9.966μs	1ms	
10,000	10μs	0.013μs	130μs	100ms	
100,000	0.10ms	0.017μs	1.67ms	10s	
1,000,000	1 ms	0.020μs	19.93ms	16.7m	
10,000,000	0.01s	0.023μs	0.23s	1.16 days	
100,000,000	0.10s	0.027μs	2.66s	115.7 days	

In Table 1-3, 1ms = 10^{-6} seconds and 1ms = 10^{-3} seconds.

Figure 1-4 shows the growth rate of functions in Table 1-3.

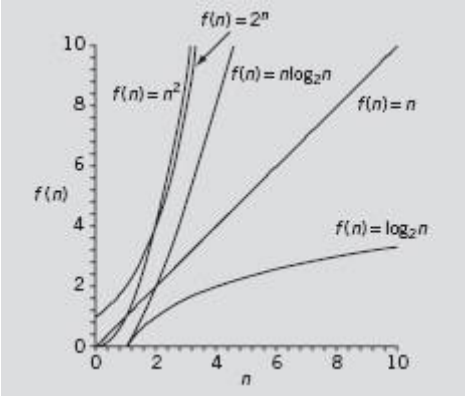


FIGURE 1-4 Growth rate of various functions

The remainder of this section develops a notation that shows how a function f(n) grows as n increases without bound. That is, we develop a notation that is useful in describing the behavior of the algorithm, which gives us the most useful information about the algorithm. First, we define the term asymptotic.

Let f be a function of n. By the term asymptotic, we mean the study of the function f as n becomes larger and larger without bound.

Consider the functions $g(n) = n^2$ and $f(n) = n^2 + 4n + 20$. Clearly, the function g does not contain any linear term, that is, the coefficient of n in g is zero. Consider Table 1-4.

TABLE 1-4: Growth rate of n^2 and $n^2 + 4n + 20$

n	$g(n) = n^2$	$f(n) = n^2 + 4n + 20$
10	100	160
50	2500	2720
100	10,000	10,420
1000	1,000,000	1,004,020
10,000	100,000,000	100,040,020

Clearly, as n becomes larger and larger the term 4n + 20 in f(n) becomes insignificant, and the term n^2 becomes the dominant term. For large values of n, we can predict the behavior of f(n) by looking at the behavior of g(n). In algorithm analysis, if the complexity of a function can be described by the complexity of a quadratic function without the linear term, we say that the function is of $O(n^2)$, called Big-O of n^2.

Let f and g be real-valued functions. Assume that f and g are nonnegative, that is, for all real numbers n, $f(n) >= 0$ and $g(n) >= 0$.

Definition: We say that f(n) is Big-O of g(n), written $f(n) = O(g(n))$, if there exists positive constants c and n_0 such that $f(n) <= cg(n)$ for all $n >= n_0$.

EXAMPLE 1-3

Let $f(n) = a$, where a is a nonnegative real number and $n \geq 0$. Note that f is a constant function. Now

$f(n) = a \leq a \cdot 1$ for all $n \geq a$.

Let $c = a$, $n_0 = a$, and $g(n) = 1$. Then $f(n) \leq cg(n)$ for all $n \geq n_0$. It now follows that $f(n) = O(g(n)) = O(1)$.

From Example 1-3, it follows that if f is a nonnegative constant function, then f is O(1).

EXAMPLE 1-4

Let $f(n) = 2n + 5$, $n \geq 0$. Note that

$f(n) = 2n + 5 \leq 2n + n = 3n$ for all $n \geq 5$.

Let $c = 3$, $n_0 = 5$, and $g(n) = n$. Then $f(n) \leq cg(n)$ for all $n \geq 5$. It now follows that $f(n) = O(g(n)) = O(n)$.

EXAMPLE 1-5

Let $f(n) = n^2 + 3n + 2$, $g(n) = n^2$, $n \geq 0$. Note that $3n + 2 \leq n^2$ for all $n \geq 4$. This implies that

$f(n) = n^2 + 3n + 2 \leq n^2 + n^2 \leq 2n^2 = 2g(n)$ for all $n \geq 4$.

Let $c = 2$ and $n_0 = 4$. Then $f(n) \leq cg(n)$ for all $n \geq 4$. It now follows that $f(n) = O(g(n)) = O(n^2)$.

In general, we can prove the following theorem. Here we state the theorem without proof. Theorem:

Let f(n) be a nonnegative real-valued function such that:

$f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0$,

where a_i's are real numbers, $a_m \neq 0$, $n \geq 0$, and m is a nonnegative integer. Then $f(n) = O(n^m)$.

In Example 1-6, we use the preceding theorem to establish the Big-O of certain functions.

EXAMPLE 1-6

In the following, f(n) is a nonnegative real-valued function.

Function	Big-O
$f(n) = an + b$, where a and b are real numbers and a is nonzero.	$f(n) = O(n)$
$f(n) = n^2 + 5n + 1$	$f(n) = O(n^2)$
$f(n) = 4n^6 + 3n^3 + 1$	$f(n) = O(n^6)$
$f(n) = 10n^7 + 23$	$f(n) = O(n^7)$
$f(n) = 6n^{15}$	$f(n) = O(n^{15})$

EXAMPLE 1-7

Suppose that $f(n) = 2\log_2 n + a$, where a is a real number. It can be shown that $f(n) = O(\log_2 n)$.

EXAMPLE 1-8

Consider the following code, where m and n are int variables and their values are nonnegative:

```
for (int i = 0; i < m; i++)          //Line 1
    for (int j = 0; j < n; j++)      //Line 2
    cout << i * j << endl;  //Line 3
```

This code contains nested for loops. The outer for loop, at Line 1, executes m times. For each iteration of the outer loop, the inner loop, at Line 2, executes n times. For each iteration of the inner loop, the output statement in Line 3 executes. It follows that the total number of iterations of the nested for loop is mn. So the number of times the statement in Line 3 executes is mn. Therefore, this algorithm is O(mn). Note that if m = n, then this algorithm is $O(n^2)$.

Table 1-5 shows some common Big-O functions that appear in the algorithm analysis. Let f(n) = O(g(n)) where n is the problem size.

TABLE 1-5: Some Big-O functions that appear in algorithm analysis

Function $g(n)$	Growth rate of $f(n)$
$g(n) = 1$	The growth rate is constant and so does not depend on n, the size of the problem.
$g(n) = \log_2 n$	The growth rate is a function of $\log_2 n$. Because a logarithm function grows slowly, the growth rate of the function f is also slow.
$g(n) = n$	The growth rate is linear. The growth rate of f is directly proportional to the size of the problem.
$g(n) = n\log_2 n$	The growth rate is faster than the linear algorithm.
$g(n) = n^2$	The growth rate of such functions increases rapidly with the size of the problem. The growth rate is quadrupled when the problem size is doubled.
$g(n) = 2^n$	The growth rate is exponential. The growth rate is squared when the problem size is doubled.

1.4 Classes

In this section, we review C++ classes. If you are familiar with how classes are implemented in C++, you can skip this section.

Recall that in OOD, the first step is to identify the components called objects; an object combines data and the operations on that data in a single unit, called encapsulation. In C++, the mechanism that allows you to combine data and the operations on that data in a single unit is called a class. A class is a collection of a fixed number of components. The components of a class are called the members of the class.

The general syntax for defining a class is:

```
class classIdentifier
{
    class members list
};
```

where class members list consists of variable declarations and/or functions. That is, a member of a class can be either a variable (to store data) or a function.

- If a member of a class is a variable, you declare it just like any other variable. Furthermore, in the definition of the class, you cannot initialize a variable when you declare it.
- If a member of a class is a function, you typically use the function prototype to define that member.
- If a member of a class is a function, it can (directly) access any member of the class—data members and function members. That is, when you write the definition of the member function, you can directly access any data member of the class without passing it as a parameter. The only obvious condition is that you must declare an identifier before you can use it.

In C++, class is a reserved word, and it defines only a data type; no memory is allocated. It announces the declaration of a class. Moreover, note the semicolon (;) after the right brace. The semicolon is part of the syntax. A missing semicolon, therefore, will result in a syntax error.

The members of a class are classified into three categories: private, public, and protected, called member access specifiers. This chapter mainly discusses the first two types—that is, private and public.

Following are some facts about private and public members of a class:

- By default, all members of a class are private.
- If a member of a class is private, you cannot access it outside the class.
- A public member is accessible outside the class.
- To make a member of a class public, you use the member access specifier public with a colon.

In C++, private, protected, and public are reserved words.

1.5 EXAMPLE 1-9

Suppose that we want to define a class, clockType, to implement the time of day in a program. Furthermore, suppose that the time is represented as a set of three integers: one to represent the hours, one to represent the minutes, and one to represent the seconds. We also want to perform the following operations on the time:

1. Set the time.
2. Return the time.
3. Print the time.
4. Increment the time by one second.
5. Increment the time by one minute.
6. Increment the time by one hour.
7. Compare two times for equality.

From this discussion, it is clear that the class clockType has 10 members: three data members and seven function members.

Some members of the class clockType will be private; others will be public. Deciding which member to make private and which to make public depends on the nature of the member. The general rule is that any member that needs to be accessed outside the class is declared public; any member that should not

be accessed directly by the user should be declared private. For example, the user should be able to set the time and print the time. Therefore, the members that set the time and print the time should be declared public.

Similarly, the members to increment the time and compare the time for equality should be declared public. On the other hand, to control the direct manipulation of the data members hr, min, and sec, we will declare these data members private. Furthermore, note that if the user has direct access to the data members, member functions such as setTime are not needed.

The following statements define the class clockType:

```
class clockType
{ public: void setTime(int hours, int minutes, int seconds);
                //Function to set the time
                //The time is set according to the parameters
                //Postcondition: hr = hours; min = minutes; sec = seconds
                // The function checks whether the values of hours, //
                minutes, and seconds are valid. If a value is invalid, //
                the default value 0 is assigned.

                void getTime(int& hours, int& minutes, int& seconds) const;
                //Function to return the time
                //Postcondition: hours = hr; minutes = min; seconds = sec

                void printTime() const; //Function
                to print the time //Postcondition:
                Time is printed in the form
                hh:mm:ss.

                void incrementSeconds();
                //Function to increment the time by one second
                //Postcondition: The time is incremented by one second.
                // If the before-increment time is 23:59:59, the time // is
```

```
        reset to 00:00:00.

        void incrementMinutes();
        //Function to increment the time by one minute
        //Postcondition: The time is incremented by one minute.
        // If the before-increment time is 23:59:53, the time // is
        reset to 00:00:53.

        void incrementHours();
        //Function to increment the time by one hour
        //Postcondition: The time is incremented by one
        hour. // If the before-increment time is 23:45:53, the
        time // is reset to 00:45:53.

        bool equalTime(const clockType& otherClock) const;
        //Function to compare the two times
        //Postcondition: Returns true if this time is equal to
        // otherClock; otherwise, returns false

    private:
        int hr;                 //stores the hours int
        min;            //store the minutes int sec;
                //store the seconds
};
```

We note the following in the definition of the class clockType:

- The class clockType has seven function members: setTime, getTime, printTime, incrementSeconds, incrementMinutes, incrementHours, and equalTime. It has three data members: hr, min, and sec.
- The three data members—hr, min, and sec—are private to the class and cannot be accessed outside the class.
- The seven function members—setTime, getTime, printTime, incrementSeconds, incrementMinutes, incrementHours, and equalTime—can directly access the data members (hr, min, and sec). In other words, we do not pass data members as parameters to member functions.
- In the function equalTime, the parameter otherClock is a constant reference parameter. That is, in a call to the function equalTime, the parameter otherClock receives the address of the actual parameter, but otherClock cannot modify the value of the actual parameter. You could have declared otherClock as a value parameter, but that would require otherClock to copy the value of the actual parameter, which could result in poor performance. (For an explanation, see the section, ‘‘Reference Parameters and Class Objects (Variables)’’ located later in this chapter.)
- The word const at the end of the member functions getTime, printTime, and equalTime specifies that these functions cannot modify the data members of a variable of type clockType.

Note that we have not yet written the definitions of the function members of the class clockType. You will learn how to write them shortly.

The function setTime sets the three data members—hr, min, and sec—to a given value. The given values are passed as parameters to the function setTime. The function printTime prints the time, that is, the values of hr, min, and sec. The function incrementSeconds increments the time by one second, the function incrementMinutes increments the time by one minute, the function incrementHours increments the time by one hour, and the function equalTime compares the two times for equality.

1.6 Constructors

C++ does not automatically initialize variables when they are declared. Therefore, when an object is instantiated, there is no guarantee that the data members of the object will beinitialized. To guarantee that the instance variables of a class are initialized, you use constructors. There are two types of constructors: with parameters and without parameters. The constructor without parameters is called the default constructor.

Constructors have the following properties:

- The name of a constructor is the same as the name of the class.
- A constructor, even though it is a function, has no type. That is, it is neither a value-returning function nor a void function.
- A class can have more than one constructor. However, all constructors of a class have the same name.
- If a class has more than one constructor, the constructors must have different formal parameter lists. That is, either they have a different number of formal parameters or, if the number of formal parameters is the same, the data type of the formal parameters, in the order you list, must differ in at least one position.
- Constructors execute automatically when a class object enters its scope. Because they have no types, they cannot be called like other functions.
- Which constructor executes depends on the types of values passed to the class object when the class object is declared.

Let us extend the definition of the class clockType by including two constructors:

```
class clockType
{ public:
        //Place the function prototypes of the functions setTime, //getTime,
        printTime, incrementSeconds, incrementMinutes,
        //incrementHours, and equalTime as described earlier, here.

        clockType(int hours, int minutes, int seconds);
        //Constructor with parameters
        //The time is set according to the parameters.
        //Postconditions: hr = hours; min = minutes; sec = seconds
        // The constructor checks whether the values of hours,
        // minutes, and seconds are valid. If a value is invalid, //
        the default value 0 is assigned.

        clockType();
        //Default constructor with parameters //The
```

```
                time is set to 00:00:00.
                //Postcondition: hr = 0; min = 0; sec = 0 private:
                int hr;   //stores the hours int min;
                //store the minutes int sec;    //store
                the seconds
};
```

1.6.1 Unified Modeling Language Diagrams

A class and its members can be described graphically using a notation known as Unified Modeling Language (UML) notation. For example, Figure 1-5 shows the UML class diagram of the class clockType.

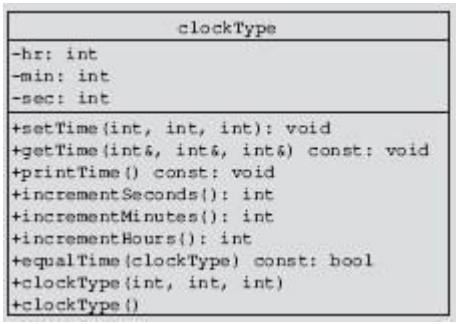


FIGURE 1-5 UML class diagram of the class clockType

The top box contains the name of the class. The middle box contains the data members and their data types. The last box contains the member function name, parameter list, and the return type of the function. A + (plus) sign in front of a member indicates that this member is a public member; a – (minus)

sign indicates that this is a private member. The symbol # before the member name indicates that the member is a protected member.

1.6.2 Variable (Object) Declaration

Once a class is defined, you can declare variables of that type. In C++ terminology, a class variable is called a class object or class instance. To help you become familiar with this terminology, from now on we will use the term class object, or simply object, for a class variable.

A class can have both types of constructors—default constructor and constructors with parameters. Therefore, when you declare a class object, either the default constructor executes or the constructor with parameters executes. The general syntax for declaring a class object that invokes the default constructor is:

```
className classObjectName;
```

For example, the statement

```
clockType myClock;
```

declares myClock to be an object of type clockType. In this case, the default constructor executes and the instance variables of myClock are initialized to 0.

The general syntax for declaring a class object that invokes a constructor with a parameter is

```
className classObjectName(argument1, argument2, ...);
```

where each of argument1, argument2, and so on is either a variable or an expression. Note the following:

- The number of arguments and their type should match the formal parameters (in the order given) of one of the constructors.
- If the type of the arguments does not match the formal parameters of any constructor (in the order given), C++ uses type conversion and looks for the best match. For example, an integer value might be converted to a floating-point value with a zero decimal part. Any ambiguity will result in a

compile-time error.

Consider the following statement:

clockType myClock(5, 12, 40);

This statement declares the object myClock of type clockType. Here, we are passing three values of type int, which matches the type of the formal parameters of the constructor with a parameter. Therefore, the constructor with parameters of the class clockType executes and the three instance variables of the object myClock are set to 5, 12, and 40.

Consider the following statements that declare two objects of type clockType:

clockType myClock(8, 12, 30); clockType yourClock(12, 35, 45);

Each object has 10 members: seven member functions and three instance variables. Each object has separate memory allocated for hr, min, and sec.

In actuality, memory is allocated only for the instance variables of each class object. The C++ compiler generates only one physical copy of a member function of a class, and each class object executes the same copy of the member function.

1.7 Accessing Class Members

Once an object of a class is declared, it can access the members of the class. The general syntax for an object to access a member of a class is:

```
classObjectName.memberName
```

In C++, the dot, . (period), is an operator called the member access operator.

The class members that a class object can access depend on where the object is declared.

- If the object is declared in the definition of a member function of the class, the object can access both the public and private members. (We will elaborate on this when we write the definition of the member function equalTime of the class clockType in the section ‘‘Implementation of Member

Functions,'' later in this chapter.)

- If the object is declared elsewhere (for example, in a user's program), the object can access only the public members of the class.

Example 1-10 illustrates how to access the members of a class.

EXAMPLE 1-10

Suppose we have the following declaration (say, in a user's program):

```
clockType myClock;
clockType yourClock;
```

Consider the following statements:

```
myClock.setTime(5, 2, 30); myClock.printTime();
if (myClock.equalTime(yourClock))
.
.
.
```

These statements are legal; that is, they are syntactically correct.

In the first statement, myClock.setTime(5, 2, 30);, the member function setTime is executed. The values 5, 2, and 30 are passed as parameters to the function setTime, and the function uses these values to set the values of the three instance variables hr, min, and sec of myClock to 5, 2, and 30, respectively.

Similarly, the second statement executes the member function printTime and outputs the contents of the three instance variables of myClock.

In the third statement, the member function equalTime executes and compares the three instance variables of myClock to the corresponding instance variables of yourClock. Because in this statement equalTime is a member of the object myClock, it has direct access to the three instance variables of myClock. So it needs one more object, which in this case is yourClock, to compare. This explains why the function equalTime has only one parameter.

The objects myClock and yourClock can access only public members of the class. Thus, the following statements are illegal because hr and min are declared as private members of the class clockType and, therefore, cannot be accessed by the objects myClock and yourClock:

```
myClock.hr = 10;                    //illegal myClock.min
= yourClock.min;        //illegal
```

1.7.1 Implementation of Member Functions

When we defined the class clockType, we included only the function prototype for the member functions. For these functions to work properly, we must write the related algorithms. One way to implement these functions is to provide the function definition rather than the function prototype in the class itself. Unfortunately, the class definition would then be long and difficult to comprehend. Another reason for providing function prototypes instead of function definitions relates to information hiding; that is, we want to hide the details of the operations on the data.

Next, let us write the definitions of the member functions of the class clockType. That is, we will write the definitions of the functions setTime, getTime, printTime, incrementSeconds, equalTime, and so on. Because the identifiers setTime, printTime, and so forth are local to the class, we cannot reference them (directly) outside the class. To reference these identifiers, we use the scope resolution operator, :: (double colon). In the function definition's heading, the name of the function is the name of the class, followed by

the scope resolution operator, followed by the function name. For example, the definition of the function setTime is as follows:

```
void clockType::setTime(int hours, int minutes, int seconds)
{
        if (0 <= hours && hours <
        24) hr = hours; else
                hr = 0;
        if (0 <= minutes && minutes < 60)
        min = minutes; else min = 0;
        if (0 <= seconds && seconds < 60)
        sec = seconds; else
                sec = 0;
}
```

Note that the definition of the function setTime checks for the valid values of hours, minutes, and seconds. If these values are out of range, the instance variables hr, min, and sec are initialized to 0.

Suppose that myClock is an object of type clockType (as declared previously). The object myClock has three instance variables. Consider the following statement:

myClock.setTime(3, 48, 52);

In the statement myClock.setTime(3, 48, 52);, setTime is accessed by the object myClock. Therefore, the three variables—hr, min, and sec—to which the body of the function setTime refers, are the three instance variables of myClock. Thus, the values 3, 48, and 52, which are passed as parameters in the preceding statement, are assigned to the three instance variables of myClock by the function setTime (see the body of the function setTime). After the previous statement executes, the object myClock is as shown in Figure 1-6.

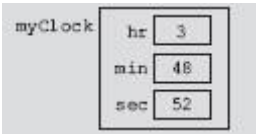


FIGURE 1-6 Object myClock after the statement myClock.setTime(3, 48, 52); executes

Next, let us give the definitions of the other member functions of the class clockType. The definitions of these functions are simple and easy to follow.

```
void clockType::getTime(int& hours, int& minutes, int& seconds) const
{
        hours = hr; minutes
        = min;
        seconds = sec;
}

void clockType::printTime() const
{
        if (hr < 10) cout <<
                "0";
```

```
        cout << hr << ":";

        if  (min  <  10)  cout
                << "0";
        cout << min << ":";

        if (sec < 10) cout <<
                "0";
        cout << sec;
}

void clockType::incrementHours()
{
        hr++; if (hr >
        23) hr = 0;
}

void clockType::incrementMinutes()
{
        min++;      if
        (min > 59)
        {
                min = 0;
                incrementHours();       //increment hours
        }
}

void clockType::incrementSeconds()
{
        sec++;
        if (sec > 59)
        { sec = 0;
                incrementMinutes();  //increment minutes
```

```
        }
}
```

From the definitions of the functions incrementMinutes and incrementSeconds, it is clear that a member function of a class can call other member functions of the class. The function equalTime has the following definition:

```
bool clockType::equalTime(const clockType& otherClock) const
{
        return (hr == otherClock.hr &&
        min == otherClock.min
        && sec == otherClock.sec);
}
```

Let us see how the member function equalTime works.

Suppose that myClock and yourClock are objects of type clockType, as declared previously. Further suppose that we have myClock and yourClock, as shown in Figure 1-7.

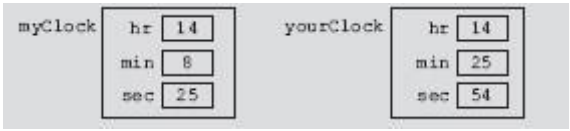


FIGURE 1-7 Objects myClock and yourClock

Consider the following statement:

```
if (myClock.equalTime(yourClock))
.
```

.

.

In the expression

myClock.equalTime(yourClock)

the object myClock accesses the member function equalTime. Because otherClock is a reference parameter, the address of the actual parameter yourClock is passed to the formal parameter otherClock, as shown in Figure 1-8.

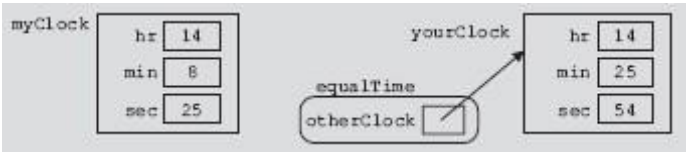


FIGURE 1-8 Object myClock and parameter otherClock

The instance variables hr, min, and sec of otherClock have the values 14, 25, and 54, respectively. In other words, when the body of the function equalTime executes, the value of otherClock.hr is 14, the value of otherClock.min is 25, and the value of otherClock.sec is 54. The function equalTime is a member of myClock. When the function equalTime executes, the variables hr, min, and sec in the body of the function equalTime are the instance variables of the variable myClock. Therefore, the member hr of myClock is compared with otherClock.hr, the member min of myClock is compared with otherClock.min, and the member sec of myClock is compared with otherClock.sec.

Once again, from the definition of the function equalTime, it is clear why this function has only one parameter.

Let us again look at the definition of the function equalTime. Notice that within the definition of this function, the object otherClock accesses the instance variables hr, min, and sec. However, these instance variables are private. So is there any violation? The answer is no. The function equalTime is a member of the class clockType and hr, min, and sec are the instance variables. Moreover, otherClock is an object of type clockType. Therefore, the object otherClock can access its private instance variables within the definition of the function equalTime.

The same is true for any member function of a class. In general, when you write the definition of a member function, say dummyFunction, of a class, say dummyClass, and the function uses an object, dummyObject

of the class dummyClass, then within the definition of dummyFunction, the object dummyObject can access its private instance variables (in fact, any private member of the class).

This definition of the class clockType includes two constructors: one with three parameters and one without any parameters. Let us now write the definitions of these constructors.

```
clockType::clockType()          //default constructor
{ hr = 0; min =
        0;
        sec = 0;
}

clockType::clockType(int hours, int minutes, int seconds)
{
        if (0 <= hours && hours <
        24) hr = hours; else
                hr = 0;
        if (0 <= minutes && minutes < 60)
        min = minutes; else min = 0;
        if (0 <= seconds && seconds < 60)
        sec = seconds; else
                sec = 0;
}
```

From the definitions of these constructors, it follows that the default constructor sets the three instance variables—hr, min, and sec—to 0. Also, the constructor with parameters sets the instance variables to whatever values are assigned to the formal parameters. Moreover, we can write the definition of the constructor with parameters by calling the function setTime, as follows:

```
clockType::clockType(int hours, int minutes, int seconds)
{
        setTime(hours, minutes, seconds);
}
```

Once a class is properly defined and implemented, it can be used in a program. A program or software that uses and manipulates the objects of a class is called a client of that class.

When you declare objects of the class clockType, every object has its own copy of the instance variables hr, min, and sec. In object-oriented terminology, variables such as hr, min, and sec are called instance variables of the class because every object has its own instance of the data.

1.7.2 Reference Parameters and Class Objects (Variables)

Recall that when a variable is passed by value, the formal parameter copies the value of the actual parameter. That is, memory to copy the value of the actual parameter is allocated for the formal parameter. As a parameter, a class object can be passed by value.

Suppose that a class has several instance variables requiring a large amount of memory to store data, and you need to pass a variable by value. The corresponding formal parameter then receives a copy of the data of the variable. That is, the compiler must allocate memory for the formal parameter, so as to copy the value of the instance variables of the actual parameter. This operation might require, in addition to a large amount of storage space, a considerable amount of computer time to copy the value of the actual parameter into the formal parameter.

On the other hand, if a variable is passed by reference, the formal parameter receives only the address of the actual parameter. Therefore, an efficient way to pass a variable as a parameter is by reference. If a variable is passed by reference, then when the formal parameter changes, the actual parameter also changes. Sometimes, however, you do not want the function to be able to change the values of the instance variables. In C++, you can pass a variable by reference and still prevent the function from changing its value by using the keyword const in the formal parameter declaration. As an example, consider the following function definition:

```
void testTime(const clockType& otherClock)
{
        clockType dClock;
        .
        .
        .
}
```

The function testTime contains a reference parameter, otherClock. The parameter otherClock is declared using the keyword const. Thus, in a call to the function testTime, the formal parameter otherClock receives the address of the actual parameter, but otherClock cannot modify the contents of the actual parameter. For example, after the following statement executes, the value of myClock will not be altered:

```
testTime(myClock);
```

Generally, if you want to declare a class object as a value parameter, you declare it as a reference parameter using the keyword const, as described previously. Recall that if a formal parameter is a value parameter, within the function definition you can change the value of the formal parameter. That is, you can use an assignment statement to change the value of the formal parameter (which, of course, would have no effect on the actual parameter). However, if a formal parameter is a constant reference parameter, you cannot use an assignment statement to change its value within the function, nor can you use any other function to change its value. Therefore, within the definition of the function testTime, you cannot alter the value of otherClock. For example, the following would be illegal in the definition of the function testTime:

```
otherClock.setTime(5, 34, 56);          //illegal otherClock
= dClock;                    //illegal
```

1.8 Built-In Operations on Classes

The two built-in operations that are defined for class objects are member access (.) and assignment (=). You have seen how to access an individual member of a class by using the name of the class object, then a dot, and then the member name.

We now show how an assignment statement works with the help of an example.

1.8.1 Assignment Operator and Classes

Suppose that myClock and yourClock are variables of type clockType as defined previously. The statement

```
myClock = yourClock;                    //Line 1
```

copies the value of yourClock into myClock. That is, the value of yourClock.hr is copied into myClock.hr; the value of yourClock.min is copied into myClock.min; and the value of yourClock.sec is copied into myClock.sec. In other words, the values of the three instance variables of yourClock are copied into the corresponding instance variables of myClock. Therefore, an assignment statement performs a memberwise copy.

1.9 Class Scope

A class object can be either automatic (that is, created each time the control reaches its declaration, and destroyed when the control exits the surrounding block) or static (that is, created once, when the control reaches its declaration, and destroyed when the program terminates). Also, you can declare an array of class objects. A class object has the same scope as other variables. A member of a class is local to the class. You access a (public) class member outside the class by using the class object name and the member access operator (.).

1.10 Functions and Classes

The following rules describe the relationship between functions and classes:

- Class objects can be passed as parameters to functions and returned as function values.
- As parameters to functions, class objects can be passed either by value or by reference.
- If a class object is passed by value, the contents of the instance variables of the actual parameter are copied into the corresponding instance variables of the formal parameter.

1.10.1 Constructors and Default Parameters

A constructor can also have default parameters. In such a case, the rules for declaring formal parameters are the same as those for declaring default formal parameters in a function. Moreover, actual parameters to a constructor with default parameters are passed according to the rules for functions with default parameters. Using the rules for defining default parameters, in the definition of the class clockType, you can replace both constructors using the following statement. (Notice that in the function prototype, the name of a formal parameter is optional.)

```
clockType clockType(int = 0, int = 0, int = 0);  //Line 1
```

In the implementation file, the definition of this constructor is the same as the definition of the constructor with parameters.

If you replace the constructors of the class clockType with the constructor in Line 1, (the constructor with the default parameters), you can declare clockType objects with 0, 1, 2, or 3 arguments as follows:

```
clockType clock1;                          //Line 2 clockType
clock2(5);                                 //Line 3
clockType clock3(12, 30);                  //Line 4 clockType
clock4(7, 34, 18);                         //Line 5
```

The data members of clock1 are initialized to 0. The data member hr of clock2 is initialized to 5, and the data members min and sec of clock2 are initialized to 0. The data member hr of clock3 is initialized to 12, the data member min of clock3 is initialized to 30, and the data member sec of clock3 is initialized to 0. The data member hr of clock4 is initialized to 7, the data member min of clock4 is initialized to 34, and the data member sec of clock4 is initialized to 18.

Using these conventions, we can say that a constructor that has no parameters, or has all default parameters, is called the default constructor.

1.10.2 Destructors

Like constructors, destructors are also functions. Moreover, like constructors, a destructor does not have a type. That is, it is neither a value-returning function nor a void function. However, a class can have only

one destructor, and the destructor has no parameters. The name of a destructor is the tilde character (~), followed by the name of the class. For example, the name of the destructor for the class clockType is:

```
~clockType();
```

The destructor automatically executes when the class object goes out of scope.

1.10.3 Structs

Structs are a special type of classes. By default, all members of a class are private, whereas by default all members of a struct are public. In C++, you define structs by using the reserved word struct. If all members of a class are public, C++ programmers prefer to use a struct to group the members, as we will do in this book. A struct is defined just like a class.

1.11 Data Abstraction, Classes, and Abstract Data Types

For the car that we drive, most of us want to know how to start the car and drive it. Most people are not concerned with the complexity of how the engine works. By separating the design details of a car's engine from its use, the manufacturer helps the driver focus on how to drive the car. Our daily life has other similar examples. For the most part, we are concerned only with how to use certain items, rather than with how they work.

Separating the design details (that is, how the car's engine works) from its use is called abstraction. In other words, abstraction focuses on what the engine does and not on how it works. Thus, abstraction is the process of separating the logical properties from the implementation details. Driving the car is a logical property; the construction of the engine constitutes the implementation details. We have an abstract view of what the engine does, but are not interested in the engine's actual implementation.

Abstraction can also be applied to data. Earlier sections of this chapter defined a data type clockType. The data type clockType has three instance variables and the following basic operations:

1. Set the time.
2. Return the time.
3. Print the time.

4. Increment the time by one second.

5. Increment the time by one minute.

6. Increment the time by one hour.

7. Compare two times to see whether they are equal.

The actual implementation of the operations, that is, the definitions of the member functions of the class, clockType was postponed.

Data abstraction is defined as a process of separating the logical properties of the data from its implementation. The definition of clockType and its basic operations are the logical properties; storing clockType objects in the computer, and the algorithms to perform these operations, are the implementation details of clockType.

Abstract data type (ADT): A data type that separates the logical properties from the implementation details.

Like any other data type, an ADT has three things associated with it: the name of the ADT, called the type name; the set of values belonging to the ADT, called the domain; and the set of operations on the data. Following these conventions, we can define the clockType ADT as follows:

dataTypeName clockType

domain

Each clockType value is a time of day in the form of hours, minutes,
and seconds.

operations

Set the time.

Return the time.

Print the time.

Increment the time by one second. Increment
the time by one minute.

Increment the time by one hour.

Compare the two times to see whether they are equal.

To implement an ADT, you must represent the data and write algorithms to perform the operations.

The previous section used classes to group data and functions together. Furthermore, our definition of a class consisted only of the specifications of the operations; functions to implement the operations were written separately. Thus, we see that classes are a convenient way to implement an ADT. In fact, in C++, classes were specifically designed to handle ADTs.

EXAMPLE 1-11

A list is defined as a set of values of the same type. Because all values in a list are of the same type, a convenient way to represent and process a list is to use an array. You can define a list as an ADT as follows:

typeName

listType

domain

Every element of listType is a set of, say at most 1000 numbers.

operations

Check to see whether the list is empty.

Check to see whether the list is full.

Search the list for a given item.

Delete an item from the list.

Insert an item in the list.

Sort the list.

Print the list.

The following class implements the ADT list. To be specific, suppose that the list is a set of elements of the type int.

```
class intListType
{ public:
        bool isEmpty();
        //Function to determine whether the list is empty.
        //Precondition: The list must exist.
        //Postcondition: Returns true if the list is empty, //
```

```
false otherwise.

bool isFull();
//Function to determine whether the list is full.
//Precondition: The list must exist.
//Postcondition: Returns true if the list is full, //
false otherwise.

int search(int searchItem);
//Function to determine whether searchItem is in the list.
//Postcondition: If searchItem is in the list, returns its
// index, that is, its position in the list; // otherwise,
it returns -1.

void insert(int newItem);
//Function to insert newItem in the list.
//Precondition: The list must exist and must not be full.
//Postcondition: newItem is inserted in the list and //
length is incremented by one.

void remove(int removeItem);
//Function to delete removeItem from the list.
//Precondition: The list must exist and must not be empty
//Postcondition: If found, removeItem is deleted from the //
list and the length is decremented by one;
// otherwise, an appropriate message is printed.

void printList();
//Function to output the elements of the list.
//Precondition: The list must exist.
//Postcondition: The elements of the list are //
printed on the standard output device.
```

```
        intListType(); //Default
        constructor
        //Postcondition: length = 0

    private:
        int list[1000]; int
        length;
};
```

EXAMPLE 1-12

The class personType that is designed in Example 1-12 is quite useful; we will use this class in subsequent chapters.

The most common attributes of a person are the person's first name and last name. The typical operations on a person's name are to set the name and print the name. The following statements define a class with these properties.

```
//*************************************************************
// class personType
// This class specifies the members to implement a name.
//*************************************************************
#include <string> using
namespace std;
class personType
{ public: void print() const;
        //Function to output the first name and last name //in
        the form firstName lastName.

        void setName(string first, string last);
        //Function to set firstName and lastName according to the //parameters.
        //Postcondition: firstName = first; lastName = last

        string getFirstName() const;
```

```
        //Function to return the first name.
        //Postcondition: The value of firstName is returned.

        string getLastName() const;
        //Function to return the last name.
        //Postcondition: The value of lastName is returned.

        personType();
        //Default constructor
        //Sets firstName and lastName to null strings.
        //Postcondition: firstName = ""; lastName = "";
        personType(string first, string last); //Constructor
        with parameters.
        //Sets firstName and lastName according to the parameters. //Postcondition:
        firstName = first; lastName = last;

    private:
        string firstName; //variable to store the first name string
        lastName; //variable to store the last name
};
```

Figure 1-9 shows the UML class diagram of the class personType.

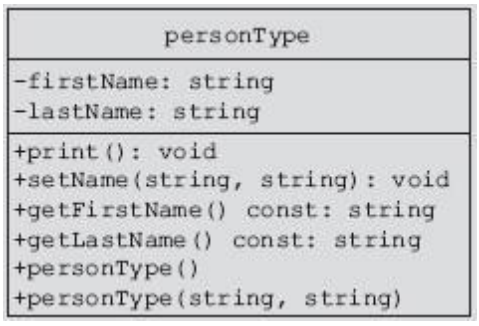


FIGURE 1-9 UML class diagram of the class personType

We now give the definitions of the member functions of the class personType.

```
void personType::print() const
{
    cout << firstName << " " << lastName;
```

```
}

void personType::setName(string first, string last)
{
        firstName = first;
        lastName = last;
}

string personType::getFirstName() const
{
        return firstName;
}

string personType::getLastName() const
{
        return lastName;
}

//Default constructor
personType::personType()
{
        firstName = "";
        lastName = "";
}

//Constructor with parameters
personType::personType(string first, string last)
{
        firstName = first;
        lastName = last;
}
```

1.12 Identifying Classes, Objects, and Operations

The hardest part of OOD is to identify the classes and objects. This section describes a common and simple technique to identify classes and objects.

We begin with a description of the problem and then identify all of the nouns and verbs. From the list of nouns we choose our classes, and from the list of verbs we choose our operations.

For example, suppose that we want to write a program that calculates and prints the volume and surface area of a cylinder. We can state this problem as follows:

Write a program to input the dimensions of a cylinder and calculate and print its surface area and volume.

In this statement, the nouns are bold and the verbs are italic. From the list of nouns— program, dimensions, cylinder, surface area, and volume—we can easily visualize cylinder to be a class—say, cylinderType—from which we can create many cylinder objects of various dimensions. The nouns—dimensions, surface area, and volume— are characteristics of a cylinder and, thus, can hardly be considered classes.

After we identify a class, the next step is to determine three pieces of information:

- Operations that an object of that class type can perform
- Operations that can be performed on an object of that class type Information
- that an object of that class type must maintain

From the list of verbs identified in the problem description, we choose a list of possible operations that an object of that class can perform, or has performed, on itself. For example, from the list of verbs for the cylinder problem description—write, input, calculate, and print—the possible operations for a cylinder object are input, calculate, and print.

For the cylinderType class, the dimensions represent the data. The center of the base, radius of the base, and height of the cylinder are the characteristics of the dimensions. You can input data to the object either by a constructor or by a function.

The verb calculate applies to determining the volume and the surface area. From this, you can deduce the operations: cylinderVolume and cylinderSurfaceArea. Similarly, the verb print applies to the display of the

volume and the surface area on an output device.

Identifying classes via the nouns and verbs from the descriptions to the problem is not the only technique possible. There are several other OOD techniques in the literature. However, this technique is sufficient for the programming exercises in this book.

Key Terms

1. Software are programs run by the computer.
2. A program goes through many phases from the time it is first conceived until the time it is retired, called the life cycle of the program.
3. The three fundamental stages through which a program goes are development, use, and maintenance.
4. The new program is created in the software development stage.
5. In the software maintenance process, the program is modified to fix the (identified) problems and/or to enhance it.
6. A program is retired if no new version of the program will be released.
7. The software development phases are analysis, design, implementation, and testing and debugging.
8. During the design phase, algorithms are designed to solve the problem.
9. An algorithm is a step-by-step problem-solving process in which a solution is arrived at in a finite amount of time.
10. Two well-known design techniques are structured-design and object oriented design.
11. In structured design, a problem is divided into smaller subproblems. Each subproblem is solved, and the solutions of all the subproblems are then combined to solve the problem.
12. In object-oriented design (OOD), a program is a collection of interacting objects.
13. An object consists of data and operations on those data.
14. The three basic principles of OOD are encapsulation, inheritance, and polymorphism.
15. In the implementation phase, you write and compile programming code to implement the classes and functions that were discovered in the design phase.
16. A precondition is a statement specifying the condition(s) that must be true before the function is called.
17. A postcondition is a statement specifying what is true after the function call is completed.
18. During the testing phase, the program is tested for its correctness; that is, for making sure that the program does what it is supposed to do.

19. Debugging refers to finding and fixing the errors, if they exist.
20. To find problems in a program, it is run through a series of test cases.
21. A test case consists of a set of inputs, user actions, or other initial conditions, and the expected output.
22. There are two types of testing—black-box testing and white-box testing.
23. While analyzing a particular algorithm, we usually count the number of operations performed by the algorithm.
24. Let f be a function of n. The term asymptotic refers to the study of the function f as n becomes larger and larger without bound.
25. A class is a collection of a fixed number of components.
26. Components of a class are called the members of the class.
27. Members of a class are accessed by name.
28. In C++, class is a reserved word.
29. Members of a class are classified into one of three categories: private, protected, and public.
30. The private members of a class are not accessible outside the class.
31. The public members of a class are accessible outside the class.
32. By default, all members of a class are private.
33. The public members are declared using the member access specifier public.
34. The private members are declared using the member access specifier private.
35. A member of a class can be a function or a variable (that is, data).
36. If any member of a class is a function, you usually use the function prototype to declare it.
37. If any member of a class is a variable, it is declared like any other variable.
38. In the definition of the class, you cannot initialize a variable when you declare it.
39. In the Unified Modeling language (UML) diagram of a class, the top box contains the name of the class. The middle box contains the data members and their data types. The last box contains the member function name, parameter list, and the return type of the function. A + (plus) sign in front of a member indicates that this member is a public member; a – (minus) sign indicates that this is a private member. The symbol # before the member name indicates that the member is a protected member.
40. In C++, a class is a definition. No memory is allocated; memory is allocated for the class variables when you declare them.
41. In C++, class variables are called class objects or simply objects.
42. A class member is accessed using the class variable name, followed by the dot operator (.), followed by the member name.
43. The only built-in operations on classes are the assignment and member selection.

44. Class objects can be passed as parameters to functions and returned as function values.
45. As parameters to functions, classes can be passed either by value or by reference.
46. Constructors guarantee that the data members are initialized when an object is declared.
47. The name of a constructor is the same as the name of the class.
48. A class can have more than one constructor.
49. A constructor without parameters is called the default constructor.
50. Constructors automatically execute when a class object enters its scope.
51. Destructors automatically execute when a class object goes out of scope.
52. A class can have only one destructor with no parameters.
53. The name of a destructor is the tilde (~), followed by the class name (no spaces in between).
54. Constructors and destructors are functions without any type; that is, they are neither valuereturning nor void. As a result, they cannot be called like other functions.
55. A data type that specifies the logical properties without the implementation details is called an abstract data type (ADT).
56. An easy way to identify classes, objects, and operations is to describe the problem in English and then identify all of the nouns and verbs. Choose your classes (objects) from the list of nouns and operations from the list of verbs.

Review Questions

1. Mark the following statements as true or false.

a. The life cycle of software refers to the phases from the point the software was conceived until it is retired.

b. The three fundamental stages of software are development, use, and discard.

c. The expression $4n + 2n^2 + 5$ is O(n).

d. The instance variables of a class must be of the same type.

e. The function members of a class must be public.

f. A class can have more than one constructor.

g. A class can have more than one destructor.

h. Both constructors and destructors can have parameters.

2. What is black-box testing?

3. What is white-box testing?

4. Consider the following function prototype, which returns the square root of a real number:

```
double sqrt(double x);
```

What should be the pre- and postconditions for this function?

5. Each of the following expressions represents the number of operations for certain algorithms. What is the order of each of these expressions?

a. $n^2 + 6n + 4$

b. $5n^3 + 2n + 8$

c. $(n^2 + 1)(3n + 5)$

d. $5(6n + 4)$

e. $n + 2\log_2 n - 6$

f. $4n \log_2 n + 3n + 8$

6. Consider the following function:

```
void funcExercise6(int x, int y)
{ int z; z = x + y;
        x = y; y
        = z; z =
        x;
        cout << "x = " << x << ", y = " << y << ", z = " << z << endl;
}
```

Find the exact number of operations executed by the function funcExercise6.

7. Consider the following function:

```
int funcExercise7(int list[], int size)
```

```
{ int sum = 0;
        for (int index = 0; index < size; index++) sum
                = sum + list[index];
        return sum;
}
```

a. Find the number of operations executed by the function funcExercise7 if the value of size is 10.

b. Find the number of operations executed by the function funcExercise7 if the value of size is n.

c. What is the order of the function funcExercise7?

8. Consider the following function prototype:

```
int funcExercise8(int x);
```

The function funcExercise8 returns the value as follows: If 0 <= x <= 50, it returns 2x; if –50 <= x < 0, it returns x2; otherwise it returns –999.What are the reasonable boundary values for the function funcExercise8?

9. Write a function that uses a loop to find the sum of the squares of all integers between 1 and n. What is the order of your function?

10. Characterize the following algorithm in terms of Big-O notation. Also find the exact number of additions executed by the loop. (Assume that all variables are properly declared.)

```
for (int i = 1; i <= n; i++)
        sum = sum + i * (i + 1);
```

11. Characterize the following algorithm in terms of Big-O notation. Also find the exact number of additions, subtractions, and multiplications executed by the loop. (Assume that all variables are properly declared.) for (int i = 5; i <= 2 * n; i++) cout << 2 * n + i - 1 << endl;

12. Characterize the following algorithm in terms of Big-O notation.

```
for (int i = 1; i <= 2 * n; i++) for
```

```
                    (int j = 1; j <= n; j++)
                            cout << 2 * i + j;
            cout << endl;
```

13. Characterize the following algorithm in terms of Big-O notation. for (int i = 1; i <= n; i++) for (int j = 1; j <= n; j++) for (int k = 1; k <= n; k++)

```
                                        cout << i + j + k;
```

14. Find the syntax errors in the definitions of the following classes: a. class AA

```
{ public: void print(); int
        sum();
                AA(); int
        AA(int, int); private:
        int x; int y;
};
```

b. class BB

```
{ int one ; int two;
public: bool equal();
print(); BB(int, int);
}
```

c. class CC

```
{ public; void set(int, int); void
        print(); CC(); CC(int,
        int); bool CC(int, int);
        private: int u;
```

```
        int v;
};
```

15. Consider the following declarations: class

```
xClass
{ public: void func(); void print()
        const; xClass (); xClass (int,
        double); private:
                int u;
                double w;
};

xClass x;
```

a. How many members does class xClass have?

b. How many private members does class xClass have?

c. How many constructors does class xClass have?

d. Write the definition of the member function func so that u is set to 10 and w is set to 15.3.

e. Write the definition of the member function print that prints the contents of u and w.

f. Write the definition of the default constructor of the class xClass so that the private data members are initialized to 0.

g. Write a C++ statement that prints the values of the data members of the object x.

h. Write a C++ statement that declares an object t of the type xClass, and initializes the data members of t to 20 and 35.0, respectively.

16. Consider the definition of the following class: class

```
CC
{ public:
        CC();                   //Line 1
        CC(int);                //Line 2
        CC(int, int);           //Line 3
        CC(double, int);        //Line 4
```

.

.

.

```
    private:
        int u;
        double v;
};
```

a. Give the line number containing the constructor that is executed in each of the following declarations:
   i. CC one; ii. CC two(5, 6); iii. CC three(3.5, 8);
b. Write the definition of the constructor in Line 1 so that the private data members are initialized to 0.
c. Write the definition of the constructor in Line 2 so that the private data member u is initialized according to the value of the parameter, and the private data member v is initialized to 0.
d. Write the definition of the constructors in Lines 3 and 4 so that the private data members are initialized according to the values of the parameters.

17. Given the definition of the class clockType with constructors (as described in this chapter), what is the output of the following C++ code?

```
clockType clock1; clockType
clock2(23, 13, 75);
clock1.printTime(); cout <<
endl; clock2.printTime();
cout << endl;
clock1.setTime(6, 59,
39); clock1.printTime();
cout << endl;
clock1.incrementMinutes();
clock1.printTime(); cout << endl; clock1.setTime(0, 13, 0); if
(clock1.equalTime(clock2)) cout << "Clock1 time is the same as
clock2 time." << endl; else cout << "The two times are
different." << endl;
```

18. Write the definition of a class that has the following properties:

a. The name of the class is secretType.

b. The class secretType has four instance variables: name of type string, age and weight of type int, and

height of type double.

c. The class secretType has the following member functions:

print—Outputs the data stored in the instance variables with the appropriate titles setName—Function to set the name setAge—Function to set the age setWeight—Function to set the weight setHeight—Function to set the height getName—Value-returning function to return the name getAge—Value-returning function to return the age
getWeight—Value-returning function to return the weight getHeight—Value-returning function to return the height
Default constructor—Sets name to the empty string and age, weight, and height to 0
Constructor with parameter—Sets the values of the instance variables to the values specified by the user

d. Write the definition of the member functions of the class secretType as described in Part c.

19. Assume the definition of the class personType as given in this chapter.

a. Write a C++ statement that declares student to be a personType object, and initialize its first name to "Buddy" and last name to "Arora".

b. Write a C++ statement that outputs the data stored in the object student.

c. Write C++ statements that change the first name of student to "Susan" and the last name to "Miller".

2 TOPIC 2: STANDARD TEMPLATE LIBRARY (STL)



LEARNING OUTCOMES:

After studying this topic you should be able to:

- Learn about the Standard Template Library (STL)
- Become familiar with the three basic components of the STL—containers, iterators, and algorithms
- Explore how vector and deque containers are used to manipulate data in a program Discover
- the use of iterators

C++ is equipped with a Standard Template Library (STL). Among other things, the STL provides class templates to process lists (contiguous or linked), stacks, and queues. This chapter discusses some of the STL's important features, and shows how to use certain tools provided by the STL in a program.

In the ensuing chapters, you will learn how to develop your own code to implement and manipulate data, as well as how to use professionally written code.

2.1 Components of the STL

The main objective of a program is to manipulate data and generate results. Achieving this goal requires the ability to store data into computer memory, access a particular piece of data, and write algorithms to manipulate the data.

For example, if all the data items are of the same type and we have some idea of the number of data items, we could use an array to store the data. We can then use an index to access a particular component of the array. Using a loop and the array index, we can step through the elements of the array. Algorithms, such as those for initializing the array, sorting, and searching, are used to manipulate the data stored in an array. On the other hand, if we do not want to worry about the size of the data, we can use a linked list, to process it. If the data needs to be processed in a Last In First Out (LIFO) manner, we can use a stack.

Similarly, if the data needs to be processed in a First In First Out (FIFO) manner, we can use a queue.

The STL is equipped with these features to effectively manipulate data. More formally, the STL has three main components:

- Containers
- Iterators
- Algorithms

Containers and iterators are class templates. Iterators are used to step through the elements of a container. Algorithms are used to manipulate data. This chapter discusses some of the containers and iterators.

2.1 Container Types

Containers are used to manage objects of a given type. The STL containers are classified into three categories:

- Sequence containers (also called sequential containers)
- Associative containers
- Container adapters

Container adapters are described in Chapters 5 and 6.

2.2 Sequence Containers

Every object in a sequence container has a specific position. The three predefined sequence containers are as follows:

- vector
- deque
- list

Before discussing container types in general, let us first briefly describe the sequence container vector.

We do so because vector containers are logically the same as arrays and, therefore, they can be processed like arrays. Also, with the help of vector containers, we can describe several properties that are common to all containers. In fact, all containers use the same names for the common operations. Of course, there are operations that are specific to a container. These operations are discussed when describing a specific container. This chapter discusses vector and deque containers. Chapter 3 discusses list containers.

2.3 Sequence Container: vector

A vector container stores and manages its objects in a dynamic array. Because an array is a random access data structure, the elements of a vector can be accessed randomly. Item insertion in the middle or beginning of an array is time consuming, especially if the array is large. However, inserting an item at the end is fast.

The name of the class that implements the vector container is vector. (Recall that containers are class templates.) The name of the header file containing the class vector is vector. Thus, to use a vector container in a program, the program must include the following statement:

```
#include <vector>
```

Furthermore, to define an object of type vector, we must specify the type of the object because the class vector is a class template. For example, the statement

```
vector<int> intList;
```

declares intList to be a vector and the component type to be int. Similarly, the statement

```
vector<string> stringList;
```

declares stringList to be a vector container and the component type to be string.

2.3.1 Declaring Vector Objects

The class vector contains several constructors, including the default constructor. Therefore, a vector container can be declared and initialized in several ways. Table 2-1 describes how a vector container of a specific type can be declared and initialized.

TABLE 2-1: Various ways to declare and initialize a vector container

Statement	Effect
`vector<elementType> vecList;`	Creates an empty vector, `vecList`, without any elements. (The default constructor is invoked.)
`vector<elementType> vecList(otherVecList);`	Creates a vector, `vecList`, and initializes `vecList` to the elements of the vector `otherVecList`. `vecList` and `otherVecList` are of the same type.
`vector<elementType> vecList(size);`	Creates a vector, `vecList`, of size `size`. `vecList` is initialized using the default constructor.
`vector<elementType> vecList(n, elem);`	Creates a vector, `vecList`, of size `n`. `vecList` is initialized using `n` copies of the element `elem`.
`vector<elementType> vecList(begin, end);`	Creates a vector, `vecList`. `vecList` is initialized to the elements in the range `[begin, end)`, that is, all elements in the range `begin...end-1`.

EXAMPLE 2-1

a. The following statement declares intList to be an empty vector container and the element type is int.
   vector<int> intList;
b. The following statement declares intList to be a vector container of size 10 and the element type is int. The elements of intList are initialized to 0. vector<int> intList(10);
c. The following statement declares intList to be a vector container of size 5 and the element type is int. The container intList is initialized using the elements of the array. int intArray[5] = {2,4,6,8,10};
   vector<int> intList(intArray, intArray + 5);

The container intList is initialized using the elements of the array intArray. That is, intList = {2,4,6,8,10}.

Now that we know how to declare a vector sequence container, let us now discuss how to manipulate data stored in a vector container. To do so, we must know the following basic operations:

- Item insertion
- Item deletion
- Stepping through the elements of a vector container

The elements in a vector container can be accessed directly by using the operations given in Table 2-2.

TABLE 2-2: Operations to access the elements of a vector container

Expression	Effect
`vecList.at(index)`	Returns the element at the position specified by `index`.
`vecList[index]`	Returns the element at the position specified by `index`.
`vecList.front()`	Returns the first element. (Does not check whether the container is empty.)
`vecList.back()`	Returns the last element. (Does not check whether the container is empty.)

From Table 2-2, it follows that that the elements in a vector can be processed just as they can in an array. (Recall that in C++, arrays start at location 0. Similarly, the first element in a vector container is at location 0.)

EXAMPLE 2-2

Consider the following statement, which declares intList to be a vector container of size 5 and the element type is int.

```
vector<int> intList(5);
```

You can use a loop, such as the following, to store elements into intList:

```
for (int j = 0; j < 5; j++)
    intList[j] = j;
```

Similarly, you can use a for loop to output the elements of intList.

The class vector provides various operations to process the elements of a vector container. Suppose that vecList is a container of type vector. Item insertion and deletion in vecList can be accomplished using the operations given in Table 2-3. These operations are implemented as member functions of the class vector and are shown in bold. Table 2-3 also shows how these operations are used.

TABLE 2-3: Various operations on a vector container

Expression	Effect
`vecList.clear()`	Deletes all elements from the container.
`vecList.erase(position)`	Deletes the element at the position specified by position.
`vecList.erase(beg, end)`	Deletes all elements starting at `beg` until `end-1`.
`vecList.insert(position, elem)`	A copy of `elem` is inserted at the position specified by `position`. The position of the new element is returned.
`vecList.insert(position, n, elem)`	`n` copies of `elem` are inserted at the position specified by `position`.
`vecList.insert(position, beg, end)`	A copy of the elements, starting at `beg` until `end-1`, is inserted into `vecList` at the position specified by `position`.
`vecList.push_back(elem)`	A copy of `elem` is inserted into `vecList` at the end.
`vecList.pop_back()`	Deletes the last element.
`vecList.resize(num)`	Changes the number of elements to `num`. If `size()`, that is, the number of elements in the container increases, the default constructor creates the new elements.
`vecList.resize(num, elem)`	Changes the number of elements to `num`. If `size()` increases, the default constructor creates the new elements.

The function push_back is quite useful. This function is used to add an element at the end of a container. The container intList of size 5 was declared in Example 2-2. You might think that you can only add five elements to the container intList. However, this is not the case. If you need to add more than five elements, you can use the function push_back. You cannot use the array subscripting operator, as in Example 2-2, to add elements past the position 4 unless you increase the size of the container.

If you do not know the number of elements you need to store in a vector container, then when you declare the vector container you do not need to specify its size (see Example 2-3). In this case, you can use the function push_back, as shown in Examples 2-3 and 2-5, to add elements into a vector container.

EXAMPLE 2-3

The following statement declares intList to be a vector container of size 0.

```
vector<int> intList;
```

To add elements into intList, we can use the function push_back as follows:

```
intList.push_back(34); intList.push_back(55);
```

After these statements execute, the size of intList is 2 and intList = {34, 55}. Of course, you could have used the resize function to increase the size of intList and then use the array subscripting operator. However, at times, the push_back function is more convenient because it does not need to know the size of the container; it simply adds elements at the end.

2.3.2 Declaring an Iterator to a Vector Container

Even though we can process a vector container just like an array using the array subscripting operator, there are situations where we would like to process the elements of a vector container using an iterator. (Recall that an iterator is just like a pointer.) For example, suppose that we want to insert an element at a specific position in a vector container. Because the element is to be inserted at a specific position, this would require shifting the elements of the container (unless the element is added at the end). Of course, we must also think about the size of the container. To make element(s) insertion convenient, the class vector provides the function insert to insert the elements at a specific position in a vector container. However, to use the function insert, the position where to insert the element(s) must be specified by an iterator. Similarly, the function erase, to remove an element, also requires the use of an iterator. This section describes how to declare and use an iterator on a vector container.

The class vector contains a typedef iterator, which is declared as a public member. An iterator to a vector container is declared using the typedef iterator.
For example, the statement

```
vector<int>::iterator intVecIter;
```

declares intVecIter to be an iterator into a vector container of type int.

Because iterator is a typedef defined inside the class vector, we must use the container name (vector), container element type, and scope resolution operator to use the typedef iterator.

Suppose that the iterator intVecIter points to an element of a vector container whose elements are of type int. The expression

```
++intVecIter
```

advances the iterator intVecIter to the next element into the container. The expression

```
*intVecIter
```

returns the element at the current iterator position.

Note that these operations are the same as the operations on pointers. Recall that when used as a unary operator, * is called the dereferencing operator.

We now discuss how to use an iterator into a vector container to manipulate the data stored into it. Suppose that we have the following statements:

```
vector<int> intList;                    //Line 1 vector<int>::iterator
intVecIter;       //Line 2
```

The statement in Line 1 declares intList to be a vector container and the element type to be int. The statement in Line 2 declares intVecIter to be an iterator into a vector container whose element type is int.

2.4 Containers and the Functions begin and end

Every container contains the member functions begin and end. The function begin returns the position of the first element into the container; the function end returns the position of the last element into the container. These functions have no parameters.

After the following statement executes:

```
intVecIter = intList.begin();
```

the iterator intVecIter points to the first element into the container intList.

The following for loop uses an iterator to output the elements of intList onto the standard output device:

```
for (intVecIter = intList.begin(); intVecIter != intList.end(); intVecList)
                                    cout << *intVecIter << " ";
```

EXAMPLE 2-4

Consider the following statements: int intArray[7] = {1, 3, 5, 7, 9, 11, 13};

```
                    //Line 1 vector<int> vecList(intArray,
intArray + 7};  //Line 2 vector<int>::iterator
intVecIter;                 //Line 3
```

The statement in Line 2 declares and initializes the vector container vecList. Now consider the following statements:

```
intVecIter = vecList.begin();                    //Line 4
++intVecIter;                          //Line  5  vecList.insert(intVecIter,  22};
                    //Line 6
```

The statement in Line 4 initializes the iterator intVecIter to the first element of vecList. The statement in Line 5 advances intVecIter to the second element of vecList. The statement in Line 6 inserts 22 at the position specified by intVecIter. After the statement in Line 6 executes, vecList = {1, 22, 3, 5, 7, 9, 11, 13}. Notice that the size of the container also increases.

The class vector also contains member functions that can be used to find the number of elements currently in the container, the maximum number of elements that can be inserted in a container, and so on. Table 2-4 describes some of these operations. (Suppose that vecCont is a vector container.)

TABLE 2-4: Functions to determine the size of a vector container

Expression	Effect
`vecCont.capacity()`	Returns the maximum number of elements that can be inserted into the container `vecCont` without reallocation.
`vecCont.empty()`	Returns `true` if the container `vecCont` is empty and `false` otherwise.
`vecCont.size()`	Returns the number of elements currently in the container `vecCont`.
`vecCont.max_size()`	Returns the maximum number of elements that can be inserted into the container `vecCont`.

Example 2-5 illustrates how to use a vector container in a program and how to process the elements into a vector container.

EXAMPLE 2-5

```
//*********************************************************** //This
program illustrates how to use a vector container in a program.
//***********************************************************
#include <iostream>                                    //Line 1 #include
<vector>                                               //Line 2 using
namespace std;                                         //Line 3 int main()
                                                       //Line 4
{                                                              //Line 5 vector<int>
        intList;                                       //Line 6

        intList.push_back(13);                                 //Line 7 intList.push_back(75);
                                                //Line 8 intList.push_back(28);
                        //Line 9
        intList.push_back(35);                                 //Line 10
```

```
cout << "Line 11: List Elements: ";                    //Line 11 for
(int i = 0; i < 4; i++)                                 //Line 12 cout <<
intList[i] << " ";                                      //Line 13
cout << endl;                                           //Line 14

for (int i = 0; i < 4; i++)                             //Line 15
intList[i] *= 2;                                        //Line 16 cout <<
"Line 17: List Elements: ";                             //Line 17

for (int i = 0; i < 4; i++)                             //Line 18 cout
<< intList[i] << " ";                                   //Line 19 cout << endl;
                                                        //Line 20

vector<int>::iterator listIt;                           //Line 21
```

```
        cout << "Line 22: List Elements: ";                          //Line 22
        for (listIt = intList.begin(); listIt != intList.end(); ++listIt)   //Line 23
            cout << *listIt << " ";                                 //Line 24
        cout << endl;                                               //Line 25


        listIt = intList.begin();                                   //Line 26
++listIt;                                                   //Line 27 ++listIt;
        //Line 28
        intList.insert(listIt, 88);                                 //Line 29


        cout << "Line 30: List Elements: ";                         //Line 30
        for (listIt = intList.begin(); listIt != intList.end(); ++listIt)   //Line 31
            cout << *listIt << " ";                                 //Line 32
        cout << endl;                                               //Line 33


        return 0;                                                   //Line 34
    }                                                               //Line 35
```

Sample Run:

Line 11: List Elements: 13 75 28 35

Line 17: List Elements: 26 150 56 70

Line 22: List Elements: 26 150 56 70

Line 30: List Elements: 26 150 88 56 70

The statement in Line 6 declares a vector container (or vector for short), intList, of type int. The statements in Lines 7 through 10 use the operation push_back to insert four numbers—13, 75, 28, and 35—into intList. The statements in Lines 12 and 13 use the for loop and the array subscripting operator [] to output the elements of intList. In the output, see the line marked Line 11, which contains the output of Lines 11

through 14 of the program. The statements in Lines 15 and 16 use a for loop to double the value of each element of intList; the statements in Lines 18 and 19 output the elements of intList. In the output, see the line marked Line 17, which contains the output of Lines 17 through 20 of the program.

The statement in Line 21 declares listIt to be a vector iterator that processes any vector container whose elements are of type int. Using the iterator listIt, the statements in Lines 23 and 24 output the elements of intList. After the statement in Line 26 executes, listIt points to the first element of intList. The statements in Lines 27 and 28 advance listIt twice; after these statements execute, listIt points to the third element of intList. The statement in Line 29 inserts 88 into intList at the position specified by the iterator listIt. Because listIt points to the component at position 2 (the third element of intList), 88 is inserted at

position 2 in intList; that is, 88 becomes the third element of intList. The statements in Lines 31 and 32 output the modified intList.

2.4.1 Member Functions Common to All Containers

The previous section discussed vector containers. We now look at operations that are common to all containers. For example, every container class has a default constructor, several constructors with parameters, a destructor, a function to insert an element into a container, and so on.

Recall that a class encapsulates data, and operations on that data, into a single unit. Because every container is a class, several operations are directly defined for a container and are provided as part of the class definition. Also, recall that the operations to manipulate the data are implemented with the help of functions and are called member functions of the class. Table 2-5 describes the member functions that are common to all containers; that is, these functions are included as members of the class template implementing the container. Suppose that ct, ct1, and ct2 are containers of the same type. Table 2-5 shows the name of the function in bold, and shows how a function is called.

TABLE 2-5: Member functions common to all containers

Member function	Effect
Default constructor	Initializes the object to an empty state.
Constructor with parameters	In addition to the default constructor, every container has constructors with parameters. We describe these constructors when we discuss a specific container.
Copy constructor	Executes when an object is passed as a parameter by value, and when an object is declared and initialized using another object of the same type.
Destructor	Executes when the object goes out of scope.
`ct.empty()`	Returns `true` if container `ct` is empty and `false` otherwise.
`ct.size()`	Returns the number of elements currently in container `ct`.
`ct.max_size()`	Returns the maximum number of elements that can be inserted into container `ct`.
`ct1.swap(ct2)`	Swaps the elements of containers `ct1` and `ct2`.
`ct.begin()`	Returns an iterator to the first element into container `ct`.
`ct.end()`	Returns an iterator to the last element into container `ct`.
`ct.rbegin()`	Reverse begin. Returns a pointer to the last element into container `ct`. This function is used to process the elements of `ct` in reverse.
`ct.rend()`	Reverse end. Returns a pointer to the first element into container `ct`.
`ct.insert(position, elem)`	Inserts `elem` into container `ct` at the position specified by the argument `position`. Note that here `position` is an iterator.
`ct.erase(begin, end)`	Deletes all elements between `begin...end-1` from container `ct`.

TABLE 2-5: Member functions common to all containers (continued)

Member function	Effect
`ct.clear()`	Deletes all elements from the container. After a call to this function, container `ct` is empty.
Operator functions	
`ct1 = ct2`	Copies the elements of `ct2` into `ct1`. After this operation, the elements in both containers are the same.
`ct1 == ct2`	Returns `true` if containers `ct1` and `ct2` are equal and `false` otherwise.
`ct1 != ct2`	Returns `true` if containers `ct1` and `ct2` are not equal and `false` otherwise.
`ct1 < ct2`	Returns `true` if container `ct1` is less than container `ct2` and `false` otherwise.
`ct1 <= ct2`	Returns `true` if container `ct1` is less than or equal to container `ct2` and `false` otherwise.
`ct1 > ct2`	Returns `true` if container `ct1` is greater than container `ct2` and `false` otherwise.
`ct1 >= ct2`	Returns `true` if container `ct1` is greater than or equal to container `ct2` and `false` otherwise.

2.4.2 Member Functions Common to Sequence Containers

The previous section described the member functions that are common to all containers. In addition to these member functions, Table 2-6 describes the member functions that are common to all sequence containers—that is, containers of type vector, deque, and list. (Suppose that seqCont is a sequence container.)

TABLE 2-6: Member functions common to all sequence containers

Expression	Effect
`seqCont.insert(position, elem)`	A copy of `elem` is inserted at the position specified by `position`. The position of the new element is returned.
`seqCont.insert(position, n, elem)`	`n` copies of `elem` are inserted at the position specified by `position`.
`seqCont.insert(position, beg, end)`	A copy of the elements, starting at `beg` until `end-1`, are inserted into `seqCont` at the position specified by `position`.
`seqCont.push_back(elem)`	A copy of `elem` is inserted into `seqCont` at the end.
`seqCont.pop_back()`	Deletes the last element.
`seqCont.erase(position)`	Deletes the element at the position specified by `position`.
`seqCont.erase(beg, end)`	Deletes all elements starting at `beg` until `end-1`.
`seqCont.clear()`	Deletes all elements from the container.
`seqCont.resize(num)`	Changes the number of elements to `num`. If `size()` grows, the new elements are created by their default constructor.
`seqCont.resize(num, elem)`	Changes the number of elements to `num`. If `size()` grows, the new elements are copies of `elem`.

2.5 The copy Algorithm

Example 2-5 used a for loop to output the elements of a vector container. The STL provides a convenient way to output the elements of a container with the help of the function copy. The function copy is

provided as a part of the generic algorithms of the STL and can be used with any container type as well as arrays. Because we frequently need to output the elements of a container, before continuing with our discussion of containers, let us describe this function.

The function copy does more than output the elements of a container. In general, it allows us to copy the elements from one place to another. For example, to output the elements of a vector or to copy the elements of a vector into another vector, we can use the function copy. The prototype of the function template copy is as follows:

```
template <class inputIterator, class outputIterator> outputItr copy(inputIterator
        first1, inputIterator last, outputIterator first2);
```

The parameter first1 specifies the position from which to begin copying the elements; the parameter last specifies the end position. The parameter first2 specifies where to copy the elements. Therefore, the parameters first1 and last specify the source, and the parameter first2 specifies the destination. Note that the elements within the range first1...last-1 are copied.

The definition of the function template copy is contained in the header file algorithm. Thus, to use the function copy, the program must include the statement

```
#include <algorithm>
```

The function copy works as follows. Consider the following statement:

```
int intArray[] = {5, 6, 8, 3, 40, 36, 98, 29, 75};   //Line 1 vector<int>
vecList(9);     //Line 2
```

This statement in Line 1 creates the array intArray of nine components—that is,
intArray = {5, 6, 8, 3, 40, 36, 98, 29, 75}

Here intArray[0] = 5, intArray[1] = 6, and so on.

The statement in Line 2 creates an empty container of nine components of type vector and the element type int.

Recall that the array name, intArray, is actually a pointer and contains the base address of the array. Therefore, intArray points to the first component of the array, intArray + 1 points to the second

component of the array, and so on.

Now consider the statement

```
copy(intArray, intArray+9, vecList.begin());          //Line 3
```

This statement copies the elements starting at the location intArray, which is the first component of the array intArray, until intArray + 9 - 1 (that is, intArray + 8), which is the last element of the array intArray, into the container vecList. (Note that here first1 is intArray, last is intArray + 9, and first2 is vecList.begin().)

After the statement in Line 3 executes,

```
vecList = {5, 6, 8, 3, 40, 36, 98, 29, 75}          //Line 4
```

Next, consider the statement

```
copy(intArray + 1, intArray + 9, intArray);   //Line 5
```

Here first1 is intArray + 1; that is, first1 points to the location of the second element of the array intArray, and last is intArray + 9. Also, first2 is intArray; that is, first2 points to the location of the first element of the array intArray. Therefore, the second array element is copied into the first array component, the third array element into the second array component, and so on. After the statement in
Line 5 executes,

```
intArray[] = {6, 8, 3, 40, 36, 98, 29, 75, 75}                    //Line 6
```

Notice that the elements of the array intArray are shifted to the left by one position.

Suppose that vecList is as in Line 4. Consider the statement

```
copy(vecList.rbegin() + 2, vecList.rend(), vecList.rbegin());  //Line 7
```

Recall that the function rbegin (reverse begin) returns a pointer to the last element into a container; it is used to process the elements of a container in reverse. Therefore, vecList.rbegin() + 2 returns a pointer to the third-to-last element into the container vecList. Similarly, the function rend (reverse end) returns a pointer to the first element of a container. The previous statement shifts the elements of the container vecList to the right by two positions. After the statement in Line 7 executes, the container vecList is as follows:

```
vecList = {5, 6, 5, 6, 8, 3, 40, 36, 98}
```

Example 2-6 shows the effect of the preceding statements using a C++ program. Before discussing Example 2-6, let us describe a special type of iterator, called ostream iterators, which work well with the function copy to copy the elements of a container to an output device.

2.6 ostream Iterator and Function copy

One way to output the contents of a container is to use a for loop and the function begin to initialize the for loop control variable, and to use the function end to set the limit. Alternatively, the function copy can

be used to output the elements of a container. In this case, an iterator of type ostream specifies the destination (ostream iterators are discussed in detail later in this chapter). When we create an iterator of type ostream, we also specify the type of element the iterator will output.

The following statement illustrates how to create an ostream iterator of type int:

```
ostream_iterator<int> screen(cout, " ");          //Line A
```

This statement creates screen to be an ostream iterator with the element type int. The iterator screen has two arguments: the object cout and a space. Thus, the iterator screen is initialized using the object cout, and when this iterator outputs the elements they are separated by a space.

The statement

```
copy(intArray, intArray+9, screen);
```

outputs the elements of intArray on the screen.

Similarly, the statement

```
copy(vecList.begin(), vecList.end(), screen);
```

outputs the elements of the container vecList on the screen.

We will frequently use the function copy to output the elements of a container by using an ostream iterator. Also, until we discuss ostream iterators in detail, we will use statements similar to the statement in Line A to create an ostream iterator.

Of course, we can directly specify an ostream iterator in the function copy. For example, the statement (shown previously)

```
copy(vecList.begin(), vecList.end(), screen);
```

is equivalent to the statement

```
copy(vecList.begin(), vecList.end(), ostream_iterator<int>(cout, " "));
```

Finally, the statement

```
copy(vecList.begin(), vecList.end(), ostream_iterator<int>(cout, ", "));
```

outputs the elements of vecList with a comma and space between them.

Example 2-6 illustrates how to use the function copy and an ostream iterator in a program.

EXAMPLE 2-6

```
//*************************************************************
* // This program illustrates how to use the function copy and // an
ostream iterator in a program.
//*************************************************************
#include <algorithm>                                    //Line 1
#include <vector>                                       //Line 2
#include <iterator>                                     //Line 3
```

```
#include <iostream>                                          //Line 4

using namespace std;                                        //Line 5

int main()                                                  //Line 6

{                                                           //Line 7

        int intArray[] = {5, 6, 8, 3, 40, 36, 98, 29, 75};  //Line 8

        vector<int> vecList(9);                             //Line 9

        ostream_iterator<int> screen(cout, " ");            //Line 10

        cout << "Line 11: intArray: ";                      //Line 11
```

```
        copy(intArray, intArray + 9, screen);                          //Line 12

        cout << endl;                                                  //Line 13

        copy(intArray, intArray + 9, vecList.begin());                 //Line 14

        cout << "Line 15: vecList: ";                                  //Line 15

        copy(vecList.begin(), vecList.end(), screen);                  //Line 16

        cout << endl;                                                  //Line 17

        copy(intArray + 1, intArray + 9, intArray);                    //Line 18
cout << "Line 19: After shifting the elements one "
             << "position to the left, intArray: " << endl;            //Line 19
        copy(intArray, intArray + 9, screen);                          //Line 20
        cout << endl;                                                  //Line 21

        copy(vecList.rbegin() + 2, vecList.rend(), vecList.rbegin());  //Line 22 cout
        << "Line 23: After shifting the elements down "
             << "by two positions, vecList:" << endl;                  //Line 23
        copy(vecList.begin(), vecList.end(), screen);                  //Line 24
        cout << endl;                                                  //Line 25

        return 0;                                                      //Line 26

    }                                                                  //Line 27
```

Sample Run:

Line 11: intArray: 5 6 8 3 40 36 98 29 75

Line 15: vecList: 5 6 8 3 40 36 98 29 75

Line 19: After shifting the elements one position to the left, intArray:

6 8 3 40 36 98 29 75 75

Line 23: After shifting the elements down by two positions, vecList: 5

6 5 6 8 3 40 36 98

2.6.1 Sequence Container: deque

This section describes the deque sequence containers. The term deque stands for double-ended queue. Deque containers are implemented as dynamic arrays in such a way that the elements can be inserted at both ends. Thus, a deque can expand in either direction. Elements can also be inserted in the middle. Inserting elements at the beginning or at the end is fast; inserting elements in the middle, however, is time consuming because the elements in the queue need to be shifted.

The name of the class defining the deque containers is deque. The definition of the class deque, and the functions to implement the various operations on a deque object, are also contained in the header file deque. Therefore, to use a deque container in a program, the program must include the following statement:

```
#include <deque>
```

The class deque contains several constructors. Thus, a deque object can be initialized in various ways when it is declared, as described in Table 2-7.

TABLE 2-7: Various ways to declare a deque object

Statement	Effect
`deque<elementType> deq;`	Creates an empty **deque** container without any elements. (The default constructor is invoked.)
`deque<elementType> deq(otherDeq);`	Creates a deque container, **deq**, and initializes **deq** to the elements of **otherDeq**; **deq** and **otherDeq** are of the same type.
`deque<elementType> deq(size);`	Creates a deque container, **deq**, of size **size**. **deq** is initialized using the default constructor.
`deque<elementType> deq(n, elem);`	Creates a deque container, **deq**, of size **n**. **deq** is initialized using **n** copies of the element **elem**.
`deque<elementType> deq(begin, end);`	Creates a deque container, **deq**. **deq** is initialized to the elements in the range **[begin, end)**—that is, all elements in the range **begin...end-1**.

In addition to the operations that are common to all containers (see Table 2-6), Table 2-8 describes the operations that can be used to manipulate the elements of a deque container. The name of the function implementing the operations is shown in bold. The statement also shows how to use a particular function. Suppose that deq is a dequecontainer.

TABLE 2-8: Various operations that can be performed on a deque object

Expression	Effect
`deq.assign(n,elem)`	Assigns `n` copies of `elem`.
`deq.assign(beg,end)`	Assigns all the elements in the range `beg...end-1`.
`deq.push_front(elem)`	Inserts `elem` at the beginning of `deq`.
`deq.pop_front()`	Removes the first element from `deq`.
`deq.at(index)`	Returns the element at the position specified by `index`.
`deq[index]`	Returns the element at the position specified by `index`.
`deq.front()`	Returns the first element. (Does not check whether the container is empty.)
`deq.back()`	Returns the last element. (Does not check whether the container is empty.)

Example 2-7 illustrates how to use a deque container in a program.

EXAMPLE 2-7

```
//*********************************************************** //
This program illustrates how to use a deque container in a program.
//***********************************************************
#include <iostream>                                                    //Line 1
#include <deque>                                                       //Line 2
#include <algorithm>                                                   //Line 3
#include <iterator>                                                    //Line 4
using namespace std;                                                   //Line 5
int main()                                                             //Line 6
{                                                                      //Line 7
        deque<int> intDeq;                                             //Line 8
        ostream_iterator<int> screen(cout, " ");                       //Line 9
        intDeq.push_back(13);                                          //Line 10
        intDeq.push_back(75);                                          //Line 11
        intDeq.push_back(28);                                          //Line 12
        intDeq.push_back(35);                                          //Line 13
```

```
cout << "Line 14: intDeq: ";                                    //Line 14
copy(intDeq.begin(), intDeq.end(), screen);                     //Line 15
cout << endl;                                                   //Line 16
intDeq.push_front(0);                                           //Line 17
intDeq.push_back(100);                                          //Line 18
cout << "Line 19: After adding two more "<< "elements, one at the front " << endl
        << " and one at the back, intDeq: ";                    //Line 19
copy(intDeq.begin(), intDeq.end(), screen);                     //Line 20
```

```
    cout << endl;                                                          //Line 21

    intDeq.pop_front();                                                    //Line 22

    intDeq. pop_front();                                                   //Line 23

    cout << "Line 24: After removing the first " << "two elements, intDeq: ";  //Line 24
    copy(intDeq.begin(), intDeq.end(), screen);                            //Line 25
    cout << endl;                                                          //Line 26

    intDeq.pop_back();                                                     //Line 27

    intDeq.pop_back();                                                     //Line 28

    cout << "Line 29: After removing the last " << "two elements, intDeq = ";  //Line 29
    copy(intDeq.begin(), intDeq.end(), screen);                            //Line 30
    cout << endl;                                                          //Line 31

    deque<int>::iterator deqIt;                                            //Line 32

    deqIt = intDeq.begin();                                                //Line 33

    ++deqIt;                          //deqIt points to the second element  //Line 34
    intDeq.insert(deqIt, 444);                                             //Line 35
    cout << "Line 36: After inserting 444, intDeq: ";                      //Line 36

    copy(intDeq.begin(), intDeq.end(), screen);                            //Line 37

    cout << endl;                                                          //Line 38

    return 0;                                                              //Line 39

}                                                                          //Line 40
```

Sample Run:

Line 14: intDeq: 13 75 28 35

Line 19: After adding two more elements, one at the front

and one at the back, intDeq: 0 13 75 28 35 100

Line 24: After removing the first two elements, intDeq: 75 28 35 100

Line 29: After removing the last two elements, intDeq = 75 28 Line

36: After inserting 444, intDeq: 75 444 28

The statement in Line 8 declares a deque container intDeq of type int; that is, all the elements of intDeq are of type int. The statement in Line 9 declares screen to be an ostream iterator initialized to the standard output device. The statements in Lines 10 through 13 use the push_back operation to insert four numbers—13, 75, 28, and 35—into intDeq. The statement in Line 15 outputs the elements of intDeq. In the output, see the line marked Line 14, which contains the output of the statements in Lines 14 through 16 of the program.

The statement in Line 17 inserts 0 at the beginning of intDeq. The statement in Line 18 inserts 100 at the end of intDeq. The statement in Line 20 outputs the modified intDeq.

The statements in Lines 22 and 23 use the operation pop_front to remove the first two elements of intDeq; the statement in Line 25 outputs the modified intDeq. The statements in Lines 27 and 28 use the operation pop_back to remove the last two elements of intDeq; the statement in Line 30 outputs the modified intDeq.

The statement in Line 32 declares deqIt to be a deque iterator that processes all deque containers whose elements are of type int. After the statement in Line 33 executes, deqIt points to the first element of intDeq. The statement in Line 34 advances deqIt to the next element of intDeq. The statement in Line 35 inserts 444 into intDeq at the position specified by deqIt. The statement in Line 37 outputs intDeq.

2.7 Iterators

Examples 2-5 through 2-7 further clarify that iterators are quite important to efficiently process the elements of a container. Let us discuss iterators in some detail.

Iterators work just like pointers. In general, an iterator points to the elements of a container (sequence or associative). Thus, with the help of iterators, we can successively access each element of a container.

The two most common operations on iterators are ++ (the increment operator) and * (the dereferencing operator). Suppose that cntItr is an iterator into a container. The statement

```
++cntItr;
```

advances cntItr so that it points to the next element in the container. The statement

```
*cntItr;
```

returns the value of the element of the container pointed to by cntItr.

2.1.1 Types of Iterators

There are five types of iterators: input iterators, output iterators, forward iterators, bidirectional iterators, and random access iterators. In the next few sections, we describe these iterators.

2.1.1.1 Input Iterators

Input iterators, with read access, step forward element-by-element and so return the values elementby-element. These iterators are provided for reading data from an input stream. Suppose inputIterator is an input iterator. Table 2-9 describes the operations on inputIterator.

TABLE 2-9: Operations on an input iterator

Expression	Effect
`*inputIterator`	Gives access to the element to which `inputIterator` points.
`inputIterator->member`	Gives access to the member of the element.
`++inputIterator`	Moves forward, returns the new position (preincrement).
`inputIterator++`	Moves forward, returns the old position (postincrement).
`inputIt1 == inputIt2`	Returns `true` if the two iterators are the same and `false` otherwise.
`inputIt1 != inputIt2`	Returns `true` if the two iterators are not the same and `false` otherwise.
`Type(inputIterator)`	Copies the iterators.

2.1.1.2 Output Iterators

Output iterators, with write access, step forward element-by-element. These iterators are typically used for writing data to an output stream. Suppose outputIterator is an output iterator. Table 2-10 describes the operations on outputIterator.

TABLE 2-10: Operations on an output iterator

Expression	Effect
`*outputIterator = value;`	Writes the `value` at the position specified by the `outputIterator`.
`++outputIterator`	Moves forward, returns the new position (preincrement).
`outputIterator++`	Moves forward, returns the old position (postincrement).
`Type(outputIterator)`	Copies the iterators.

2.1.1.3 Forward Iterators

Forward iterators combine all of the functionality of input iterators and almost all of the functionality of output iterators. Suppose forwardIterator is a forward iterator. Table 2-11 describes the operations on forwardIterator.

TABLE 2-11: Operations on a forward iterator

Expression	Effect
`*forwardIterator`	Gives access to the element to which `forwardIterator` points.
`forwardIterator->member`	Gives access to the member of the element.
`++forwardIterator`	Moves forward, returns the new position (preincrement).
`forwardIterator++`	Moves forward, returns the old position (postincrement).
`forwardIt1 == forwardIt2`	Returns `true` if the two iterators are the same and `false` otherwise.
`forwardIt1 != forwardIt2`	Returns `true` if the two iterators are not the same and `false` otherwise.
`forwardIt1 = forwardIt2`	Assignment.

2.1.1.4 Bidirectional Iterators

Bidirectional iterators are forward iterators that can also iterate backward over the elements. Suppose biDirectionalIterator is a bidirectional iterator. The operations defined for forward iterators (Table 2-11) are also applicable to bidirectional iterators. To step backward, the decrement operations are also defined for biDirectionalIterator. Table 2-12 shows additional operations on a bidirectional iterator.

TABLE 2-12: Additional operations on a bidirectional iterator

Expression	Effect
`--biDirectionalIterator`	Moves backward, returns the new position (predecrement).
`biDirectionalIterator--`	Moves backward, returns the old position (postdecrement).

2.1.1.5 Random Access Iterators

Random access iterators are bidirectional iterators that can randomly process the elements of a container. These iterators can be used with containers of type vector, deque, string, and arrays. The operations defined for bidirectional iterators (for example, Tables 2-11 and 2-12) are also applicable to random access

iterators. Table 2-13 describes the additional operations that are defined for random access iterators. (Suppose rAccessIterator is a random access iterator.)

TABLE 2-13: Additional operations on a random access iterator

Expression	Effect
`rAccessIterator[n]`	Accesses the nth element.
`rAccessIterator += n`	Moves `rAccessIterator` forward n elements if `n >= 0` and backward if `n < 0`.
`rAccessIterator -= n`	Moves `rAccessIterator` backward n elements if `n >= 0` and forward if `n < 0`.
`rAccessIterator + n`	Returns the iterator of the next nth element.
`n + rAccessIterator`	Returns the iterator of the next nth element.
`rAccessIterator - n`	Returns the iterator of the previous nth element.
`rAccessIt1 - rAccessIt2`	Returns the distance between the iterators `rAccessIt1` and `rAccessIt2`.
`rAccessIt1 < rAccessIt2`	Returns `true` if `rAccessIt1` is before `rAccessIt2` and `false` otherwise.
`rAccessIt1 <= rAccessIt2`	Returns `true` if `rAccessIt1` is before or equal to `rAccessIt2` and `false` otherwise.
`rAccessIt1 > rAccessIt2`	Returns `true` if `rAccessIt1` is after `rAccessIt2` and `false` otherwise.
`rAccessIt1 >= rAccessIt2`	Returns `true` if `rAccessIt1` is after or equal to `rAccessIt2` and `false` otherwise.

Figure 2-1 shows the iterator hierarchy.

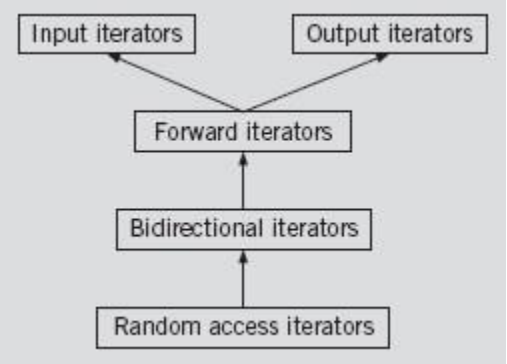


FIGURE 2-1 Iterator hierarchy

Now that you know the different types of iterators, next we describe how to declare an iterator to a container.

typedef iterator Every container (sequence or associative) contains a typedef iterator. Thus, an iterator into a container is declared using the typedef iterator. For example, the statement vector<int>::iterator intVecIter;

declares intVecIter to be an iterator into a vector container of type int. Moreover, the iterator intVecIter can be used on any vector<int>, but not on any other container, such as vector<double>, vector<string>, and deque.

Because iterator is a typedef defined inside a container (that is, a class) such as vector, we must use the appropriate container name, container element type, and the scope resolution operator to use the typedef iterator.

typedef const_iterator Because an iterator works like a pointer, with the help of an iterator into a container and the dereferencing operator, *, we can modify the elements of the container. However, if a container is declared as const, then we must prevent the iterator from modifying the elements of the container, especially accidentally. To handle this situation, every container contains another typedef const_iterator.
For example, the statement

vector<int>::const_iterator intConstVecIt;

declares intConstVecIt to be an iterator into a vector container whose elements are of type int. The iterator intConstVecIt is used to process the elements of those vector containers that are declared as constant vector containers of type vector<int>.

An iterator of type const_iterator is a read-only iterator.

typedef reverse_iterator Every container also contains the typedef reverse_iterator. An iterator of this type is used to iterate through the elements of a container in reverse.

typedef const_reverse_iterator An iterator of this type is a read-only iterator and is used to iterate through the elements of a container in reverse. It is required if the container is declared as const and we need to iterate through the elements of the container in reverse.

In addition to the previous four typedefs, several other typedefs are common to all containers. Table 2-14 describes them.

TABLE 2-14: Various typedefs common to all containers

typedef	Effect
difference_type	The type of result from subtracting two iterators referring to the same container.
pointer	A pointer to the type of elements stored in the container.
reference	A reference to the type of elements stored in the container.
const_reference	A constant reference to the type of elements stored in the container. A constant reference is read-only.
size_type	The type used to count the elements in a container. This type is also used to index through sequence containers, except `list` containers.
value_type	The type of container elements.

2.1.1.6 Stream Iterators

Another useful set of iterators is stream iterators—istream iterators and ostream iterators. This section describes both types of iterators.

istream_iterator The istream iterator is used to input data into a program from an input stream. The class istream_iterator contains the definition of an input stream iterator. The general syntax to use an istream iterator is as follows: istream_iterator<Type> isIdentifier(istream&);

where Type is either a built-in type or a user-defined class type, for which an input iterator is defined. The identifier isIdentifier is initialized using the constructor whose argument is either an istream class object such as cin, or any publicly defined istream subtype, such as ifstream.

ostream_iterator The ostream iterators are used to output data from a program into an output stream. These iterators were defined earlier in this chapter. We review them here for the sake of completeness.

The class ostream_iterator contains the definition of an output stream iterator. The general syntax to use an ostream iterator is as follows:

ostream_iterator<Type> osIdentifier(ostream&);

or ostream_iterator<Type> osIdentifier(ostream&, char* deLimit);

where Type is either a built-in type or a user-defined class type, for which an output iterator is defined. The identifier osIdentifier is initialized using the constructor whose argument is either an ostream class object such as cout, or any publicly defined ostream subtype, such as ofstream. In the second form used for declaring an ostream iterator, by using the second argument (deLimit) of the initializing constructor, we can specify a character separating the output.

Key Terms

1. The STL provides class templates that process lists, stacks, and queues.
2. The three main components of the STL are containers, iterators, and algorithms.
3. STL containers are class templates.
4. Iterators are used to step through the elements of a container.
5. Algorithms are used to manipulate the elements in a container.
6. The main categories of containers are sequence containers, associative containers, and container adapters.
7. The three predefined sequence containers are vector, deque, and list.
8. A vector container stores and manages its objects in a dynamic array.
9. Because an array is a random access data structure, elements of a vector can be accessed randomly.
10. The name of the class that implements the vector container is vector.
11. Item insertion in a vector container is accomplished by using the operations insert and push_back.
12. Item deletion in a vector container is accomplished by using the operations pop_back, erase, and clear.
13. An iterator to a vector container is declared using the typedef iterator, which is declared as a public member of the class vector.
14. Member functions common to all containers are the default constructor, constructors with parameters, the copy constructor, the destructor, empty, size, max_size, swap, begin, end, rbegin, rend, insert, erase, clear, and the relational operator functions.
15. The member function begin returns an iterator to the first element into the container.
16. The member function end returns an iterator to the last element into the container.
17. In addition to the member functions listed in 14, the other member functions common to all sequence containers are insert, push_back, pop_back, erase, clear, and resize. 18. The copy algorithm is used to copy the elements in a given range to another place.
19. The function copy, using an ostream iterator, can also be used to output the elements of a container.
20. When we create an iterator of the type ostream, we also specify the type of element that the iterator will output.
21. Deque containers are implemented as dynamic arrays in such a way that the elements can be inserted at both ends of the array.
22. A deque can expand in either direction.
23. The name of the header file containing the definition of the class deque is deque.
24. In addition to the operations that are common to all containers, the other operations that can be used to manipulate the elements of a deque are assign, push_front, pop_front, at, array subscripting

operator [], front, and back.

25. The five categories of iterators are: input, output, forward, bidirectional, and random access iterator.
26. Input iterators are used to input data from an input stream.
27. Output iterators are used to output data to an output stream.
28. A forward iterator can refer to the same element in the same collection and process the same element more than once.
29. Bidirectional iterators are forward iterators that can also iterate backwards over the elements.
30. Bidirectional iterators can be used with containers of type list, set, multiset, map, and multimap.
31. Random access iterators are bidirectional iterators that can randomly process the elements of a container.
32. Random access iterators can be used with containers of type vector, deque, string, and arrays.

Review Questions

1. What are the three main components of the STL?

2. What is the difference between an STL container and an STL iterator?

3. Write a statement that declares a vector object that can store 50 decimal numbers.

4. Write a statement that declares and stores the elements of the following array into a vector object:
 char vowels[5] = {'a', 'e', 'i', 'o', 'u'};

5. Write a statement to declare screen to be an ostream_iterator initialized to the standard output device that outputs the elements of an int vector object.

6. Consider the following statements:
 vector<int> intVector;

Suppose that intVector = {5, 7, 9, 11, 13}. Moreover, suppose that screen is an ostream_iterator initialized to the standard output device to output the elements of an int vector object. What is the effect of the following statement?

```
copy(vecList.begin(), vecList.end(), screen);
```

7. What is the output of the following program segment?

```
vector<int> vecList(5);

for (int j = 0; j < 5; j++) vecList[j]
        = 2 * j;
for (int j = 0; j < 5; j++) cout <<
        vecList[j] << " ";
```

```
cout << endl;
```

8. What is the output of the following program segment? (Assume that screen is an ostream_iterator initialized to the standard output device to output elements of type int.)

```
int list[5] = {2,4,6,8,10};
vector<int> vecList(5);
copy(list, list + 5, vecList.begin());
copy(vecList.begin(), vecList.end(), screen); cout
<< endl;
```

9. What is the output of the following program segment? (Assume that screen is an ostream_iterator initialized to the standard output device to output elements of type int.)

```
vector<int> vecList;
vector<int>::iterator vecIt;

vecList.push_back(3);
vecList.push_back(5);
vecList.push_back(7);
vecIt = vecList.begin();
++vecIt; vecList.erase(vecIt);
vecList.push_back(9);

copy(vecList.begin(), vecList.end(), screen); cout
<< endl;
```

10. What is the output of the following program segment? (Assume that screen is an ostream_iterator initialized to the standard output device to output elements of type int.)

```
int list[5] = {2,4,6,8,10};
vector<int> vecList(7);

copy(list, list + 5, vecList.begin());
vecList.push_back(12);
copy(vecList.begin(), vecList.end(), screen); cout
<< endl;
```

11. What is the output of the following program segment? (Assume that screen is an ostream_iterator initialized to the standard output device to output elements of type double.)

```
vector<double> sales(3);
sales[0] = 50.00; sales[1]
= 75.00;
sales[2] = 100.00;
```

```
sales.resize(5);
sales[3] = 200.00;
sales[4] = 95.00;

copy(sales.begin(), sales.end(), screen); cout
<< endl;
```

12. What is the output of the following program segment? (Assume that screen is an ostream_iterator initialized to the standard output device that outputs elements of type int.)

```
vector<int> intVector; vector<int>::iterator
vecIt; intVector.push_back(15);
intVector.push_back(2);
intVector.push_back(10);
intVector.push_back(7); vecIt =
intVector.begin(); vecIt++;
intVector.erase(vecIt);
intVector.pop_back();

copy(intVector.begin(),intVector.end(),  screen);
```

13. Suppose that vecList is a vector container and
vecList = {12, 16, 8, 23, 40, 6, 18, 9, 75}

Show vecList after the following statement executes:
copy(vecList.begin() + 2, vecList.end(), vecList.begin());

14. Suppose that vecList is a vector container and
vecList = {12, 16, 8, 23, 40, 6, 18, 9, 75}

Show vecList after the following statement executes:
copy(vecList.rbegin() + 3, vecList.rend(), vecList.rbegin());

15. What is the output of the following program segment?

```
deque<int> intDeq;
ostream_iterator<int> screen(cout, " "); deque<int>::iterator
deqIt;

intDeq.push_back(5);
```

```
intDeq.push_front(23);
intDeq.push_front(45);
intDeq.push_back(35);
intDeq.push_front(0);
intDeq.push_back(50);
intDeq.push_front(34);

deqIt = intDeq.begin();
intDeq.insert(deqIt,76);
intDeq.pop_back();
deqIt = intDeq.begin();
++deqIt;
++deqIt;

intDeq.erase(deqIt);
```

```
intDeq.push_front(2 * intDeq.back()); intDeq.push_back(3
* intDeq.front());

copy(intDeq.begin(), intDeq.end(), screen); cout
<< endl;
```

3 TOPIC 3: LINKED LISTS



LEARNING OUTCOMES:

After studying this topic you should be able to:

- Learn about linked lists
- Become aware of the basic properties of linked lists
- Explore the insertion and deletion operations on linked lists
- Discover how to build and manipulate a linked list
- Learn how to construct a doubly linked list
- Discover how to use the STL container list
- Learn about linked lists with header and trailer nodes
- Become aware of circular linked lists

You have already seen how data is organized and processed sequentially using an array, called a sequential list. You have performed several operations on sequential lists, such as sorting, inserting, deleting, and searching. You also found that if data is not sorted, searching for an item in the list can be very time consuming, especially with large lists. Once the data is sorted, you can use a binary search and improve the search algorithm. However, in this case, insertion and deletion become time consuming, especially with large lists because these operations require data movement. Also, because the array size must be fixed during execution, new items can be added only if there is room. Thus, there are limitations when you organize data in an array.

This chapter helps you to overcome some of these problems. This chapter uses pointers to organize and process data in lists, called linked lists. Recall that when data is stored in an array, memory for the components of the array is contiguous—that is, the blocks are allocated one after the other.However, as we will see, the components (called nodes) of a linked list need not be contiguous.

3.1 Linked Lists

A linked list is a collection of components, called nodes. Every node (except the last node) contains the address of the next node. Thus, every node in a linked list has two components: one to store the relevant

information (that is, data) and one to store the address, called the link, of the next node in the list. The address of the first node in the list is stored in a separate location, called the head or first. Figure 3-1 is a pictorial representation of a node.

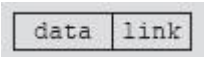


FIGURE 3-1 Structure of a node

Linked list: A list of items, called nodes, in which the order of the nodes is determined by the address, called the link, stored in each node.

The list in Figure 3-2 is an example of a linked list.

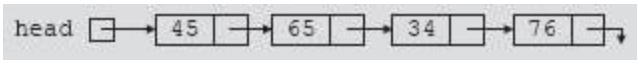


FIGURE 3-2 Linked list

The arrow in each node indicates that the address of the node to which it is pointing is stored in that node. The down arrow in the last node indicates that this link field is NULL.

For a better understanding of this notation, suppose that the first node is at memory location 1200, and the second node is at memory location 1575, see Figure 3-3.

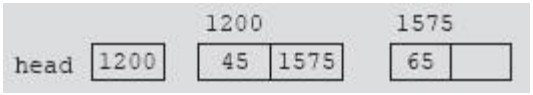


FIGURE 3-3 Linked list and values of the links

The value of the head is 1200, the data part of the first node is 45, and the link component of the first node contains 1575, the address of the second node. If no confusion arises, we will use the arrow notation whenever we draw the figure of a linked list.

For simplicity and for the ease of understanding and clarity, Figures 3-3 through 3-5 use decimal integers as the values of memory addresses. However, in computer memory the memory addresses are in binary.

Because each node of a linked list has two components, we need to declare each node as a class or struct. The data type of each node depends on the specific application—that is, what kind of data is being processed. However, the link component of each node is a pointer. The data type of this pointer variable is the node type itself. For the previous linked list, the definition of the node is as follows. (Suppose that the data type is int.)

```
struct nodeType
{
        int info; nodeType
        *link;
};
```

The variable declaration is as follows:

```
nodeType *head;
```

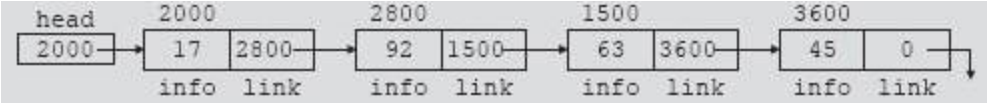


FIGURE 3-4 Linked list with four nodes

3.1.1 Linked Lists: Some Properties

To better understand the concept of a linked list and a node, some important properties of linked lists are described next.

Consider the linked list in Figure 3-4.

This linked list has four nodes. The address of the first node is stored in the pointer head. Each node has two components: info, to store the info, and link, to store the address of the next node. For simplicity, we assume that info is of type int.

Suppose that the first node is at location 2000, the second node is at location 2800, the third node is at location 1500, and the fourth node is at location 3600. Table 3-1 shows the values of head and some other nodes in the list shown in Figure 3-4.

TABLE 3-1: Values of head and some of the nodes of the linked list in Figure 3-4

	Value	Explanation
`head`	2000	
`head->info`	17	Because `head` is 2000 and the `info` of the node at location 2000 is 17
`head->link`	2800	
`head->link->info`	92	Because `head->link` is 2800 and the `info` of the node at location 2800 is 92

Suppose that current is a pointer of the same type as the pointer head. Then the statement

```
current = head;
```

copies the value of head into current. Now consider the following statement:

```
current = current->link;
```

This statement copies the value of current->link, which is 2800, into current. Therefore, after this statement executes, current points to the second node in the list. (When working with linked lists, we typically use these types of statements to advance a pointer to the next node in the list.) See Figure 3-5.

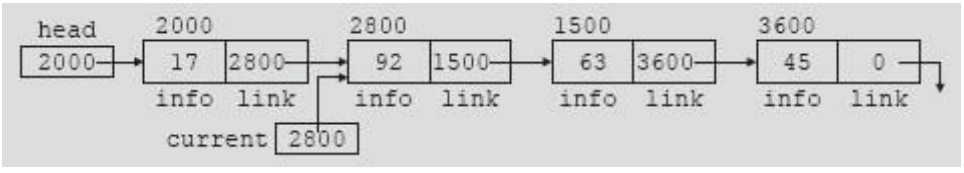


FIGURE 3-5 List after the statement current = current->link; executes

Table 3-2 shows the values of current, head, and some other nodes in Figure 3-5.

TABLE 3-2: Values of current, head, and some of the nodes of the linked list in Figure 3-5

	Value
current	2800
current->info	92
current->link	1500
current->link->info	63
head->link->link	1500
head->link->link->info	63
head->link->link->link	3600
current->link->link->link	0 (that is, NULL)
current->link->link->link->info	Does not exist (run-time error)

From now on, when working with linked lists, we will use only the arrow notation.

3.1.2 Traversing a Linked List

The basic operations of a linked list are as follows: Search the list to determine whether a particular item is in the list, insert an item in the list, and delete an item from the list. These operations require the list to be traversed. That is, given a pointer to the first node of the list, we must step through the nodes of the list.

Suppose that the pointer head points to the first node in the list, and the link of the last node is NULL. We cannot use the pointer head to traverse the list because if we use the head to traverse the list, we would lose the nodes of the list. This problem occurs because the links are in only one direction. The pointer head contains the address of the first node, the first node contains the address of the second node, the second node contains the address of the third node, and so on. If we move head to the second node, the first node is lost (unless we save a pointer to this node). If we keep advancing head to the next node, we will lose all the nodes of the list (unless we save a pointer to each node before advancing head, which is impractical because it would require additional computer time and memory space to maintain the list).

Therefore, we always want head to point to the first node. It now follows that we must traverse the list using another pointer of the same type. Suppose that current is a pointer of the same type as head. The

following code traverses the list:

```
current = head;

while (current != NULL)
{
        //Process current current
        = current->link; }
```

For example, suppose that head points to a linked list of numbers. The following code outputs the data stored in each node:

```
current = head;

while (current != NULL)
{ cout << current->info << " ";
        current = current->link;
}
```

3.2 Item Insertion and Deletion

This section discusses how to insert an item into, and delete an item from, a linked list. Consider the following definition of a node. (For simplicity, we assume that the info type is int. The next section, which discusses linked lists as an abstract data type (ADT) using templates, uses the generic definition of a node.)

```
struct nodeType
{
        int info; nodeType
        *link;
};
```

We will use the following variable declaration:

```
nodeType *head, *p, *q, *newNode;
```

3.2.1 Insertion

Consider the linked list shown in Figure 3-6.



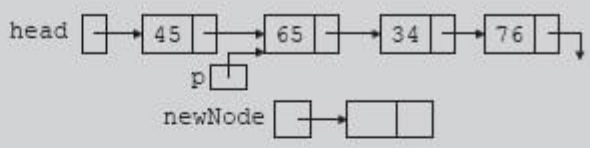
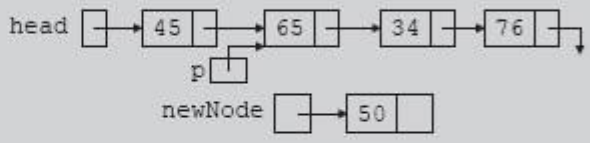
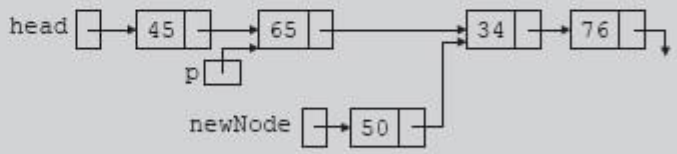
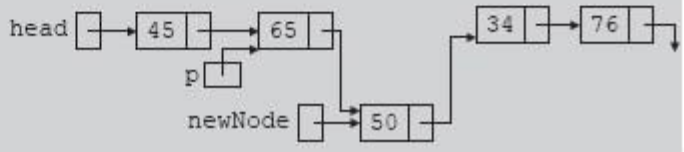
FIGURE 3-6 Linked list before item insertion

Suppose that p points to the node with info 65, and a new node with info 50 is to be created and inserted after p. Consider the following statements:

```
newNode = new nodeType;          //create newNode
newNode->info = 50;              //store 50 in the new node
newNode->link = p->link; p->link
= newNode;
```

Table 3-3 shows the effect of these statements.

TABLE 3-3: Inserting a node in a linked list

Statement	Effect
`newNode = new nodeType;`	
`newNode->info = 50;`	
`newNode->link = p->link;`	
`p->link = newNode;`	

Note that the sequence of statements to insert the node, that is,

```
newNode->link = p->link; p->link
= newNode;
```

is very important because to insert newNode in the list we use only one pointer, p, to adjust the links of the nodes of the linked list. Suppose that we reverse the sequence of the statements and execute the statements in the following order:

```
p->link = newNode;
newNode->link = p->link;
```

Figure 3-7 shows the resulting list after these statements execute.

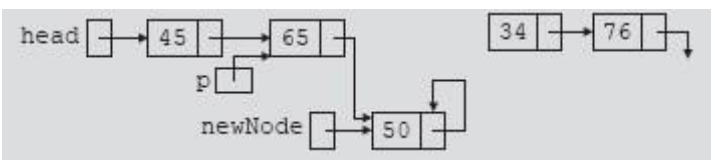


FIGURE 3-7 List after the execution of the statement p->link = newNode; followed by the execution of the statement newNode->link = p->link;

From Figure 3-7, it is clear that newNode points back to itself and the remainder of the list is lost.

Using two pointers, we can simplify the insertion code somewhat. Suppose q points to the node with info 34. (See Figure 3-8.)

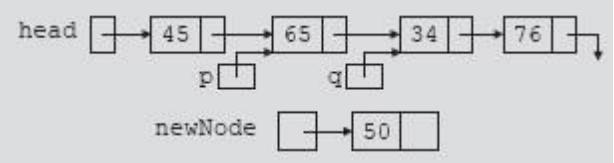


FIGURE 3-8 List with pointers p and q

The following statements insert newNode between p and q:

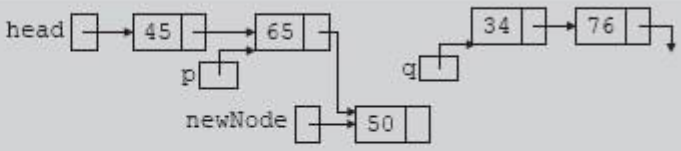
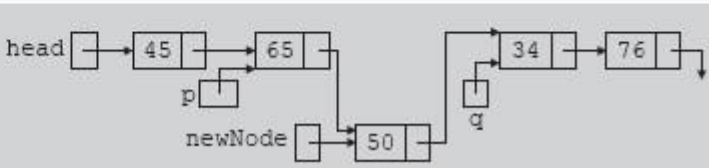
```
newNode->link = q; p->link
= newNode;
```

The order in which these statements execute does not matter. To illustrate this, suppose that we execute the statements in the following order:

```
p->link = newNode; newNode->link
= q;
```

Table 3-4 shows the effect of these statements.

TABLE 3-4: Inserting a node in a linked list using two pointers

Statement	Effect
`p->link = newNode;`	
`newNode->link = q;`	

3.2.2 Deletion

Consider the linked list shown in Figure 3-9.



FIGURE 3-9 Node to be deleted is with info 34

Suppose that the node with info 34 is to be deleted from the list. The following statement removes the node from the list:

```
p->link = p->link->link;
```

Figure 3-10 shows the resulting list after the preceding statement executes.

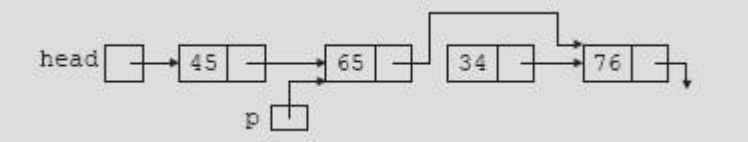


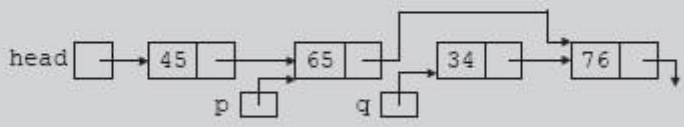
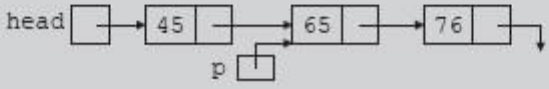
FIGURE 3-10 List after the statement p->link = p->link->link; executes

From Figure 3-10, it is clear that the node with info 34 is removed from the list. However, the memory is still occupied by this node and this memory is inaccessible; that is, this node is dangling. To deallocate the memory, we need a pointer to this node. The following statements delete the node from the list and deallocate the memory occupied by this node:

```
q = p->link; p->link
= q->link; delete
q;
```

Table 3-5 shows the effect of these statements.

TABLE 3-5: Deleting a node from a linked list

Statement	Effect
`q = p->link;`	
`p->link = q->link;`	
`delete q;`	

3.2.3 Building a Linked List

Now that we know how to insert a node in a linked list, let us see how to build a linked list. First, we consider a linked list in general. If the data we read is unsorted, the linked list will be unsorted. Such a list can be built in two ways: forward and backward. In the forward manner, a new node is always inserted at the end of the linked list. In the backward manner, a new node is always inserted at the beginning of the

list. We will consider both cases.

3.2.4 Building a Linked List Forward

Suppose that the nodes are in the usual info-link form and info is of type int. Let us assume that we process the following data:

2 15 8 24 34

We need three pointers to build the list: one to point to the first node in the list, which cannot be moved, one to point to the last node in the list, and one to create the new node. Consider the following variable declaration:

```
nodeType *first, *last, *newNode; int
num;
```

Suppose that first points to the first node in the list. Initially, the list is empty, so both first and last are NULL. Thus, we must have the statements

```
first = NULL;
last = NULL;
```

to initialize first and last to NULL.

Next, consider the following statements:

```
1       cin >> num;                          //read and store a number in num
2       newNode = new nodeType;              //allocate memory of type nodeType
                                             //and store the address of the
                                             //allocated memory in newNode 3
newNode->info = num;                         //copy the value of num into the
                                             //info field of newNode
4       newNode->link = NULL;                //initialize the link field of
                                             //newNode to NULL
5       if (first == NULL)                   //if first is NULL, the list is empty;
                                             //make first and last point to newNode
        {
5a              first = newNode;
5b              last = newNode;
        }
6       else                                 //list is not empty
```

```
        {
6a              last->link = newNode;                   //insert newNode at the end of the list 6b
        last = newNode;                         //set last so that it points to the
                                                        //actual last node in the list
        }
```

Let us now execute these statements. Initially, both first and last are NULL. Therefore, we have the list as shown in Figure 3-11.

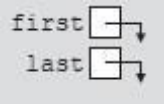


FIGURE 3-11 Empty list

After statement 1 executes, num is 2. Statement 2 creates a node and stores the address of that node in newNode. Statement 3 stores 2 in the info field of newNode, and statement 4 stores NULL in the link field of newNode. (See Figure 3-12.)

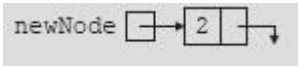


FIGURE 3-12 newNode with info 2

Because first is NULL, we execute statements 5a and 5b. Figure 3-13 shows the resulting list.

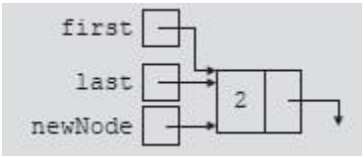


FIGURE 3-13 List after inserting newNode in it

We now repeat statements 1 through 6b. After statement 1 executes, num is 15. Statement 2 creates a node and stores the address of this node in newNode. Statement 3 stores 15 in the info field of newNode, and statement 4 stores NULL in the link field of newNode. (See Figure 3-14.)



FIGURE 3-14 List and newNode with info 15

Because first is not NULL, we execute statements 6a and 6b. Figure 3-15 shows the resulting list.

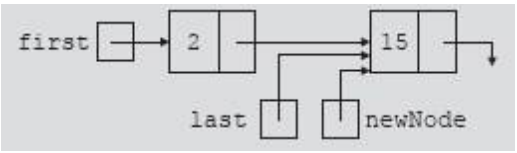


FIGURE 3-15 List after inserting newNode at the end

We now repeat statements 1 through 6b three more times. Figure 3-16 shows the resulting list.

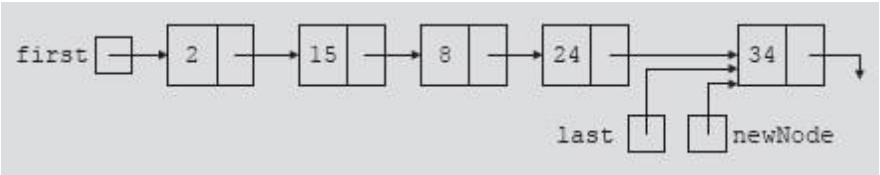


FIGURE 3-16 List after inserting 8, 24, and 34

We can put the previous statements in a loop, and execute the loop until certain conditions are met, to build the linked list. We can, in fact, write a C++ function to build a linked list.

Suppose that we read a list of integers ending with -999. The following function, buildListForward, builds a linked list (in a forward manner) and returns the pointer of the built list:

nodeType* buildListForward()

```
{
        nodeType *first, *newNode, *last; int
        num;

        cout << "Enter a list of integers ending with -999."<< endl; cin
        >> num;
        first = NULL;

        while (num != -999)
        {
                newNode = new nodeType; newNode->info
                = num;
                newNode->link = NULL;

                if (first == NULL)
                {
                        first = newNode;
                        last = newNode;
                }
                else
                {
                        last->link = newNode;
                        last = newNode;
                }

                cin >> num;
        }                                       //end while return
        first;
}                                               //end buildListForward
```

3.2.5 Building a Linked List Backward

Now we consider the case of building a linked list backward. For the previously given data—2, 15, 8, 24,

and 34—the linked list is as shown in Figure 3-17.

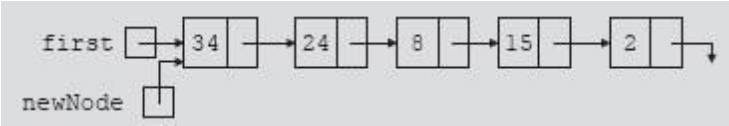


FIGURE 3-17 List after building it backward

Because the new node is always inserted at the beginning of the list, we do not need to know the end of the list, so the pointer last is not needed. Also, after inserting the new node at the beginning, the new node becomes the first node in the list. Thus, we need to update the value of the pointer first to correctly point to the first node in the list. We see, then, that we need only two pointers to build the linked list: one to point to the list and one to create the new node. Because initially the list is empty, the pointer first must be initialized to NULL. The following C++ function builds the linked list backward and returns the pointer of the built list:

```
nodeType* buildListBackward()
```

```
{
        nodeType *first, *newNode;
        int num;
        cout << "Enter a list of integers ending with -999."<< endl;
        cin >> num; first = NULL; while (num != -999)
        {
                newNode = new nodeType;                    //create a node newNode->info = num;
                                                   //store the data in newNode
                newNode->link = first;                              //put newNode at the beginning
                                                                    //of the list
                first = newNode;                                    //update the head pointer of
                                                                    //the list, that is, first
                cin >> num;                                         //read the next number
        }
return first;

}                                                                   //end buildListBackward
```

3.3 Linked List as an ADT

The previous sections taught you the basic properties of linked lists and how to construct and manipulate linked lists. Because a linked list is a very important data structure, rather than discuss specific lists such as a list of integers or a list of strings, this section discusses linked lists as an abstract data type (ADT). Using templates, this section gives a generic definition of linked lists, which is then used in the next section and later in this book. The programming example at the end of this chapter also uses this generic definition of linked lists.

The basic operations on linked lists are as follows:

1. Initialize the list.
2. Determine whether the list is empty.
3. Print the list.
4. Find the length of the list.
5. Destroy the list.

6. Retrieve the info contained in the first node.
7. Retrieve the info contained in the last node.
8. Search the list for a given item.
9. Insert an item in the list.
10. Delete an item from the list.
11. Make a copy of the linked list.

In general, there are two types of linked lists—sorted lists, whose elements are arranged according to some criteria, and unsorted lists, whose elements are in no particular order. The algorithms to implement the operations search, insert, and remove slightly differ for sorted and unsorted lists. Therefore, we will define the class linkedListType to implement the basic operations on a linked list as an abstract class.Using the principal of inheritance, we derive two classes— unorderedLinkedList and orderedLinkedList—from the class linkedListType.

Objects of the class unorderedLinkedList would arrange list elements in no particular order, that is, these lists may not be sorted. On the other hand, objects of the class orderedLinkedList would arrange elements

according to some comparison criteria, usually less than or equal to. That is, these lists will be in ascending order. Moreover, after inserting an element into or removing an element from an ordered list, the resulting list will be ordered.

If a linked list is unordered, we can insert a new item at either the end or the beginning. Furthermore, you can build such a list in either a forward manner or a backward manner. The function buildListForward inserts the new item at the end, whereas the function buildListBackward inserts the new item at the beginning. To accommodate both operations, we will write two functions: insertFirst to insert the new item at the beginning of the list and insertLast to insert the new item at the end of the list. Also, to make the algorithms more efficient, we will use two pointers in the list: first, which points to the first node in the list, and last, which points to the last node in the list.

3.3.1 Structure of Linked List Nodes

Recall that each node of a linked list must store the data as well as the address for the next node in the list (except the last node of the list). Therefore, the node has two instance variables. To simplify operations such as insert and delete, we define the class to implement the node of a linked list as a struct. The definition of the struct nodeType is as follows:

```
//Definition of the node template
<class Type>
struct nodeType
{
        Type info;
        nodeType<Type> *link;
};
```

3.3.2 Member Variables of the class linkedListType

To maintain a linked list, we use two pointers—first and last. The pointer first points to the first node in the list, and last points to the last node in the list. We also keep a count of the number of nodes in the list. Therefore, the class linkedListType has three instance variables, as follows:

protected:

```
int count;                    //variable to store the number of elements in the list
nodeType<Type> *first;        //pointer to the first node of the list nodeType<Type>
*last;   //pointer to the last node of the list
```

3.4 Linked List Iterators

One of the basic operations performed on a list is to process each node of the list. This requires the list to be traversed starting at the first node. Moreover, a specific application requires each node to be processed in a very specific way. A common technique to accomplish this is to provide an iterator. So what is an iterator? An iterator is an object that produces each element of a container, such as a linked list, one element at a time. The two most common operations on iterators are ++ (the increment operator) and * (the dereferencing operator). The increment operator advances the iterator to the next node in the list while the dereferencing operator returns the info of the current node.

Note that an iterator is an object. So we need to define a class, which we will call linkedListIterator, to create iterators to objects of the class linkedListType. The iterator class would have one member variable pointing to (the current) node.

```
//*************************************************************
//This class specifies the members to implement an iterator to a linked list.
//*************************************************************
template <class Type>
class linkedListIterator
{
        public:
```

```
};
        linkedListIterator(); //Default
        constructor
        //Postcondition: current = NULL;

        linkedListIterator(nodeType<Type> *ptr); //Constructor
        with a parameter.
        //Postcondition: current = ptr;

        Type operator*();
        //Function to overload the dereferencing operator *.
        //Postcondition: Returns the info contained in the node.

        linkedListIterator<Type> operator++();
        //Overload the preincrement operator.
        //Postcondition: The iterator is advanced to the next node.

        bool operator==(const linkedListIterator<Type>& right) const; //Overload
        the equality operator.
        //Postcondition: Returns true if this iterator is equal
        to // the iterator specified by right, otherwise it
        returns // false.

        bool operator!=(const linkedListIterator<Type>& right) const; //Overload
        the not equal to operator.
        //Postcondition: Returns true if this iterator is not equal to
        // the iterator specified by right, otherwise it returns //
        false.
         private:
        nodeType<Type> *current; //pointer to point to the current //node
        in the linked list
```

Figure 3-18 shows the UML class diagram of the class linkedListIterator.

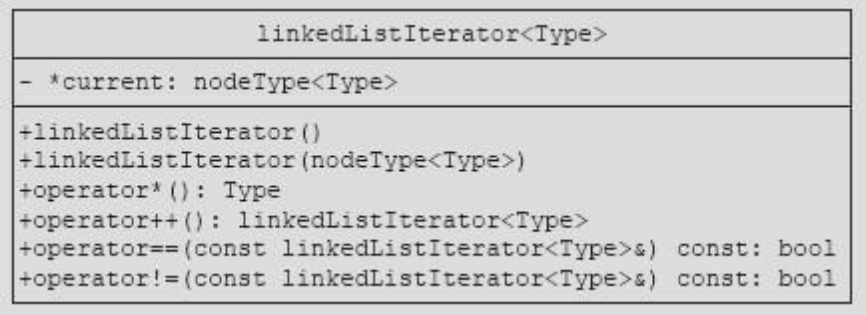


FIGURE 3-18 UML class diagram of the class linkedListIterator

The definitions of the functions of the class linkedListIterator are as follows:

```
template <class Type>
linkedListIterator<Type>::linkedListIterator()
{
        current = NULL;
}

template <class Type>
linkedListIterator<Type>::linkedListIterator(nodeType<Type> *ptr)
{ current = ptr;
}

template <class Type>
Type linkedListIterator<Type>::operator*()
{ return current->info;
}

template <class Type>
linkedListIterator<Type>  linkedListIterator<Type>::operator++()
{ current = current->link;
        return *this;
}
```

```
template <class Type>
bool linkedListIterator<Type>::operator==(const linkedListIterator<Type>& right) const
{ return (current == right.current);
}

template <class Type>
bool linkedListIterator<Type>::operator!=(const linkedListIterator<Type>& right) const
{ return (current != right.current);
}
```

From the definitions of the functions and constructors of the class linkedListIterator, it follows that each function and the constructors are of O(1).

Now that we have defined the classes to implement the node of a linked list and an iterator to a linked list, next, we describe the class linkedListType to implement the basic porperties of a linked list.

The following abstract class defines the basic properties of a linked list as an ADT:

```
//***********************************************************
// This class specifies the members to implement the basic //
properties of a linked list. This is an abstract class.
// We cannot instantiate an object of this class.
//***********************************************************  template
<class Type>
class linkedListType
{
        public:
                const linkedListType<Type>& operator=(const linkedListType<Type>&); //Overload
                the assignment operator.

                void initializeList();
                //Initialize the list to an empty state.
                //Postcondition: first = NULL, last = NULL, count = 0;
```

```
bool isEmptyList() const;
//Function to determine whether the list is empty.
//Postcondition: Returns true if the list is empty, otherwise //
it returns false.

void print() const;
//Function to output the data contained in each node.
//Postcondition: none

int length() const;
//Function to return the number of nodes in the list.
//Postcondition: The value of count is returned.

void destroyList();
//Function to delete all the nodes from the list.
//Postcondition: first = NULL, last = NULL, count = 0;

Type front() const;
//Function to return the first element of the list.
//Precondition: The list must exist and must not be empty.
//Postcondition: If the list is empty, the program terminates; //
otherwise, the first element of the list is returned.

Type back() const;
//Function to return the last element of the list.
//Precondition: The list must exist and must not be empty.
```

```
//Postcondition: If the list is empty, the program
// terminates; otherwise, the last // element of
the list is returned.

virtual bool search(const Type& searchItem) const = 0;
//Function to determine whether searchItem is in the list.
//Postcondition: Returns true if searchItem is in the list, //
otherwise the value false is returned.

virtual void insertFirst(const Type& newItem) = 0; //Function
to insert newItem at the beginning of the list.
//Postcondition: first points to the new list, newItem is
// inserted at the beginning of the list, last points to //
the last node in the list, and count is incremented by //
1.

virtual void insertLast(const Type& newItem) = 0; //Function
to insert newItem at the end of the list.
//Postcondition: first points to the new list, newItem
is // inserted at the end of the list, last points to the //
last node in the list, and count is incremented by 1.

virtual void deleteNode(const Type& deleteItem) = 0; //Function
to delete deleteItem from the list.
//Postcondition: If found, the node containing deleteItem is
// deleted from the list. first points to the first node,
// last points to the last node of the updated list, and
// count is decremented by 1.

linkedListIterator<Type> begin();
//Function to return an iterator at the beginning of the //linked
list.
```

```
        //Postcondition: Returns an iterator such that current is set //
        to first.

        linkedListIterator<Type> end();
        //Function to return an iterator one element past the //last
        element of the linked list.
        //Postcondition: Returns an iterator such that current is set //
        to NULL.

        linkedListType();
        //default constructor
        //Initializes the list to an empty state.
        //Postcondition: first = NULL, last = NULL, count = 0;

        linkedListType(const linkedListType<Type>& otherList);
        //copy constructor
        ~linkedListType();
        //destructor
        //Deletes all the nodes from the list.
        //Postcondition: The list object is destroyed.

    protected:
        int count;                  //variable to store the number of list elements
        nodeType<Type> *first;      //pointer to the first node of the list nodeType<Type> *last;
        //pointer to the last node of the list private:
        void copyList(const linkedListType<Type>& otherList); //Function
        to make a copy of otherList.
        //Postcondition: A copy of otherList is created and assigned
        // to this list.
};
```

Figure 3-19 shows the UML class diagram of the class linkedListType.

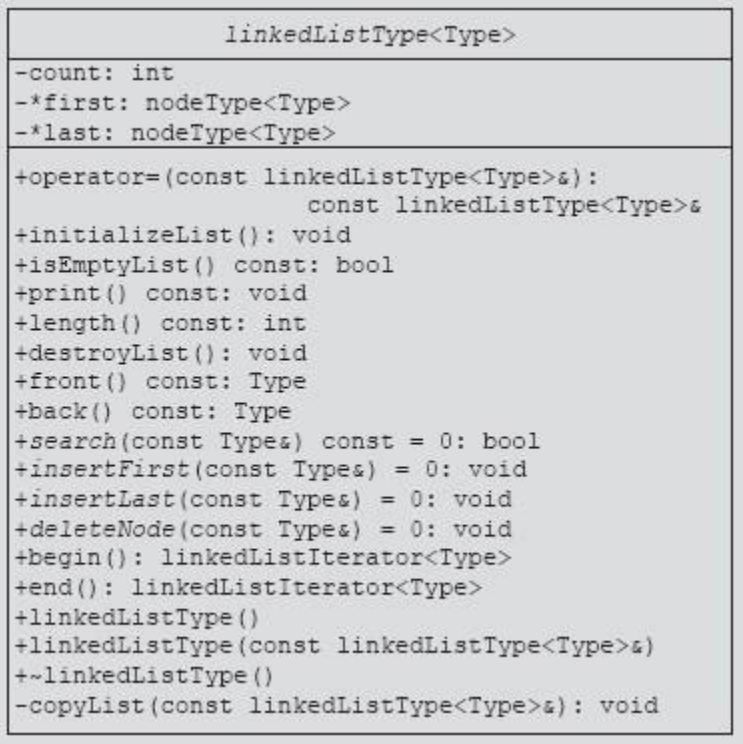


FIGURE 3-19 UML class diagram of the class linkedListType

Note that, typically, in the UML class diagram the names of an abstract class and abstract function are shown in italic.

The instance variables first and last, as defined earlier, of the class linkedListType are protected, not private, because as noted previously, we will derive the classes unorderedLinkedList and orderedLinkedList from the class linkedListType. Because each of the classes unorderedLinkedList and orderedLinkedList will provide separate definitions of the functions search, insertFirst, insertLast, and deleteNode, and because these functions would access the instance variable, to provide direct access to the instance variables, the instance variables are declared as protected.

The definition of the class linkedListType includes a member function to overload the assignment operator. For classes that include pointer data members, the assignment operator must be explicitly overloaded. For the same reason, the definition of the class also includes a copy constructor.

Notice that the definition of the class linkedListType contains the member function copyList, which is declared as a private member. This is because this function is used only to implement the copy constructor and overload the assignment operator.

Next, we write the definitions of the nonabstract functions of the class LinkedListClass.
The list is empty if first is NULL. Therefore, the definition of the function isEmptyList to implement this operation is as follows:

```
template <class Type>
bool linkedListType<Type>::isEmptyList() const
{ return (first == NULL);
}
```

3.4.1 Default Constructor

The default constructor, linkedListType, is quite straightforward. It simply initializes the list to an empty state. Recall that when an object of the linkedListType type is declared and no value is passed, the default constructor is executed automatically.

```
template <class Type>
linkedListType<Type>::linkedListType() //default constructor
{ first = NULL; last =
        NULL;
        count = 0;
}
```

From the definitions of the functions isEmptyList and the default constructor, it follows that each of these functions is of O(1).

3.4.2 Destroy the List

The function destroyList deallocates the memory occupied by each node. We traverse the list starting from the first node and deallocate the memory by calling the operator delete. We need a temporary pointer to deallocate the memory. Once the entire list is destroyed, we must set the pointers first and last to NULL and count to 0.

```
template <class Type>
void linkedListType<Type>::destroyList()
{
    nodeType<Type> *temp;                    //pointer to deallocate the memory
                                             //occupied by the node
    while (first != NULL)                    //while there are nodes in the list
    {
```

```
        temp = first;                    //set temp to the current node
        first = first->link;             //advance first to the next node
        delete temp;                     //deallocate the memory occupied by temp
    }
    last = NULL;                         //initialize last to NULL; first has already
                                         //been set to NULL by the while loop
    count = 0;
}
```

If the list has n items, the while loop executes n times. From this, it follows that the function destroyList is of O(n).

3.4.3 Initialize the List

The function initializeList initializes the list to an empty state. Note that the default constructor or the copy constructor has already initialized the list when the list object was declared. This operation, in fact, reinitializes the list to an empty state, and so it must delete the nodes (if any) from the list. This task can be accomplished by using the destroyList operation, which also resets the pointers first and last to NULL and sets count to 0.

```
template <class Type>
void linkedListType<Type>::initializeList()
{
    destroyList();                       //if the list has any nodes, delete them
}
```

The function initializeList uses the function destroyList, which is of O(n). Therefore, the function initializeList is of O(n).

3.4.4 Print the List

The member function print prints the data contained in each node. To print the data contained in each node, we must traverse the list starting at the first node. Because the pointer first always points to the

first node in the list, we need another pointer to traverse the list. (If we use first to traverse the list, the entire list will be lost.)

```
template <class Type>
void linkedListType<Type>::print() const
{
        nodeType<Type> *current;                    //pointer to traverse the list
        current = first;                            //set current point to the first node
        while (current != NULL)                     //while more data to print
        { cout << current->info << " ";
                current = current->link;
        }
}                                                   //end print
```

As in the case of the function destroyList, the function print is of O(n).

3.4.5 Length of a List

The length of a linked list (that is, how many nodes are in the list) is stored in the variable count. Therefore, this function returns the value of this variable.

```
template <class Type>
int linkedListType<Type>::length() const
{ return count;
}
```

3.1.1 Retrieve the Data of the First Node

The function front returns the info contained in the first node, and its definition is straightforward.

```
template <class Type>
Type linkedListType<Type>::front() const
{ assert(first != NULL);
        return first->info;                    //return the info of the first node
}                                              //end front
```

Notice that if the list is empty, the assert statement terminates the program. Therefore, before calling this function check, you have to check to see whether the list is nonempty.

3.1.2 Retrieve the Data of the Last Node

The function back returns the info contained in the last node. Its definition is as follows:

```
template <class Type>
Type linkedListType<Type>::back() const
{ assert(last != NULL);
        return last->info;                     //return the info of the last node }//end
back
```

Notice that if the list is empty, the assert statement terminates the program. Therefore, before calling this function, you have to check to see whether the list is nonempty.

From the definitions of the functions length, front, and back, it follows easily that each of these functions are of O(1).

3.1.3 Begin and End

The function begin returns an iterator to the first node in the linked list and the function end returns an iterator to the last node in the linked list. Their definitions are as follows:

```
template <class Type>
linkedListIterator<Type>  linkedListType<Type>::begin()
{
        linkedListIterator<Type> temp(first); return
        temp;
}

template <class Type>
linkedListIterator<Type>  linkedListType<Type>::end()
{
        linkedListIterator<Type> temp(NULL); return
        temp;
}
```

From the definitions of the functions length, front, back, begin, and end, it follows easily that each of these functions are of O(1).

3.1.4 Copy the List

The function copyList makes an identical copy of a linked list. Therefore, we traverse the list to be copied starting at the first node. Corresponding to each node in the original list, we do the following:

1. Create a node and call it newNode.
2. Copy the info of the node (in the original list) into newNode.
3. Insert newNode at the end of the list being created.

The definition of the function copyList is as follows:

```
template <class Type> void
linkedListType<Type>::copyList (const
linkedListType<Type>& otherList)
{
        nodeType<Type> *newNode;                    //pointer to create a node
```

```
nodeType<Type> *current;                    //pointer to traverse the list
if (first != NULL)                          //if the list is nonempty, make it empty
        destroyList();
if (otherList.first == NULL)                //otherList is empty
{ first = NULL; last =
        NULL;
        count = 0;
}
else
{
        current    =    otherList.first;    //current points to the //list
        count = otherList.count;            to be copied

                                            //copy the first node
        first = new nodeType<Type>;         //create the node
```

```
            first->info = current->info;                    //copy the info
            first->link = NULL;                             //set the link field of the node to NULL
            last = first;                                   //make last point to the first node
            current = current->link;                        //make current point to the next
                                                            // node
                                                            //copy the remaining list
            while (current != NULL)
            {
                  newNode = new nodeType<Type>;  //create a node newNode->info
                  = current->info;  //copy the info
                  newNode->link = NULL;                     //set the link of newNode to NULL
                  last->link = newNode;                     //attach newNode after last
                  last = newNode;                           //make last point to the actual last
                                                            //node
                  current = current->link;                  //make current point to the
                                                            //next node
            }                                               //end while
      }                                                     //end else
}                                                           //end copyList
```

The function copyList contains a while loop. The number of times the while loop executes depends on the number of items in the list. If the list contains n items, the while loop executes n times. Therefore, the function copyList is of O(n).

3.1.5 Destructor

The destructor deallocates the memory occupied by the nodes of a list when the class object goes out of scope. Because memory is allocated dynamically, resetting the pointers first and last does not deallocate the memory occupied by the nodes in the list. We must traverse the list, starting at the first node, and delete each node in the list. The list can be destroyed by calling the function destroyList. Therefore, the definition of the destructor is as follows:

```
template <class Type>
linkedListType<Type>::~linkedListType()                    //destructor
{ destroyList();
}
```

3.1.6 Copy Constructor

Because the class linkedListType contains pointer data members, the definition of this class contains the copy constructor. Recall that, if a formal parameter is a value parameter, the copy constructor provides the formal parameter with its own copy of the data. The copy constructor also executes when an object is declared and initialized using another object.

The copy constructor makes an identical copy of the linked list. This can be done by calling the function copyList. Because the function copyList checks whether the original is empty by checking the value of first, we must first initialize the pointer first to NULL before calling the function copyList.

The definition of the copy constructor is as follows:

```
template <class Type>
linkedListType<Type>::linkedListType (const
linkedListType<Type>& otherList)
{ first = NULL;
        copyList(otherList);
}                                                    //end copy constructor
```

3.1.7 Overloading the Assignment Operator

The definition of the function to overload the assignment operator for the class linkedListType is similar to the definition of the copy constructor. We give its definition for the sake of completeness.

```
//overload the assignment operator template
<class Type>
const linkedListType<Type>& linkedListType<Type>::operator=(const linkedListType<Type>& otherList)
{
        if (this != &otherList)                    //avoid self-copy
        {
                copyList(otherList);
        }                                          //end else return
        *this;
}
```

The destructor uses the function destroyList, which is of O(n). The copy constructor and the function to overload the assignment operator use the function copyList, which is of O(n). Therefore, each of these functions are of O(n).

TABLE 3-6 Time-complexity of the operations of the class linked ListType

Function	Time-complexity
`isEmptyList`	$O(1)$
default constructor	$O(1)$
`destroyList`	$O(n)$
`front`	$O(1)$
`end`	$O(1)$
`initializeList`	$O(n)$
`print`	$O(n)$
`length`	$O(1)$
`front`	$O(1)$
`back`	$O(1)$
`copyList`	$O(n)$
destructor	$O(n)$
copy constructor	$O(n)$
Overloading the assignment operator	$O(n)$

3.5 Unordered Linked Lists

As described in the previous section, we derive the class unorderedLinkedList from the abstract class linkedListType and implement the operations search, insertFirst, insertLast, and deleteNode.

The following class defines an unordered linked list as an ADT:

```
//*************************************************************
// This class specifies the members to implement the
basic // properties of an unordered linked list. This class is
// derived from the class linkedListType.
//************************************************************* template
<class Type>
```

```
class unorderedLinkedList: public linkedListType<Type>
{ public:
        bool search(const Type& searchItem) const;
        //Function to determine whether searchItem is in the list.
        //Postcondition: Returns true if searchItem is in the list, //
        otherwise the value false is returned.
```

```
        void insertFirst(const Type& newItem);
        //Function to insert newItem at the beginning of the list.
        //Postcondition: first points to the new list, newItem
        is // inserted at the beginning of the list, last points to
        // the last node, and count is incremented by 1.

        void insertLast(const Type& newItem);
        //Function to insert newItem at the end of the list.
        //Postcondition: first points to the new list, newItem
        is // inserted at the end of the list, last points to the //
        last node, and count is incremented by 1.

        void deleteNode(const Type& deleteItem); //Function
        to delete deleteItem from the list.
        //Postcondition: If found, the node containing deleteItem
        // is deleted from the list. first points to the first //
        node, last points to the last node of the updated //
        list, and count is decremented by 1.
};
```

Figure 3-20 shows a UML class diagram of the class unorderedLinkedList and the inheritance hierarchy.

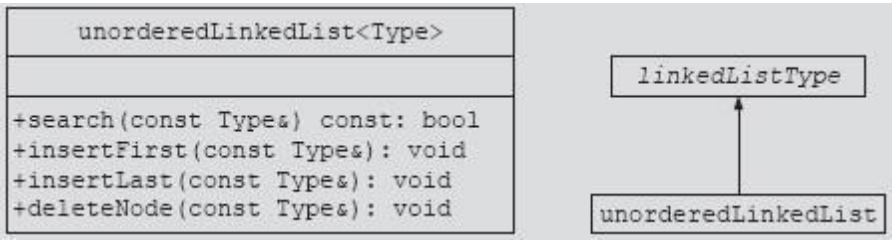


FIGURE 3-20 UML class diagram of the class unorderedLinkedList and the inheritance hierarchy

Next we give the definitions of the member functions of the class unorderedLinkedList.

3.1.8 Search the List

The member function search searches the list for a given item. If the item is found, it returns true; otherwise, it returns false. Because a linked list is not a random access data structure, we must sequentially search the list starting from the first node.

This function has the following steps:

1. Compare the search item with the current node in the list. If the info of the current node is the same as the search item, stop the search; otherwise, make the next node the current node.
2. Repeat Step 1 until either the item is found or no more data is left in the list to compare with the search item.

```
template <class Type>
bool unorderedLinkedList<Type>::search(const Type& searchItem) const
{
```

```
    nodeType<Type> *current;                    //pointer to traverse the list
    bool found = false;
    current = first;                            //set current to point to the first
                                                //node in the list
    while (current != NULL && !found)           //search the list
        if (current->info == searchItem)        //searchItem is found
            found = true;
        else
            current = current->link;            //make current point to
                                                //the next node
        return found;
}                                               //end search
```

The number of times the while loop executes, in the function search, depends on where in the list the search item is located. Suppose the list has n items. If the search item is not in the list, the while loop executes n times. On the other hand, if the search item is the first item, the while loop executes 1 time. Similarly, if the search item is the ith item in the list, the while loop executes i times. From these observations, we can show that the function search is of O(n). We will explicitly analyze the sequential search algorithm in Chapter 7.

3.1.9 Insert the First Node

The function insertFirst inserts the new item at the beginning of the list—that is, before the node pointed to by first. The steps needed to implement this function are as follows:

1. Create a new node.
2. If unable to create the node, terminate the program.
3. Store the new item in the new node.
4. Insert the node before first.
5. Increment count by 1.

```
template <class Type>
void unorderedLinkedList<Type>::insertFirst(const Type& newItem)
```

```
{
    nodeType<Type> *newNode;                    //pointer to create the new node newNode
    = new nodeType<Type>;       //create the new node
    newNode->info = newItem;                    //store the new item in the node
    newNode->link = first;                      //insert newNode before first
    first = newNode;                            //make first point to the actual first node
    count++;                                    //increment count
    if (last == NULL)                           //if the list was empty, newNode is also
                                                //the last node in the list
        last = newNode;
}                                               //end insertFirst
```

3.1.10 Insert the Last Node

The definition of the member function insertLast is similar to the definition of the member function insertFirst. Here, we insert the new node after last. Essentially, the function insertLast is as follows:

```
template <class Type>
void unorderedLinkedList<Type>::insertLast(const Type& newItem)
{
    nodeType<Type> *newNode;          //pointer to create the new node newNode
    = new nodeType<Type>;  //create the new node
    newNode->info = newItem;          //store the new item in the node
    newNode->link = NULL;             //set the link field of newNode to NULL
    if (first == NULL)                //if the list is empty, newNode is
    {                                 //both the first and last node
        first = newNode;
        last = newNode;
        count++;                      //increment count
    }
    else                              //the list is not empty, insert newNode after last
    {
        last->link = newNode;         //insert newNode after last
        last = newNode;               //make last point to the actual
                                      //last node in the list
        count++;                      //increment count
    }
}                                     //end insertLast
```

From the definitions of the functions insertFirst and insertLast, it follows that each of these functions is of O(1).

3.1.11 Delete a Node

Next, we discuss the implementation of the member function deleteNode, which deletes a node from the list with a given info. We need to consider the following cases:

- The list is empty.
- The node is nonempty and the node to be deleted is the first node.
- The node is nonempty and the node to be deleted is not the first node, it is somewhere in the list.
- The node to be deleted is not in the list.

If list is empty, we can simply print a message indicating that the list is empty. If list is not empty, we search the list for the node with the given info and, if such a node is found, we delete this node. After deleting the node, count is decremented by 1. In pseudocode, the algorithm is as follows:

```
if list is empty
        Output(cannot delete from an empty list); else
{ if the first node is the node with the given info
        adjust the head pointer, that is, first, and deallocate
        the memory; else
        { search the list for the node with the given info if
                such a node is found, delete it and adjust the
                values of last (if necessary) and count.
        }
}
```

Case 1: The list is empty. If the list is empty, output an error message as shown in the pseudocode.

Case 2: The list is not empty and the node to be deleted is the first node. This case has two scenarios: list has only one node, and list has more than one node. If list has only one node, then after deletion, the list becomes empty. Therefore, after deletion, both first and last are set to NULL and count is set to 0.

Suppose that the node to be deleted is the first node and list has more than one node. Then after deleting this node, the second node becomes the first node. Therefore, after deleting the node the value of the pointer first changes and it contains the address of the second node.

Case 3: The node to be deleted is not the first node, but is somewhere in the list.

This case has two subcases: (a) the node to be deleted is not the last node, and (b) the node to be deleted is the last node. Let us illustrate the first cases.

Case 3a: The node to be deleted is not the last node.

Consider the list shown in Figure 3-21.

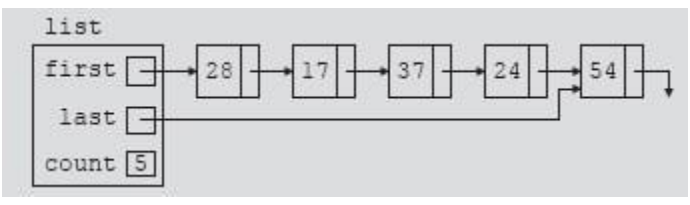


FIGURE 3-21 list before deleting 37

Suppose that the node to be deleted is 37. After deleting this node, the resulting list is as shown in Figure 3-22. (Notice that the deletion of 37 does not require us to change the values of first and last. The link field of the previous node—that is, 17—changes. After deletion, the node with info 17 contains the address of the node with 24.)

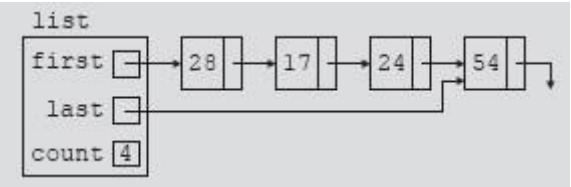


FIGURE 3-22 list after deleting 37

Case 3b: The node to be deleted is the last node. In this case, after deleting the node, the value of the pointer last changes. It contains the address of the node just before the node to be deleted. For example, consider the list given in Figure 3- 21 and the node to be deleted is 54. After deleting 54, last contains the address of the node with info 24. Also, count is decremented by 1.

Case 4: The node to be deleted is not in the list. In this case, the list requires no adjustment. We simply output an error message, indicating that the item to be deleted is not in the list.

From cases 2, 3, and 4, it follows that the deletion of a node requires us to traverse the list. Because a linked list is not a random access data structure, we must sequentially search the list. We handle case 1 separately because it does not require us to traverse the list. We sequentially search the list starting at the second node. If the node to be deleted is in the middle of the list, we need to adjust the link field of the node just before the node to be deleted. Thus, we need a pointer to the previous node. When we search the list for the given info, we use two pointers: one to check the info of the current node, and one to keep track of the node just before the current node. If the node to be deleted is the last node, we must adjust the pointer last.

The definition of the function deleteNode is as follows:

```
template <class Type>
void unorderedLinkedList<Type>::deleteNode(const Type& deleteItem)
{
        nodeType<Type> *current;                    //pointer to traverse the list
        nodeType<Type> *trailCurrent;                       //pointer just before current bool
        found;

        if (first == NULL)                                  //Case 1; the list is empty.
                cout << "Cannot delete from an empty list."<< endl; else
        {
```

```
if (first->info == deleteItem)                    //Case 2
{
        current = first; first
        = first->link;
        count--;

        if (first == NULL)                        //the list has only one node
                last = NULL;


        delete current;
}
```

```
        else                                    //search the list for the node with the given
                                                //info
        {
            found = false;
            trailCurrent = first;               //set trailCurrent to point
                                                //to the first node
            current = first->link;              //set current to point to
                                                //the second node
            while (current != NULL && !found)
            {
                if (current->info != deleteItem)
                { trailCurrent = current;
                    current = current-> link;
                }
                else
                    found = true;
            }                                   //end while
            if (found)                          //Case 3; if found, delete the node
            { trailCurrent->link = current->link;
                count--;

                if (last == current)            //node to be deleted was the
                                                //last node
                    last = trailCurrent;        //update the value of last
                delete current;                 //delete the node from the list
            }
            else
                cout << "The item to be deleted is not in "<< "the list." << endl;
        }                                       //end else
    }                                           //end else
}                                               //end deleteNode
```

From the definition of the function deleteNode, it can be shown that this function is ofO(n).

Table 3-7 gives the time-complexity of the operations of the class unorderedLinkedList.

TABLE 3-7: Time-complexity of the operations of the class unorderedLinkedList

Function	Time-complexity
`search`	$O(n)$
`insertFirst`	$O(1)$
`insertLast`	$O(1)$
`deleteNode`	$O(n)$

3.1.12 Header File of the Unordered Linked List

For the sake of completeness, we show how to create the header file that defines the class unorderedListType and the operations on such lists. (We assume that the definition of the class linkedListType and the definitions of the functions to implement the operations are in the header file linkedList.h.)

```
#ifndef H_UnorderedLinkedList
#define H_UnorderedLinkedList
//***********************************************************
// This class specifies the members to implement the
basic // properties of an unordered linked list. This class is
// derived from the class linkedListType.
//***********************************************************
#include "linkedList.h" using
namespace std; template
<class Type>
class unorderedLinkedList: public linkedListType<Type>
{ public:
        bool search(const Type& searchItem) const;
        //Function to determine whether searchItem is in the list.
        //Postcondition: Returns true if searchItem is in the list, //
        otherwise the value false is returned.

        void insertFirst(const Type& newItem);
        //Function to insert newItem at the beginning of the list.
        //Postcondition: first points to the new list, newItem
        is // inserted at the beginning of the list, last points to
        // the last node, and count is incremented by 1.

        void insertLast(const Type& newItem);
        //Function to insert newItem at the end of the list.
        //Postcondition: first points to the new list, newItem
```

```
                is // inserted at the end of the list, last points to the //
                last node, and count is incremented by 1.

                void deleteNode(const Type& deleteItem); //Function
                to delete deleteItem from the list.
                //Postcondition: If found, the node containing deleteItem
                // is deleted from the list. first points to the first //
                node, last points to the last node of the updated list, //
                and count is decremented by 1.
};

                //Place the definitions of the functions search, insertNode, //insertFirst,
                insertLast, and deleteNode here.
                .
                .
```

.

#endif

3.6 Ordered Linked Lists

The preceding section described the operations on an unordered linked list. This section deals with ordered linked lists. As noted earlier, we derive the class orderedLinkedList from the class linkedListType and provide the definitions of the abstract functions insertFirst, insertLast, search, and deleteNode to take advantage of the fact that the elements of an ordered linked list are arranged using some ordering criteria. For simplicity, we assume that elements of an ordered linked list are arranged in ascending order.

Because the elements of an ordered linked list are in order, we include the function insert to insert an element in an ordered list at the proper place.

The following class defines an ordered linked list as an ADT:

```
//*************************************************************
// This class specifies the members to implement the
basic // properties of an ordered linked list. This class is //
derived from the class linkedListType.
//*************************************************************  template
<class Type>
class orderedLinkedList: public linkedListType<Type>
{ public:
        bool search(const Type& searchItem) const;
        //Function to determine whether searchItem is in the list.
        //Postcondition: Returns true if searchItem is in the list, //
        otherwise the value false is returned.

        void insert(const Type& newItem); //Function
        to insert newItem in the list.
        //Postcondition: first points to the new list, newItem
        // is inserted at the proper place in the list, and //
        count is incremented by 1.
```

```
void insertFirst(const Type& newItem);
//Function to insert newItem at the beginning of the list.
//Postcondition: first points to the new list, newItem is
// inserted at the beginning of the list, last points to the //
last node in the list, and count is incremented by 1.

void insertLast(const Type& newItem);
//Function to insert newItem at the end of the list.
//Postcondition: first points to the new list, newItem is //
inserted at the end of the list, last points to the
// last node in the list, and count is incremented by 1.

void deleteNode(const Type& deleteItem); //Function
to delete deleteItem from the list.
//Postcondition: If found, the node containing deleteItem is
```

```
        // deleted from the list; first points to the first node
        // of the new list, and count is decremented by 1. If //
        deleteItem is not in the list, an appropriate message
        // is printed.
};
```

Figure 3-23 shows a UML class diagram of the class orderedLinkedList and the inheritance hierarchy.

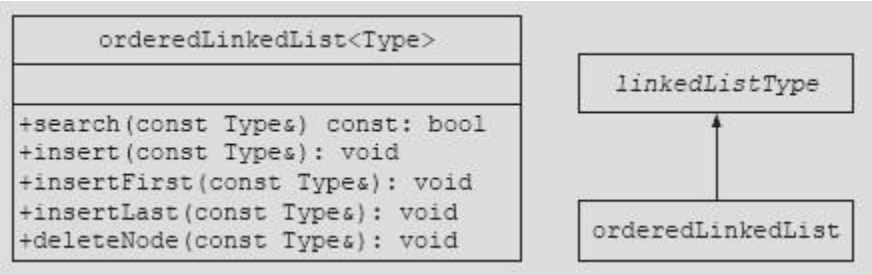


FIGURE 3-23 UML class diagram of the class orderedLinkedList and the inheritance hierarchy

Next we give the definitions of the member functions of the class orderedLinkedList.

3.1.13 Search the List

First, we discuss the search operation. The algorithm to implement the search operation is similar to the search algorithm for general lists discussed earlier. Here, because the list is sorted, we can improve the search algorithm somewhat. As before, we start the search at the first node in the list. We stop the search as soon as we find a node in the list with info greater than or equal to the search item, or we have searched the entire list.

The following steps describe this algorithm:

1. Compare the search item with the current node in the list. If the info of the current node is greater than or equal to the search item, stop the search; otherwise, make the next node the current node.
2. Repeat Step 1 until either an item in the list that is greater than or equal to the search item is found, or no more data is left in the list to compare with the search item.

Note that the loop does not explicitly check whether the search item is equal to an item in the list. Thus, after the loop executes, we must check whether the search item is equal to the item in the list.

```
template <class Type>
bool orderedLinkedList<Type>::search(const Type& searchItem) const
{
        bool found = false;
        nodeType<Type> *current;                        //pointer to traverse the list
        current = first;                                //start the search at the first
        node while (current != NULL && !found) if (current->info >= searchItem) found = true;
        else current = current->link;
        if (found) found = (current->info == searchItem);  //test for equality
        return found;
}                                                       //end search
```

As in the case of the search function of the class unorderedLinkedList, the search function of the class orderedLinkedList is also of O(n).

3.1.14 Insert a Node

To insert an item in an ordered linked list, we first find the place where the new item is supposed to go, then we insert the item in the list. To find the place for the new item in the list, as before, we search the list. Here we use two pointers, current and trailCurrent, to search the list. The pointer current points to the node whose info is being compared with the itemto be inserted, and trailCurrent points to the node just before current. Because the list is in order, the search algorithm is the same as before. The following cases arise:

Case 1: The list is initially empty. The node containing the new item is the only node and, thus, the first node in the list.

Case 2: The new item is smaller than the smallest item in the list. The new item goes at the beginning of the list. In this case, we need to adjust the list's head pointer— that is, first. Also, count is incremented by 1.

Case 3: The item is to be inserted somewhere in the list.

Case 3a: The new item is larger than all the items in the list. In this case, the new item is inserted at the end of the list. Thus, the value of current is NULL and the new item is inserted after trailCurrent. Also, count is incremented by 1.

Case 3b: The new item is to be inserted somewhere in the middle of the list. In this case, the new item is inserted between trailCurrent and current. Also, count is incremented by 1.

The following statements can accomplish both cases 3a and 3b. Assume newNode points to the new node.

```
trailCurrent->link = newNode; newNode->link
= current;
```

Let us next illustrate Case 3.

Case 3: The list is not empty, and the item to be inserted is larger than the first item in the list. As indicated previously, this case has two scenarios.

Case 3a: The item to be inserted is larger than the largest item in the list; that is, it goes at the end of the list. Consider the list shown in Figure 3-24.

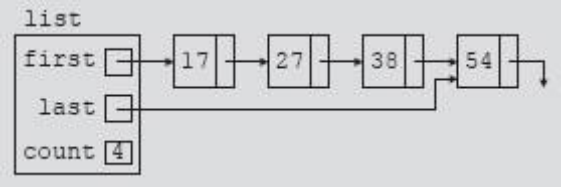


FIGURE 3-24 list before inserting 65

Suppose that we want to insert 65 in the list. After inserting 65, the resulting list is as shown in Figure 3-25.

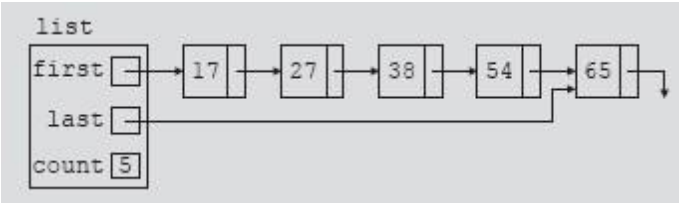


FIGURE 3-25 list after inserting 65

Case 3b: The item to be inserted goes somewhere in the middle of the list. Again consider the list shown in Figure 3-24. Suppose that we want to insert 25 in this list. Clearly, 25 goes between 17 and 27, which would require the link of the node with info 17 to be changed. After inserting 25, the resulting list is as shown in Figure 3-26.

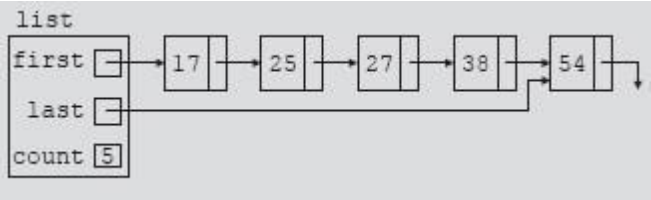


FIGURE 3-26 list after inserting 25 in the list in Figure 5-24

From case 3, it follows that we must first traverse the list to find the place where the new item is to be inserted. It also follows that we should traverse the list with two pointers— say, current and trailCurrent. The pointer current is used to traverse the list and compare the info of the node in the list with the item to be inserted. The pointer trailCurrent points to the node just before current. For example, in case 3b, when the search stops, trailCurrent points to node 17 and current points to node 27. The item is inserted after trailCurrent. In case 3a, after searching the list to find the place for 65, trailCurrent points to node 54 and current is NULL.

The definition of the function insert is as follows:

```
template <class Type>
void orderedLinkedList<Type>::insert(const Type& newItem)
{
        nodeType<Type> *current;              //pointer to traverse the list nodeType<Type>
        *trailCurrent;          //pointer just before current nodeType<Type> *newNode;
                //pointer to create a node bool found;

        newNode = new nodeType<Type>;              //create the node
```

```
newNode->info = newItem;          //store newItem in the node
newNode->link = NULL;             //set the link field of the node
                                  //to NULL
```

```
if (first == NULL)                          //Case 1
{
        first = newNode;
        last = newNode;
        count++;
}
else
{ current = first;
        found = false;

        while (current != NULL && !found)  //search the list
                if (current->info >= newItem) found = true;
                else
                { trailCurrent = current;
                        current = current->link;
                }

        if (current == first)                   //Case 2
        {
                newNode->link = first; first
                = newNode;
                count++;
        }
        else                                    //Case 3
        {
                trailCurrent->link = newNode;
                newNode->link = current;

                if (current == NULL) last
                        = newNode;
                count++;
        }
```

```
        }                                       //end else
}                                               //end insert
```

The function insert uses a while loop to find the place where the new item is to be inserted and this loop is similar to the while loop used in the search function. It can be shown that the function insert is of O(n).

3.1.15 Insert First and Insert Last

The function insertFirst inserts the new item at the beginning of the list. However, because the resulting list must be sorted, the new item must be inserted at the proper place. Similarly, the function insertLast must insert the new item at the proper place. We, therefore, use the function insert to insert the new item at its proper place. The definitions of these functions are as follows:

```
template <class Type>
void orderedLinkedList<Type>::insertFirst(const Type& newItem)
```

```
{
        insert(newItem);
}               //end insertFirst

template <class Type>
void orderedLinkedList<Type>::insertLast(const Type& newItem)
{
        insert(newItem);
}               //end insertLast
```

Note that in reality, the functions insertFirst and insertLast do not apply to an ordered linked list because the new item must be inserted at the proper place in the list. However, you must provide its definition as these functions are declared as abstract in the parent class.

The functions insertFirst and insertLast use the function insert, which is of O(n). It follows that these functions are of O(n).

3.1.16 Delete a Node

To delete a given item from an ordered linked list, first we search the list to see whether the item to be deleted is in the list. The function to implement this operation is the same as the delete operation on general linked lists. Here, because the list is sorted, we can somewhat improve the algorithm for ordered linked lists.

As in the case of insertNode, we search the list with two pointers, current and trailCurrent. Similar to the operation insertNode, several cases arise:

Case 1: The list is initially empty.We have an error. We cannot delete from an empty list.

Case 2: The item to be deleted is contained in the first node of the list. We must adjust the head pointer of the list—that is, first.

Case 3: The item to be deleted is somewhere in the list. In this case, current points to the node containing the item to be deleted, and trailCurrent points to the node just before the node

pointed to by current.

Case 4: The list is not empty, but the item to be deleted is not in the list.

After deleting a node, count is decremented by 1. The definition of the function deleteNode is as follows:

```
template <class Type>
void orderedLinkedList<Type>::deleteNode(const Type& deleteItem)
{
    nodeType<Type> *current;                    //pointer to traverse the list
    nodeType<Type> *trailCurrent;                    //pointer just before current bool
    found;

    if (first == NULL)                                    //Case 1
        cout << "Cannot delete from an empty list." << endl;
    else
```

```
{
        current = first;
        found = false;

        while (current != NULL && !found)    //search the list if
                (current->info >= deleteItem) found = true;
                else
                {
                        trailCurrent = current;
                        current = current->link;
                }

        if (current == NULL)                        //Case 4
                cout << "The item to be deleted is not in the list."<< endl;
        else if (current->info == deleteItem)  //the item to be
                                                    //deleted is in the list
                {
                        if (first == current)           //Case 2
                        {
                                first = first->link;

                                if (first == NULL) last
                                        = NULL;

                                delete current;
                        }
                        else                            //Case 3
                        {
                                trailCurrent->link = current->link;

                                if (current == last) last =
                                        trailCurrent;
                                delete current;
```

```
                }
                count--;
            }
            else                                    //Case 4
                cout << "The item to be deleted is not in the "<< "list." << endl;
        }
}                                                   //end deleteNode
```

From the definition of the function deleteNode, it can be shown that this function is of O(n).

Table 3-8 gives the time-complexity of the operations of the class orderedLinkedList.

TABLE 3-8: Time-complexity of the operations of the class orderedLinkedList

Function	Time-complexity
search	$O(n)$
insert	$O(n)$
insertFirst	$O(n)$
insertLast	$O(n)$
deleteNode	$O(n)$

3.1.17 Header File of the Ordered Linked List

For the sake of completeness, we show how to create the header file that defines the class orderedListType and the operations on such lists. (We assume that the definition of the class linkedListType and the definitions of the functions to implement the operations are in the header file linkedList.h.)

```
#ifndef H_orderedListType
#define H_orderedListType
//***********************************************************
// This class specifies the members to implement the
basic // properties of an ordered linked list. This class is //
derived from the class linkedListType.
//***********************************************************
#include "linkedList.h" using
namespace std; template
<class Type>
class orderedLinkedList: public linkedListType<Type>
{ public:
        bool search(const Type& searchItem) const;
        //Function to determine whether searchItem is in the list.
        //Postcondition: Returns true if searchItem is in the list, //
        otherwise the value false is returned.
```

```
void insert(const Type& newItem); //Function
to insert newItem in the list.
//Postcondition: first points to the new list, newItem
is // inserted at the proper place in the list, and count
// is incremented by 1.

void insertFirst(const Type& newItem);
//Function to insert newItem at the beginning of the list.
//Postcondition: first points to the new list, newItem is
// inserted at the beginning of the list, last points to the //
last node in the list, and count is incremented by 1.
```

```
        void insertLast(const Type& newItem);
        //Function to insert newItem at the end of the list.
        //Postcondition: first points to the new list, newItem
        is // inserted at the end of the list, last points to the //
        last node in the list, and count is incremented by 1.

        void deleteNode(const Type& deleteItem); //Function
        to delete deleteItem from the list.
        //Postcondition: If found, the node containing deleteItem is
        // deleted from the list; first points to the first node of
        // the new list, and count is decremented by 1. If //
        deleteItem is not in the list, an appropriate message
        // is printed.
};

        //Place the definitions of the functions search, insert, //insertfirst,
        insertLast, and deleteNode here.
        .
        .
        .
#endif
```

The following program tests various operations on an ordered linked list:

```
//*********************************************************
// This program tests the various operations on an ordered
// linked list.
//*********************************************************
#include <iostream>                                          //Line 1
#include "orderedLinkedList.h"                               //Line 2
using namespace std;                                         //Line 3
```

```
int main()                                                              //Line 4
{
    orderedLinkedList<int> list1, list2;                                //Line 5
    int num;                                                            //Line 6
    cout << "Line 7: Enter numbers ending "<< "with -999." << endl;     //Line 7
    cin >> num;                                                         //Line 8
    while (num != -999)                                                 //Line 9
    {                                                                   //Line 10
        list1.insert(num);                                              //Line 11
```

```
        cin >> num;                                                    //Line 12
    }                                                                  //Line 13
    cout << endl;                                                      //Line 14
    cout << "Line 15: list1: ";      //Line 15 list1.print();        //Line 16 cout << endl;        //Line
    17 list2 = list1;     //test the assignment operator Line 18 cout << "Line 19: list2: ";      //Line
    19 list2.print();          //Line 20 cout << endl;          //Line 21 cout << "Line 22: Enter the
    number to be "<< "deleted: ";   //Line 22 cin >> num;      //Line 23 cout << endl;      //Line 24
    list2.deleteNode(num);      //Line 25 cout << "Line 26: After deleting "<< num << ", list2: " <<
    endl;    //Line 26 list2.print();                //Line 27 cout << endl;          //Line 28 return 0;
    //Line 29
}                                                                      //Line 30
```

Sample Run: In this sample run, the user input is shaded:

Line 7: Enter numbers ending with -999.
23 65 34 72 12 82 36 55 29 -999

Line 15: list1: 12 23 29 34 36 55 65 72 82

Line 19: list2: 12 23 29 34 36 55 65 72 82

Line 22: Enter the number to be deleted: 34

Line 26: After deleting 34, list2:
12 23 29 36 55 65 72 82

The preceding output is self-explanatory. The details are left as an exercise for you.

3.7 Doubly Linked Lists

A doubly linked list is a linked list in which every node has a next pointer and a back pointer. In other words, every node contains the address of the next node (except the last node), and every node contains the address of the previous node (except the first node).(See Figure 3-27.)

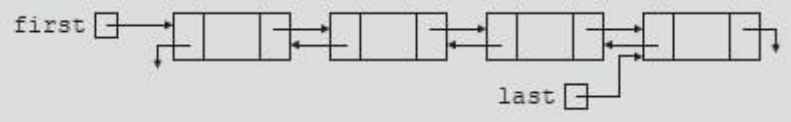


FIGURE 3-27 Doubly linked list

A doubly linked list can be traversed in either direction. That is, we can traverse the list starting at the first node or, if a pointer to the last node is given, we can traverse the list starting at the last node. As before, the typical operations on a doubly linked list are as follows: Initialize the list, destroy the list, determine whether the list is empty, search the list for a given item, insert an item, delete an item, and so on. The following class defines a doubly linked list as an ADT and specifies the basic operations on a doubly linked list:

```
//*************************************************************
* // This class specifies the members to implement the basic // properties
of an ordered doubly linked list.
//*************************************************************
//Definition of the node template
<class Type>
struct nodeType
{
        Type info; nodeType<Type>
        *next;
        nodeType<Type> *back;
};

template <class Type> class
doublyLinkedList
{
        public:
                const doublyLinkedList<Type>& operator=(const doublyLinkedList<Type> &); //Overload
                the assignment operator.

                void initializeList();
                //Function to initialize the list to an empty state.
                //Postcondition: first = NULL; last = NULL; count = 0;

                bool isEmptyList() const;
                //Function to determine whether the list is empty.
                //Postcondition: Returns true if the list is empty, //
                otherwise returns false.

                void destroy();
                //Function to delete all the nodes from the list.
                //Postcondition: first = NULL; last = NULL; count = 0;
```

void print() const;
//Function to output the info contained in each node.

void reversePrint() const;
//Function to output the info contained in each node //in reverse order.

int length() const;
//Function to return the number of nodes in the list.
//Postcondition: The value of count is returned.

Type front() const;
//Function to return the first element of the list.

```
//Precondition: The list must exist and must not be empty.
//Postcondition: If the list is empty, the program terminates; //
otherwise, the first element of the list is returned.
Type back() const;
//Function to return the last element of the list.
//Precondition: The list must exist and must not be empty.
//Postcondition: If the list is empty, the program terminates; //
otherwise, the last element of the list is returned.

bool search(const Type& searchItem) const;
//Function to determine whether searchItem is in the list.
//Postcondition: Returns true if searchItem is found in the //
list, otherwise returns false.

void insert(const Type& insertItem);
//Function to insert insertItem in the list.
//Precondition: If the list is nonempty, it must be in order.
//Postcondition: insertItem is inserted at the proper place
// in the list, first points to the first node, last points
// to the last node of the new list, and count is //
incremented by 1.

void deleteNode(const Type& deleteItem); //Function
to delete deleteItem from the list.
//Postcondition: If found, the node containing deleteItem is
// deleted from the list; first points to the first node of
// the new list, last points to the last node of the new
// list, and count is decremented by 1; otherwise an //
appropriate message is printed.

doublyLinkedList();
//default constructor
//Initializes the list to an empty state.
```

```
        //Postcondition: first = NULL; last = NULL; count = 0;

        doublyLinkedList(const doublyLinkedList<Type>& otherList);
        //copy constructor

        ~doublyLinkedList();
        //destructor
        //Postcondition: The list object is destroyed.

protected:
        int count;
        nodeType<Type> *first;          //pointer to the first node
nodeType<Type> *last;  //pointer to the last node private:
        void copyList(const doublyLinkedList<Type>& otherList); //Function
        to make a copy of otherList.
```

```
        //Postcondition: A copy of otherList is created and assigned
        // to this list.
};
```

We leave the UML class diagram of the class doublyLinkedList as an exercise for you.

The functions to implement the operations of a doubly linked list are similar to the ones discussed earlier. Here, because every node has two pointers, back and next, some of the operations require the adjustment of two pointers in each node. For the insert and delete operations, because we can traverse the list in either direction, we use only one pointer to traverse the list. Let us call this pointer current. We can set the value of trailCurrent by using both the current pointer and the back pointer of the node pointed to by current. We give the definition of each function here, with four exceptions. Definitions of the functions copyList, the copy constructor, overloading the assignment operator, and the destructor are left as exercises for you. Furthermore, the function copyList is used only to implement the copy constructor and overload the assignment operator.

3.5.1 Default Constructor

The default constructor initializes the doubly linked list to an empty state. It sets first and last to NULL and count to 0.

```
template <class Type>
doublyLinkedList<Type>::doublyLinkedList()
{ first= NULL; last =
        NULL;
        count = 0;
}
```

3.5.2 isEmptyList

This operation returns true if the list is empty; otherwise, it returns false. The list is empty if the pointer first is NULL.

```
template <class Type>
```

```
bool doublyLinkedList<Type>::isEmptyList() const
{ return (first == NULL);
}
```

3.5.3 Destroy the List

This operation deletes all the nodes in the list, leaving the list in an empty state.We traverse the list starting at the first node and then delete each node. Furthermore, count is set to 0.

```
template <class Type>
void doublyLinkedList<Type>::destroy()
{
        nodeType<Type> *temp; //pointer to delete the node while
        (first != NULL)
```

```
        { temp = first; first =
                first->next;
                delete temp;
        }
        last = NULL;
        count = 0;
}
```

3.5.4 Initialize the List

This operation reinitializes the doubly linked list to an empty state. This task can be done by using the operation destroy. The definition of the function initializeList is as follows:

```
template <class Type>
void doublyLinkedList<Type>::initializeList()
{ destroy();
}
```

3.5.5 Length of the List

The length of a linked list (that is, how many nodes are in the list) is stored in the variable count. Therefore, this function returns the value of this variable.

```
template <class Type>
int doublyLinkedList<Type>::length() const
{ return count;
}
```

3.5.6 Print the List

The function print outputs the info contained in each node. We traverse the list starting from the first node.

```
template <class Type>
void doublyLinkedList<Type>::print() const
{
        nodeType<Type> *current;                    //pointer to traverse the list
        current = first;                            //set current to point to the first node
        while (current != NULL)
        {
                cout << current->info << " ";       //output info
                current = current->next;
        }                                           //end while
}                                                   //end print
```

3.5.7 Reverse Print the List

This function outputs the info contained in each node in reverse order. We traverse the list in reverse order starting from the last node. Its definition is as follows:

```
template <class Type>
void doublyLinkedList<Type>::reversePrint() const
{
        nodeType<Type> *current;                    //pointer to traverse the list
        current = last;                             //set current to point to the last node
        while (current != NULL)
        { cout << current->info << " ";
                current = current->back;
        }                                           //end while
}                                                   //end reversePrint
```

3.5.8 Search the List

The function search returns true if searchItem is found in the list; otherwise, it returns false.The search algorithm is exactly the same as the search algorithm for an ordered linked list.

```
template <class Type>
bool doublyLinkedList<Type>::search(const Type& searchItem) const
{
        bool found = false;
        nodeType<Type> *current;    //pointer to traverse the list current = first;
        while (current != NULL && !found) if
                (current->info >= searchItem)
                found = true; else current =
                current->next;
```

```
    if (found) found = (current->info == searchItem);  //test for equality
    return found;
}                                                      //end search
```

3.5.9 First and Last Elements

The function front returns the first element of the list and the function back returns the last element of the list. If the list is empty, both functions terminate the program. Their definitions are as follows:

```
template <class Type>
Type doublyLinkedList<Type>::front() const
{ assert(first != NULL);
```

```
        return first->info;
}
template <class Type>
Type doublyLinkedList<Type>::back() const
{ assert(last != NULL);
        return last->info;
}
```

3.5.10 Insert a Node

Because we are inserting an item in a doubly linked list, the insertion of a node in the list requires the adjustment of two pointers in certain nodes. As before, we find the place where the new item is supposed to be inserted, create the node, store the new item, and adjust the link fields of the new node and other particular nodes in the list.

There are four cases:

Case 1:	Insertion in an empty list
Case 2:	Insertion at the beginning of a nonempty list
Case 3:	Insertion at the end of a nonempty list
Case 4:	Insertion somewhere in a nonempty list

Both cases 1 and 2 require us to change the value of the pointer first. Cases 3 and 4 are similar. After inserting an item, count is incremented by 1. Next, we show case 4.

Consider the doubly linked list shown in Figure 3-28.

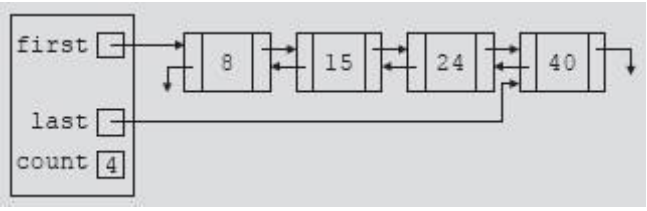


FIGURE 3-28 Doubly linked list before inserting 20

Suppose that 20 is to be inserted in the list. After inserting 20, the resulting list is as shown in Figure 3-29.

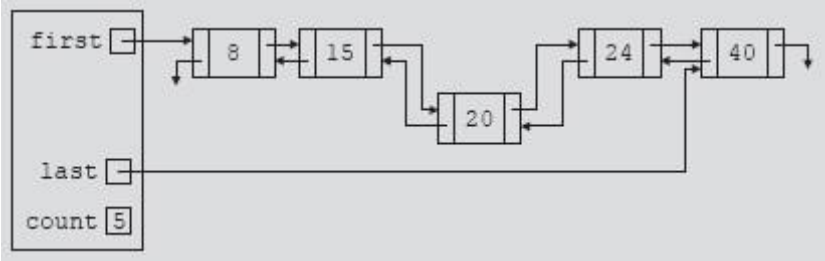


FIGURE 3-29 Doubly linked list after inserting 20

From Figure 3-29, it follows that the next pointer of node 15, the back pointer of node 24, and both the next and back pointers of node 20 need to be adjusted.

The definition of the function insert is as follows:

```
template <class Type>
void doublyLinkedList<Type>::insert(const Type& insertItem)
{
    nodeType<Type> *current;                    //pointer to traverse the list
    nodeType<Type> *trailCurrent;               //pointer just before current
    nodeType<Type> *newNode;                    //pointer to create a node
    bool found;

    newNode = new nodeType<Type>;               //create the node
    newNode->info = insertItem;                 //store the new item in the node
    newNode->next = NULL;
    newNode->back = NULL;

    if (first == NULL)                          //if list is empty, newNode is the only node
    {
        first = newNode;
        last = newNode;
        count++;
    }
    else
    { found = false;
        current =
        first;

        while (current != NULL && !found)  //search the list
            if (current->info >= insertItem) found = true;
            else
            { trailCurrent = current;
                current = current->next;
            }
        if (current == first)                   //insert newNode before first
```

```
{       first->back = newNode;
        newNode->next = first;
        first = newNode; count++;


}
else
{                                       //insert newNode between trailCurrent and current
        if (current != NULL)
        {
                trailCurrent->next = newNode;
                newNode->back = trailCurrent;
                newNode->next = current;
                current->back = newNode;
        }
```

```
                else
                {
                        trailCurrent->next = newNode;
                        newNode->back = trailCurrent;
                        last = newNode;
                }

                count++;
            }                                   //end else
        }                                       //end else
}                                               //end insert
```

3.5.11 Delete a Node

This operation deletes a given item (if found) from the doubly linked list. As before, we first search the list to see whether the item to be deleted is in the list. The search algorithm is the same as before. Similar to the insert operation, this operation (if the item to be deleted is in the list) requires the adjustment of two pointers in certain nodes. The delete operation has several cases:

Case 1: The list is empty.

Case 2: The item to be deleted is in the first node of the list, which would require us to change the value of the pointer first.

Case 3: The item to be deleted is somewhere in the list.

Case 4: The item to be deleted is not in the list.

After deleting a node, count is decremented by 1. Let us demonstrate case 3. Consider the list shown in Figure 3-30.

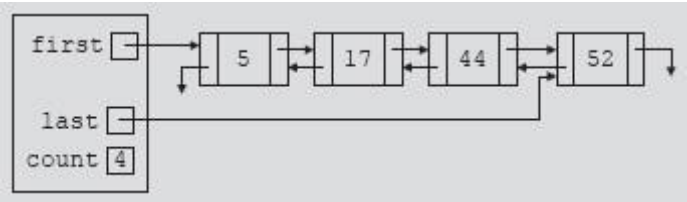


FIGURE 3-30 Doubly linked list before deleting 17

Suppose that the item to be deleted is 17. First, we search the list with two pointers and find the node

with info 17, and then adjust the link field of the affected nodes. (See Figure 3-31.)

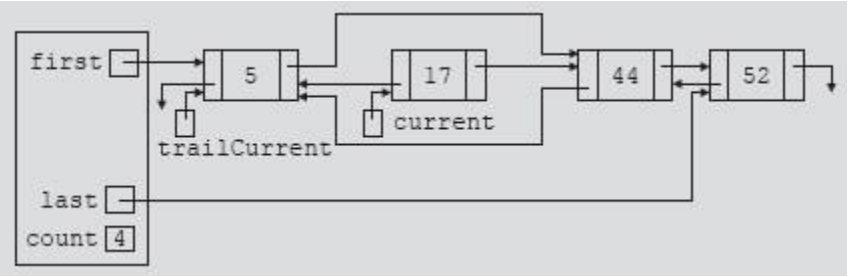


FIGURE 3-31 List after adjusting the links of the nodes before and after the node with info 17

Next, we delete the node pointed to by current. (See Figure 3-32.)

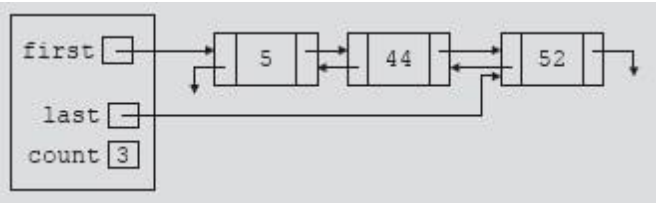


FIGURE 3-32 List after deleting the node with info 17

The definition of the function deleteNode is as follows:

```
template <class Type>
void doublyLinkedList<Type>::deleteNode(const Type& deleteItem)
{
        nodeType<Type> *current;                    //pointer to traverse the list
        nodeType<Type> *trailCurrent;                       //pointer just before current bool
        found;

        if (first == NULL) cout << "Cannot delete from an empty list."
                << endl;
        else if (first->info == deleteItem)                 //node to be deleted is the first node
        { current = first; first =
                first->next;

                if (first != NULL) first-
                >back = NULL; else last =
                NULL;

                count--;
                delete current;
        }
        else
        {
                found = false;
                current = first;

                while (current != NULL && !found)   //search the list if
```

```
(                       t->info >= deleteItem) found = true; else current
c                       = current->next;
u
r               if (current == NULL) cout << "The item to be deleted is not in "<< "the
r                       list." << endl;
e               else if (current->info == deleteItem)           //check for equality
n               {
                        trailCurrent = current->back; trailCurrent-
                        >next = current->next; if (current->next !=
                        NULL) current->next->back = trailCurrent;
                        if (current == last)
                                last = trailCurrent;

                        count--;
```

```
                delete current;
            }
            else
                cout << "The item to be deleted is not in list." endl;
        }                                                   //end else
}                                                           //end deleteNode
```

3.8 STL Sequence Container: list

Chapter 2 listed three types of sequence containers—vector, deque, and list. The sequence containers vector and deque are described in Chapter 2. This section describes the STL sequence container list. List containers are implemented as doubly linked lists. Thus, every element in a list points to its immediate predecessor and to its immediate successor (except the first and last elements). Recall that a linked list is not a random access data structure such as an array. Therefore, to access, for example, the fifth element in the list, we must first traverse the first four elements.

The name of the class containing the definition of the class list is list. The definition of the class list, and the definitions of the functions to implement the various operations on a list, are contained in the header file list. Therefore, to use list in a program, the program must include the following statement:

```
#include <list>
```

Like other container classes, the class list contains several constructors. Thus, a list object can be initialized in several ways when it is declared, as described in Table 3-9.

TABLE 3-9: Various ways to declare a list object

Statement	Description
`list<elemType> listCont;`	Creates the empty `list` container `listCont`. (The default constructor is invoked.)
`list<elemType> listCont(otherList);`	Creates the `list` container `listCont` and initializes it to the elements of `otherList`. `listCont` and `otherList` are of the same type.
`list<elemType> listCont(size);`	Creates the `list` container `listCont` of size `size`. `listCont` is initialized using the default constructor.
`list<elemType> listCont(n, elem);`	Creates the `list` container `listCont` of size `n`. `listCont` is initialized using `n` copies of the element `elem`.
`list<elemType> listCont(beg, end);`	Creates the `list` container `listCont`. `listCont` is initialized to the elements in the range `[beg, end)`, that is, all the elements in the range `beg...end-1`. Both `beg` and `end` are iterators.

Table 3-5 describes the operations that are common to all containers, and Table 3-6 describes the operations that are common to all sequence containers. In addition to these common operations, Table 3-10 describes the operations that are specific to a list container. The name of the function implementing the operation is shown in bold. (Suppose that listCont is a container of type list.)

TABLE 3-10: Operations specific to a list container

Expression	Description
`listCont.assign(n, elem)`	Assigns `n` copies of `elem`.
`listCont.assign(beg, end)`	Assigns all the elements in the range `beg...end-1`. Both `beg` and `end` are iterators.
`listCont.push_front(elem)`	Inserts `elem` at the beginning of `listCont`.
`listCont.pop_front()`	Removes the first element from `listCont`.
`listCont.front()`	Returns the first element. (Does not check whether the container is empty.)
`listCont.back()`	Returns the last element. (Does not check whether the container is empty.)
`listCont.remove(elem)`	Removes all the elements that are equal to `elem`.
`listCont.remove_if(oper)`	Removes all the elements for which `oper` is `true`.
`listCont.unique()`	If the consecutive elements in `listCont` have the same value, removes the duplicates.
`listCont.unique(oper)`	If the consecutive elements in `listCont` have the same value, removes the duplicates, for which `oper` is `true`.
`listCont1.splice(pos, listCont2)`	All the elements of `listCont2` are moved to `listCont1` before the position specified by the iterator `pos`. After this operation, `listCont2` is empty.

TABLE 3-10: Operations specific to a list container (continued)

Expression	Description
`listCont1.splice(pos, listCont2, pos2)`	All the elements starting at `pos2` of `listCont2` are moved to `listCont1` before the position specified by the iterator `pos`.
`listCont1.splice(pos, listCont2, beg, end)`	All the elements in the range `beg...end-1` of `listCont2` are moved to `listCont1` before the position specified by the iterator `pos`. Both `beg` and `end` are iterators.
`listCont.sort()`	The elements of `listCont` are sorted. The sort criterion is `<`.
`listCont.sort(oper)`	The elements of `listCont` are sorted. The sort criterion is specified by `oper`.
`listCont1.merge(listCont2)`	Suppose that the elements of `listCont1` and `listCont2` are sorted. This operation moves all the elements of `listCont2` into `listCont1`. After this operation, the elements in `listCont1` are sorted. Moreover, after this operation, `listCont2` is empty.
`listCont1.merge(listCont2, oper)`	Suppose that the elements of `listCont1` and `listCont2` are sorted according to the sort criteria `oper`. This operation moves all the elements of `listCont2` into `listCont1`. After this operation, the elements in `listCont1` are sorted according to the sort criteria `oper`.
`listCont.reverse()`	The elements of `listCont` are reversed.

Example 3-1 shows how to use various operations on a list container.

EXAMPLE 3-1

```
//*************************************************************
// This program illustrates how to use a list container in a //
program.
//*************************************************************
#include <iostream>                                          //Line 1
```

```
#include <list>                                    //Line 2
#include <iterator>                                //Line 3
#include <algorithm>                               //Line 4
using namespace std;                               //Line 5
int main()                                         //Line 6
```

```
{                                                                    //Line 7
    list<int> intList1, intList2;                                    //Line 8
    ostream_iterator<int> screen(cout, " ");                         //Line 9
    intList1.push_back(23);          //Line 10 intList1.push_back(58);          //Line 11
    intList1.push_back(58);          //Line 12 intList1.push_back(36);          //Line 13 intList1
    .push_back(15);      //Line 14 intList1.push_back(98);      //Line 15 intList1.push_back(58);
    //Line 16 cout << "Line 17: intList1: ";          //Line 17 copy(intList1.begin(), intList1.end(),
    screen);    //Line 18 cout << endl;          //Line 19 intList2 = intList1;      //Line 20 cout <<
    "Line 21: intList2: ";    //Line 21 copy(intList2.begin(), intList2.end(), screen);   //Line 22 cout
    << endl;              //Line 23 intList1.unique();      //Line 24
    cout << "Line 25: After removing the consecutive "<< "duplicates," <<endl
         <<" intList1:";         //Line 25 copy(intList1.begin(), intList1.end(), screen);  //Line 26
    cout << endl;    //Line 27 intList2.sort();    //Line 28 cout << "Line 29: After sorting, intList2:
    ";      //Line 29 copy(intList2.begin(), intList2.end(), screen);      //Line 30 cout << endl;
            //Line 31 return 0;         //Line 32
}                                                                    //Line 33
```

Sample Run:

Line 17: intList1: 23 58 58 36 15 98 58

Line 21: intList2: 23 58 58 36 15 98 58

Line 25: After removing the consecutive duplicates,
intList1: 23 58 36 15 98 58

Line 29: After sorting, intList2: 15 23 36 58 58 58 98

For the most part, the output of the preceding program is straightforward. The statements in Lines 10 through 16 insert the element numbers 23, 58, 58, 36, 15, 98, and 58 (in that order) into intList1. The statement in Line 20 copies the elements of intList1 into intList2. After this statement executes, intList1 and intList2 are identical. The statement in Line 24 removes any consecutive occurrences of the same elements. For example, the number 58 appears consecutively two times. The operation unique removes two occurrences of 58. Note that this operation has no effect on the 58 that appears at the end of intList1. The statement in Line 28 sorts intList2.

3.9 Linked Lists with Header and Trailer Nodes

When inserting and deleting items from a linked list (especially an ordered list), we saw that there are special cases, such as inserting (or deleting) at the beginning (the first node) of the list or in an empty list. These cases needed to be handled separately. As a result, the insertion and deletion algorithms were not as simple and straightforward as we would like.One way to simplify these algorithms is to never insert an item before the first or last item and to never delete the first node. Next we discuss how to accomplish this.

Suppose the nodes of a list are in order; that is, they are arranged with respect to a given key. Suppose it is possible for us to determine what the smallest and largest keys are in the given data set. In this case, we can set up a node, called the header, at the beginning of the list containing a value smaller than the smallest value in the data set. Similarly, we can set up a node, called the trailer, at the end of the list containing a value larger than the largest value in the data set. These two nodes, header and trailer, serve merely to simplify the insertion and deletion algorithms and are not part of the actual list. The actual list is between these two nodes.

For example, suppose the data are ordered according to the last name. Further, assume that the last name is a string of at most 8 characters. The smallest last name is larger than the string "A" and the largest last name is smaller than the string "zzzzzzzz". We can set up the header node with the value "A" and the trailer node with the value "zzzzzzzz". Figure 3-33 shows an empty and a nonempty linked list with header and trailer nodes.

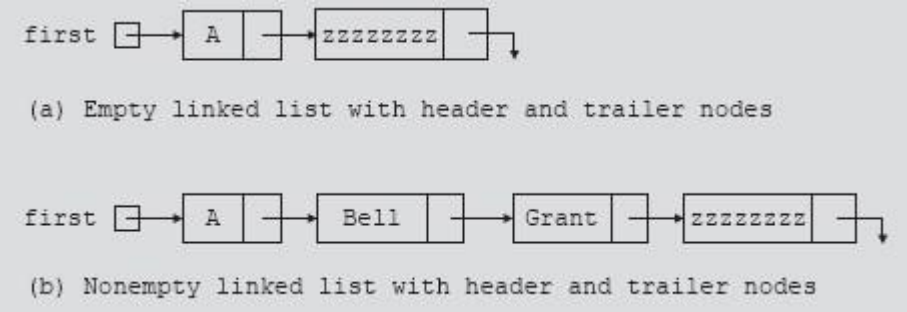


FIGURE 3-33 Linked list with header and trailer nodes

As before, the usual operations on lists with header and trailer nodes are as follows: Initialize the list (to an empty state), destroy the list, print the list, find the length of the list, search the list for a given item, insert an item in the list, delete an item from the list, and copy the list.

We leave it as an exercise for you to design a class to implement a linked list with header and trailer nodes.

3.6 Circular Linked Lists

A linked list in which the last node points to the first node is called a circular linked list. Figure 3-34 shows various circular linked lists.

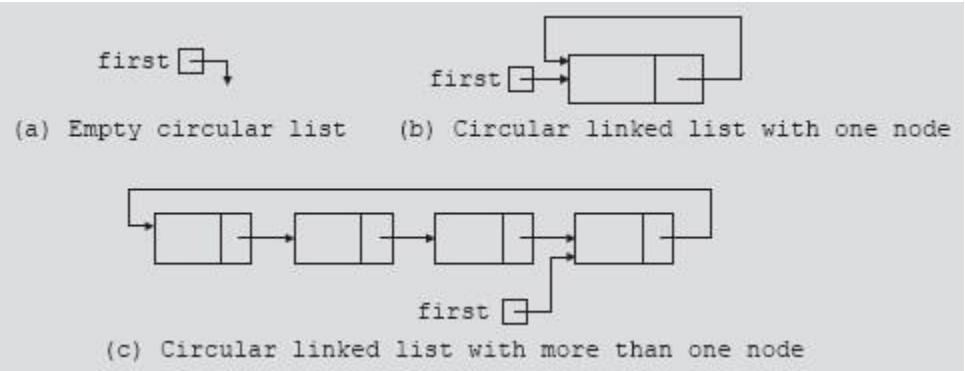


FIGURE 3-34 Circular linked lists

In a circular linked list with more than one node, as in Figure 3-34(c), it is convenient to make the pointer

first point to the last node of the list. Then by using first you can access both the first and the last node of the list. For example, first points to the last node and first->link points to the first node.

As before, the usual operations on a circular list are as follows: Initialize the list (to an empty state), determine if the list is empty, destroy the list, print the list, find the length of the list, search the list for a given item, insert an item in the list, delete an item from the list, and copy the list.

We leave it as exercise for you to design a class to implement a sorted circular linked list.

Key Terms

1. A linked list is a list of items, called nodes, in which the order of the nodes is determined by the address, called a link, stored in each node.
2. The pointer to a linked list—that is, the pointer to the first node in the list—is stored in a separate location, called the head or first.
3. A linked list is a dynamic data structure.
4. The length of a linked list is the number of nodes in the list.
5. Item insertion and deletion from a linked list does not require data movement; only the pointers are adjusted.
6. A (single) linked list is traversed in only one direction.
7. The search on a linked list is sequential.
8. The first (or head) pointer of a linked list is always fixed, pointing to the first node in the list.
9. To traverse a linked list, the program must use a pointer different than the head pointer of the list, initialized to the first node in the list.
10. In a doubly linked list, every node has two links: one points to the next node, and one points to the previous node.
11. A doubly linked list can be traversed in either direction.
12. In a doubly linked list, item insertion and deletion requires the adjustment of two pointers in a node.
13. The name of the class containing the definition of the class list is list.
14. In addition to the operations that are common to sequence containers (see Chapter 2), the other operations that can be used to manipulate the elements in a list container are assign, push_front, pop_front, front, back, remove, remove_if, unique, splice, sort, merge, and reverse.
15. A linked list with header and trailer nodes simplifies the insertion and deletion operations.
16. The header and trailer nodes are not part of the actual list. The actual list elements are between the header and trailer nodes.
17. A linked list with header and trailer nodes is empty if the only nodes in the list are the header and the trailer.
18. A circular linked list is a list in which, if the list is nonempty, the last node points to the first node.

Review Questions

1. Mark the following statements as true or false.

a. In a linked list, the order of the elements is determined by the order in which the nodes were created to store the elements.

b. In a linked list, memory allocated for the nodes is sequential.

c. A single linked list can be traversed in either direction.

d. In a linked list, nodes are always inserted either at the beginning or the end because a linked link is not a random access data structure.

e. The head pointer of a linked list cannot be used to traverse the list.

Consider the linked list shown in Figure 3-35. Assume that the nodes are in the usual info-link form. Use this list to answer Exercises 2 through 7. If necessary, declare additional variables. (Assume that list, p, s, A, and B are pointers of type nodeType.)

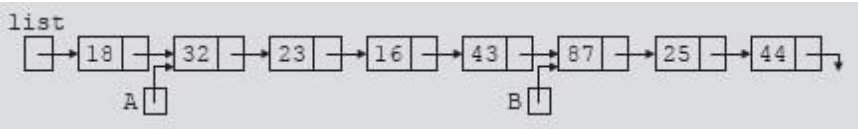


FIGURE 3-35 Linked list for Exercises 2–7

2. What is the output of each of the following C++ statements? a.

cout << list->info;

b. cout << A->info;

c. cout << B->link->info;

d. cout << list->link->link->info

3. What is the value of each of the following relational expressions? a.

list->info >= 18

b. list->link == A

c. A->link->info == 16

d. B->link == NULL

e. list->info == 18

4. Mark each of the following statements as valid or invalid. If a statement is invalid, explain why. a.

A = B;

b. list->link = A->link;

c. list->link->info = 45;

d. *list = B;

e. *A = *B;

f. B = A->link->info;

g. A->info = B->info;

h. list = B->link->link;

i. B = B->link->link->link;

5. Write C++ statements to do the following:

a. Make A point to the node containing info 23.

b. Make list point to the node containing 16.

c. Make B point to the last node in the list.

d. Make list point to an empty list.

e. Set the value of the node containing 25 to 35.

f. Create and insert the node with info 10 after the node pointed to by A.

g. Delete the node with info 23. Also, deallocate the memory occupied by this node.

6. What is the output of the following C++ code? p

= list;

```
while (p != NULL)
        cout << p->info << " "; p
= p->link;
cout << endl;
```

7. If the following C++ code is valid, show the output. If it is invalid, explain why. a. s = A;

```
p = B; s-
>info = B; p
= p->link;
cout << s->info << " " << p->info << endl;
```

b.
```
p = A;
p = p->link; s =
p; p->link =
NULL; s = s-
>link;
cout << p->info << " " << s->info << endl;
```

8. Show what is produced by the following C++ code. Assume the node is in the usual info-link form with the info of type int. (list and ptr are pointers of type nodeType.)

a.
```
list = new nodeType;
list->info = 10; ptr =
new nodeType; ptr-
>info = 13; ptr->link
= NULL; list->link =
ptr;
ptr = new nodeType;
ptr->info = 18; ptr-
>link = list->link; list-
>link = ptr;
cout << list->info << " " << ptr->info << " "; ptr
= ptr->link;
cout << ptr->info << endl;
```

b.
```
list = new nodeType;
list->info = 20; ptr =
new nodeType; ptr-
>info = 28; ptr->link
```

```
= NULL; list->link =
ptr;
ptr = new nodeType;
ptr->info = 30; ptr-
>link = list;
list = ptr;
ptr = new nodeType;
ptr->info = 42; ptr-
>link = list->link; list-
>link = ptr;
ptr = List;
while (ptr != NULL)
{
        cout << ptr->info << endl;
        ptr = ptr->link;
```

```
}
```

9. Consider the following C++ statements. (The class unorderedLinkedList is as defined in this chapter.)

```
unorderedLinkedList<int> list;
list.insertFirst(15);
list.insertLast(28);
list.insertFirst(30);
list.insertFirst(2);
list.insertLast(45);
list.insertFirst(38);
list.insertLast(25);
list.deleteNode(30);
list.insertFirst(18);
list.deleteNode(28);
list.deleteNode(12);
list.print();
```

What is the output of this program segment?

10. Suppose the input is:

18 30 4 32 45 36 78 19 48 75 -999

What is the output of the following C++ code? (The class unorderedLinkedList is as defined in this chapter.)

```
unorderedLinkedList<int> list;
unorderedLinkedList<int>
copyList; int num; cin >> num;
while (num != -999)
{
        if (num % 5 == 0 || num % 5 == 3)
        list.insertFirst(num); else
        list.insertLast(num);
        cin >> num;
} list.print();
cout <<
endl;
```

```
copyList = list;
copyList.deleteNode(78);
copyList.deleteNode(35);
cout << "Copy List = ";
copyList.print(); cout << endl;
```

11. Draw the UML diagram of the class doublyLinkedList as discussed in this chapter.

12. Suppose that intList is a list container and

intList = {3, 23, 23, 43, 56, 11, 11, 23, 25}

Show intList after the following statement executes: intList.unique();

13. Suppose that intList1 and intList2 are list containers and

intList1 = {3, 58, 78, 85, 6, 15, 93, 98, 25} intList2

= {5, 24, 16, 11, 60, 9}

Show intList1 after the following statement executes:

intList1.splice(intList1.begin(), intList2);

14. What is the output of the following program segment?

```
list<int> intList;
ostream_iterator<int> screen(cout, " ");
list<int>::iterator listIt;
intList.push_back(5);
intList.push_front(23);
intList.push_front(45);
intList.pop_back();
intList.push_back(35);
intList.push_front(0);
intList.push_back(50);
intList.push_front(34);
copy(intList.begin(), intList.end(), screen);
cout << endl; listIt = intList.begin();
intList.insert(listIt,76);
++listIt; ++listIt;
intList.insert(listIt,38)
; intList.pop_back();
++listIt; ++listIt; intList.erase(listIt);
intList.push_front(2 * intList.back());
intList.push_back(3 * intList.front());
copy(intList.begin(), intList.end(),
screen); cout << endl;
```

4 TOPIC 4: RECURSION



LEARNING OUTCOMES:

After studying this topic you should be able to:

- Learn about recursive definitions
- Explore the base case and the general case of a recursive definition
- Learn about recursive algorithm
- Learn about recursive functions
- Explore how to use recursive functions to implement recursive algorithms

In previous chapters, to devise solutions to problems we used the most common technique, called iteration. For certain problems, however, using the iterative technique to obtain the solution is quite complicated. This chapter introduces another problemsolving technique, called recursion, and provides several examples demonstrating how recursion works.

4.1 Recursive Definitions

The process of solving a problem by reducing it to smaller versions of itself is called recursion. Recursion is a very powerful way to solve certain problems for which the solution would otherwise be very complicated. Let us consider a problem that is familiar to most everyone.

In an algebra course, you probably learned how to find the factorial of a nonnegative integer. For example, the factorial of 5, written 5!, is 5 x 4 x 3 x 2 x 1 = 120. Similarly, 4! = 4 x 3 x 2 x 1 = 24. Also, factorial of 0 is defined to be 0! = 1. Note that 5! = 5 x 4 x 3 x 2 x 1 = 5 x (4 x 3 x 2 x 1) = 5 x 4!. In general, if n is a nonnegative, the factorial of n, written as n! can be defined as follows:

0! = 1 (Equation 4-1) n! = n x (n - 1)! if n > 0 (Equation 4-2)

In this definition, 0! is defined to be 1, and if n is an integer greater than 0, first we find (n - 1)! and then multiply it by n. To find (n - 1)!, we apply the definition again. If (n - 1) > 0, then we use Equation 4-2; otherwise, we use Equation 4-1. Thus, for an integer n greater than 0, n! is obtained by first finding (n - 1)! and then multiplying (n - 1)! by n.

Let us apply this definition to find 3!. Here n = 3. Because n > 0, we use Equation 4-2 to obtain:

$$3! = 3 \times 2!$$

Next, we find 2! Here n = 2. Because n > 0, we use Equation 4-2 to obtain:

$$2! = 2 \times 1!$$

Now to find 1!, we again use Equation 4-2 because n = 1 > 0. Thus:

$$1! = 1 \times 0!$$

Finally, we use Equation 4-1 to find 0!, which is 1. Substituting 0! into 1! gives 1! = 1. This gives 2! = 2 x 1! = 2 x 1 = 2, which in turn gives 3! = 3 x 2! = 3 x 2 = 6.

The solution in Equation 4-1 is direct—that is, the right side of the equation contains no factorial notation. The solution in Equation 4-2 is given in terms of a smaller version of itself. The definition of the factorial given in Equations 4-1 and 4-2 is called a recursive definition. Equation 4-1 is called the base case (that is, the case for which the solution is obtained directly); Equation 4-2 is called the general case. Recursive definition: A definition in which something is defined in terms of a smaller version of itself.

From the previous example (factorial), it is clear that:

1. Every recursive definition must have one (or more) base cases.
2. The general case must eventually be reduced to a base case.
3. The base case stops the recursion.

The concept of recursion in computer science works similarly. Here, we talk about recursive algorithms and recursive functions. An algorithm that finds the solution to a given problem by reducing the problem to smaller versions of itself is called a recursive algorithm. The recursive algorithm must have one or more base cases, and the general solution must eventually be reduced to a base case.

A function that calls itself is called a recursive function. That is, the body of the recursive function contains a statement that causes the same function to execute again before completing the current call. Recursive algorithms are implemented using recursive functions.

Next, let us write the recursive function that implements the factorial function.

```
int fact(int num)
{ if (num == 0) return 1; else return num *
        fact(num - 1);
}
```

Figure 4-1 traces the execution of the following statement:

```
cout << fact(3) << endl;
```

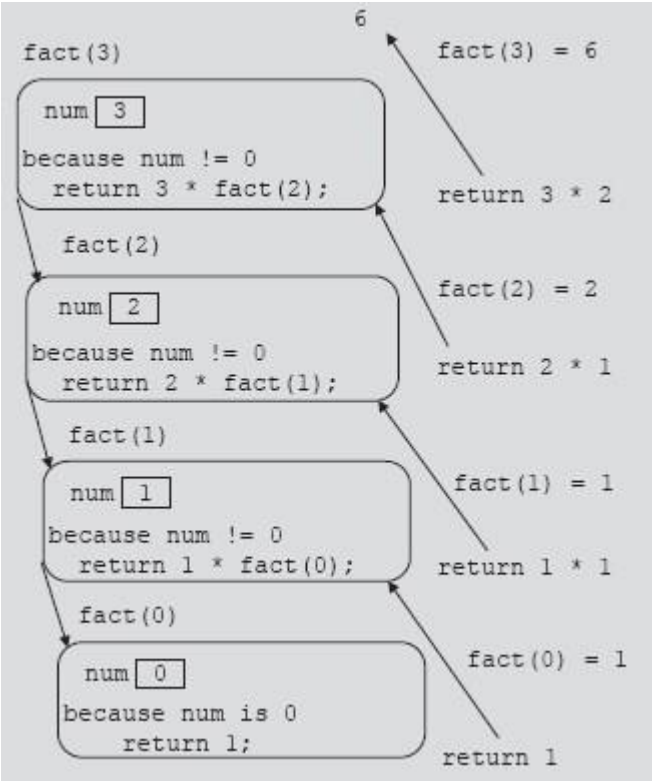


FIGURE 4-1 Execution of fact(4)

The output of the previous cout statement is: 6

In Figure 4-1, the down arrow represents the successive calls to the function fact, and the upward arrows represent the values returned to the caller, that is, the calling function.

Let us note the following from the previous example, involving the factorial function:

- Logically, you can think of a recursive function as having an unlimited number of copies of itself. • Every call to a recursive function—that is, every recursive call—has its own code and its own set of parameters and local variables.
- After completing a particular recursive call, control goes back to the calling environment, which is the previous call. The current (recursive) call must execute completely before control goes back to the previous call. The execution in the previous call begins from the point immediately following the recursive call.

4.1.1 Direct and Indirect Recursion

A function is called directly recursive if it calls itself. A function that calls another function and eventually results in the original function call is said to be indirectly recursive. For example, if a function A calls a

function B and function B calls function A, then function A is indirectly recursive. Indirect recursion can be several layers deep. For example, suppose that function A calls function B, function B calls function C, function C calls function D, and function D calls function A. Function A is then indirectly recursive.

Indirect recursion requires the same careful analysis as direct recursion. The base cases must be identified and appropriate solutions to them must be provided. However, tracing through indirect recursion can be tedious. You must, therefore, exercise extra care when designing indirect recursive functions. For simplicity, the problems in this book involve only direct recursion.

A recursive function in which the last statement executed is the recursive call is called a tail recursive function. The function fact is an example of a tail recursive function.

4.1.2 Infinite Recursion

Figure 4-1 shows that the sequence of recursive calls eventually reached a call that made no further recursive calls. That is, the sequence of recursive calls eventually reached a base case. On the other hand, if every recursive call results in another recursive call, the recursive function (algorithm) is said to have infinite recursion. In theory, infinite recursion executes forever. Every call to a recursive function requires the system to allocate memory for the local variables and formal parameters. The system also saves this information so that after completing a call, control can be transferred back to the right caller. Therefore, because computer memory is finite, if you execute an infinite recursive function on a computer, the function executes until the system runs out of memory and results in an abnormal termination of the program.

Recursive functions (algorithms) must be carefully designed and analyzed. You must make sure that every recursive call eventually reduces to a base case. This chapter provides several examples that illustrate how to design and implement recursive algorithms.

To design a recursive function, you must do the following:

1. Understand the problem requirements.
2. Determine the limiting conditions. For example, for a list, the limiting condition is the number of elements in the list.
3. Identify the base cases and provide a direct solution to each base case.
4. Identify the general cases and provide a solution to each general case in terms of smaller versions of itself.

4.2 Problem Solving Using Recursion

The next few sections illustrate how recursive algorithms are developed and implemented in C++ using recursive functions.

4.1.3 Largest Element in an Array

In this example, we use a recursive algorithm to find the largest element in an array. Consider the list given in Figure 4-2.

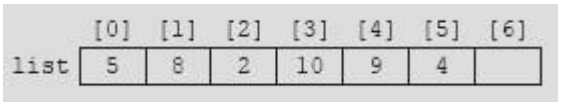


FIGURE 4-2 list with six elements

The largest element in the list in Figure 4-2 is 10.

Suppose list is the name of the array containing the list elements. Also, suppose that list[a]...list[b] stands for the array elements list[a], list[a + 1], ..., list[b]. For example, list[0]...list[5] represents the array elements list[0], list[1], list[2], list[3], list[4], and list[5]. Similarly, list[1]...list[5] represents the array

elements list[1], list[2], list[3], list[4], and list[5]. To write a recursive algorithm to find the largest element in list, let us think in terms of recursion.

If list is of length 1, then list has only one element, which is the largest element. Suppose the length of list is greater than 1. To find the largest element in list[a]...list[b], we first find the largest element in list[a + 1]...list[b] and then compare this largest element with list[a]. That is, the largest element in list[a]...list[b] is given by:

maximum(list[a], largest(list[a + 1]...list[b]))

Let us apply this formula to find the largest element in the list shown in Figure 4-2. This list has six elements, given by list[0]...list[5]. Now the largest element in list is given by:
maximum(list[0], largest(list[1]...list[5]))

That is, the largest element in list is the maximum of list[0] and the largest element in list[1]...list[5]. To find the largest element in list[1]...list[5], we use the same formula again because the length of this list is greater than 1. The largest element in list[1]...list[5] is then:
maximum(list[1], largest(list[2]...list[5]))

and so on. We see that every time we use the previous formula to find the largest element in a sublist, the length of the sublist in the next call is reduced by one. Eventually, the sublist is of length 1, in which case the sublist contains only one element, which is the largest element in the sublist. From this point onward, we backtrack through the recursive calls. This discussion translates into the following recursive algorithm, which is presented in pseudocode:

Base Case: The size of the list is 1
The only element in the list is the largest element

General Case: The size of the list is greater than 1
To find the largest element in list[a]...list[b]
1. Find the largest element in list[a + 1]...list[b] and call it max
2. Compare the elements list[a] and max if (list[a] >= max)
the largest element in list[a]...list[b] is list[a]

otherwise the largest element in list[a]...list[b] is

max

This algorithm translates into the following C++ function to find the largest element in an array:

```
int largest(const int list[], int lowerIndex, int upperIndex)
{
        int max;
        if (lowerIndex == upperIndex) //size of the sublist is
        one return list[lowerIndex]; else
        {
                max = largest(list, lowerIndex + 1, upperIndex);
                if (list[lowerIndex] >= max) return
                list[lowerIndex]; else
                        return max;
```

```
    }
}
```

Consider the list given in Figure 4-3.

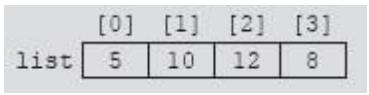


FIGURE 4-3 list with four elements

Let us trace the execution of the following statement:

```
cout << largest(list, 0, 3) << endl;
```

Here upperIndex = 3 and the list has four elements. Figure 4-4 traces the execution of largest(list, 0, 3).

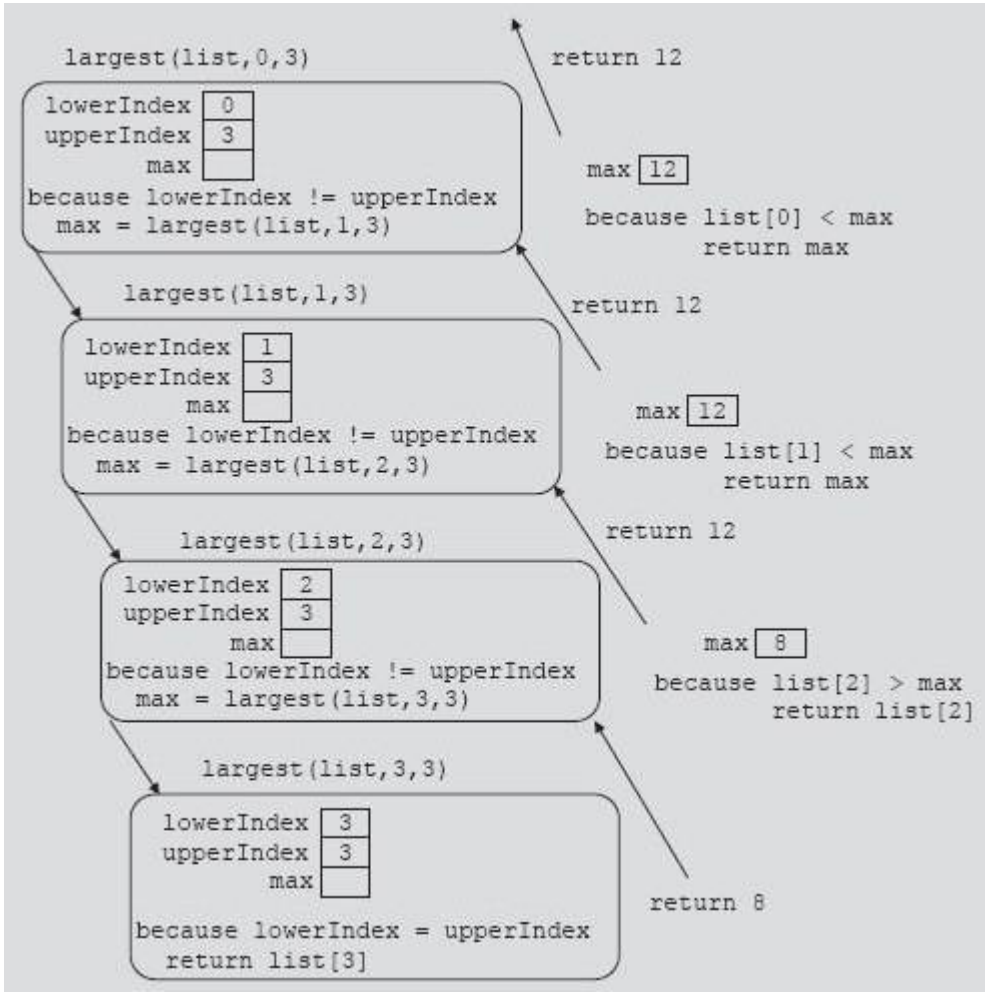


FIGURE 4-4 Execution of largest(list, 0, 3)

The value returned by the expression largest(list, 0, 3) is 12, which is the largest element in list.

The followingC++program uses the function largest to determine the largest element in a list:

```
//*************************************************************
//This program uses a recursive function to find the largest element in a list.
//*************************************************************
```

```
#include <iostream> using
namespace std;
int largest(const int list[], int lowerIndex, int upperIndex); int
main()
{ int intArray[10] = {23, 43, 35, 38, 67, 12, 76, 10, 34, 8};
        cout << "The largest element in intArray: "<< largest(intArray, 0, 9); cout
        << endl;
        return 0;
}

int largest(const int list[], int lowerIndex, int upperIndex)
{
        int max;
        if (lowerIndex == upperIndex) //size of the sublist is
        one return list[lowerIndex]; else
        {
                max = largest(list, lowerIndex + 1, upperIndex);
                if (list[lowerIndex] >= max) return
                list[lowerIndex]; else return max;
        }
}
```

Sample Run:

The largest element in intArray: 76

4.1.4 Print a Linked List in Reverse Order

The nodes of an ordered linked list (as constructed in Chapter 3) are in ascending order. Certain applications, however,might require the data to be printed in descending order, which means that we must print the list backward. We now discuss the function reversePrint. Given a pointer to a list, this function prints the elements of the list in reverse order.

Consider the linked list shown in Figure 4-5.

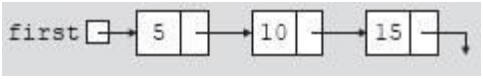


FIGURE 4-5 Linked list

For the list in Figure 4-5, the output should be in the following form:

15 10 5

Because the links are in only one direction, we cannot traverse the list backward starting from the last node. Let us see how we can effectively use recursion to print the list in reverse order.

Let us think in terms of recursion. We cannot print the info of the first node until we have printed the remainder of the list (that is, the tail of the first node). Similarly, we cannot print the info of the second node until we have printed the tail of the second node, and so on. Every time we consider the tail of a

node, we reduce the size of the list by 1. Eventually, the size of the list is reduced to zero, in which case the recursion stops.

Base Case: List is empty: no action

General Case: List is nonempty

1. Print the tail
2. Print the element

Let us write this algorithm. (Suppose that current is a pointer to a linked list.)

```
if (current != NULL)
{ reversePrint(current->link); //print the tail cout <<
        current->info << endl; //print the node
}
```

Here, we do not see the base case; it is hidden. The list is printed only if the pointer, current, to the list is not NULL. Also, inside the if statement the recursive call is on the tail of the list. Because eventually the tail of a list will be empty, the if statement in the next call fails and the recursion stops. Also, note that statements (for example, printing the info of the node) appear after the recursive call; thus, when the transfer comes back to the calling function, we must execute the remaining statements. Recall that the function exits only after the last statement executes. (By the ‘‘last statement,’’ we do not mean the physical last statement, but rather the logical last statement.)

Let us write a function template to implement the previous algorithm and then apply it to a list.

```
template <class Type> void
linkedListType<Type>::reversePrint
(nodeType<Type> *current) const
{
        if (current != NULL)
        { reversePrint(current->link); //print the tail cout <<
                current->info << " "; //print the node
        }
}
```

Consider the statement

```
reversePrint(first);
```

where first is a pointer of type nodeType<Type>.

Let us trace the execution of this statement; which is a function call, for the list shown in Figure 4-5. Because the formal parameter is a value parameter, the value of the actual parameter is passed to the formal parameter. See Figure 4-6.

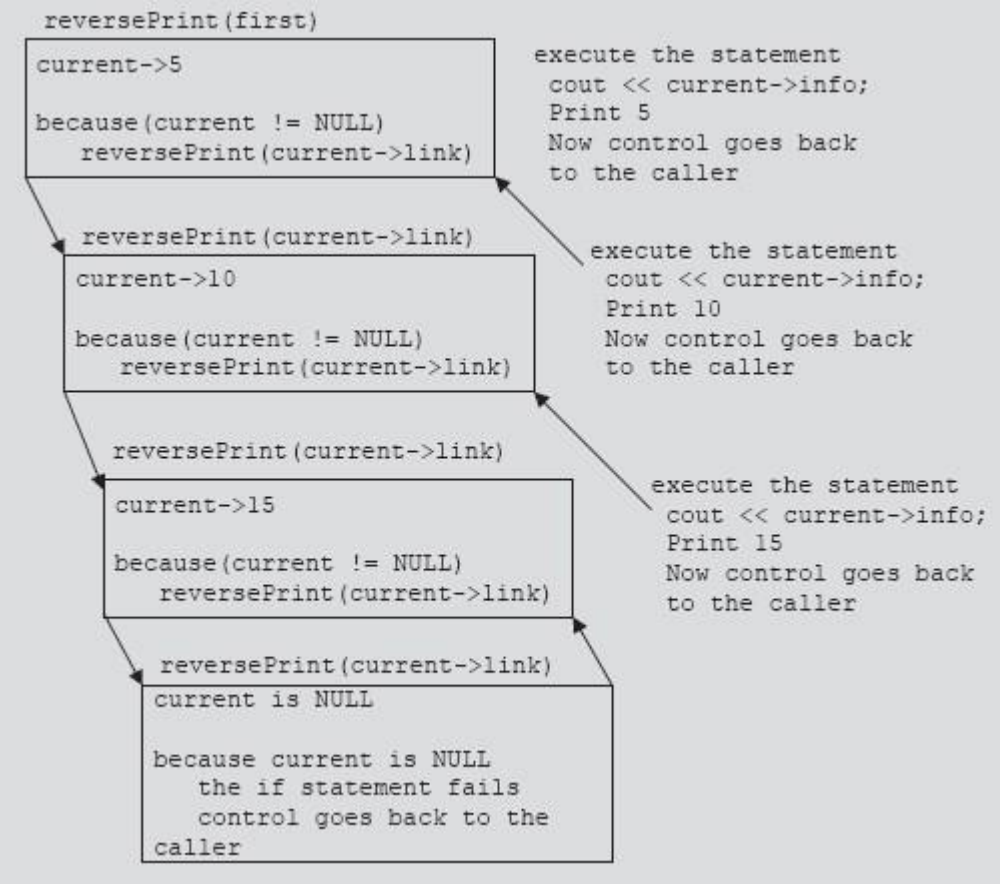


FIGURE 4-6 Execution of the statement reversePrint(first);

4.1.4.1 The Function printListReverse

Now that we have written the function reversePrint, we can write the definition of the function printListReverse, which can be used to print an ordered linked list contained in an object of the type linkedListType. Its definition is as follows:

```
template <class Type>
void linkedListType<Type>::printListReverse() const
{ reversePrint(first);
        cout << endl;
}
```

We can include the function printListReverse as a public member in the definition of the class and the function reversePrint as a private member. We include the function reversePrint as a private member because it is used only to implement the function printListReverse.

4.1.5 Fibonacci Number

Consider the following sequence of numbers:

1; 1; 2; 3; 5; 8; 13; 21; 34; . . .

Given the first two numbers of the sequence (say a_1 and a_2), the nth number a_n, n >= 3, of this sequence is given by:

$a_n = a_{n-1} + a_{n-2}$

Thus:

$a_3 = a_2 + a_1 = 1 + 1 = 2$, $a_4 = a_3 + a_2 = 2 + 1 = 3$, and so on.

Such a sequence is called a Fibonacci sequence. In the previous sequence, $a_2 = 1$ and $a_1 = 1$. However, given any first two numbers, using this process, you can determine the nth number, a_n, n >= 3, of such a sequence. The number determined this way is called the nth Fibonacci number. Suppose $a_2 = 6$ and $a_1 = 3$.

Then:

$a_3 = a_2 + a_1 = 6 + 3 = 9$; $a_4 = a_3 + a_2 = 9 + 6 = 15$:

In this example, we write a recursive function, **rFibNum**, to determine the desired Fibonacci number. The function **rFibNum** takes as parameters three numbers representing the first two numbers of the Fibonacci sequence and a number n, the desired nth Fibonacci number. The function **rFibNum** returns the nth Fibonacci number in the sequence.

Recall that the third Fibonacci number is the sum of the first two Fibonacci numbers. The fourth Fibonacci number in a sequence is the sum of the second and third Fibonacci numbers. Therefore, to calculate the fourth Fibonacci number, we add the second Fibonacci number and the third Fibonacci number (which is itself the sum of the first two Fibonacci numbers). The following recursive algorithm calculates the nth Fibonacci number, where *a* denotes the first Fibonacci number, *b* the second Fibonacci number, and n the

$$rFibNum(a, b, n) = \begin{cases} a & \text{if } n = 1 \\ b & \text{if } n = 2 \\ rFibNum(a, b, n-1) + \\ rFibNum(a, b, n-2) & \text{if } n > 2. \end{cases}$$

(Equation 4-3)

The following recursive function implements this algorithm:

```
int rFibNum(int a, int b, int n)
{ if (n == 1) return a; else if (n == 2) return b; else return
```

```
        rFibNum(a, b, n - 1) + rFibNum(a, b, n - 2);
}
```

Let us trace the execution of the following statement:

```
        cout << rFibNum(2, 3, 4) << endl;
```

In this statement, the first number is 2, the second number is 3, and we want to determine the 4th Fibonacci number of the sequence. Figure 4-7 traces the execution of the expression **rFibNum(2,3,4)**. The value returned is 8, which is the 4th Fibonacci number of the sequence whose first number is 2 and whose second number is 3.

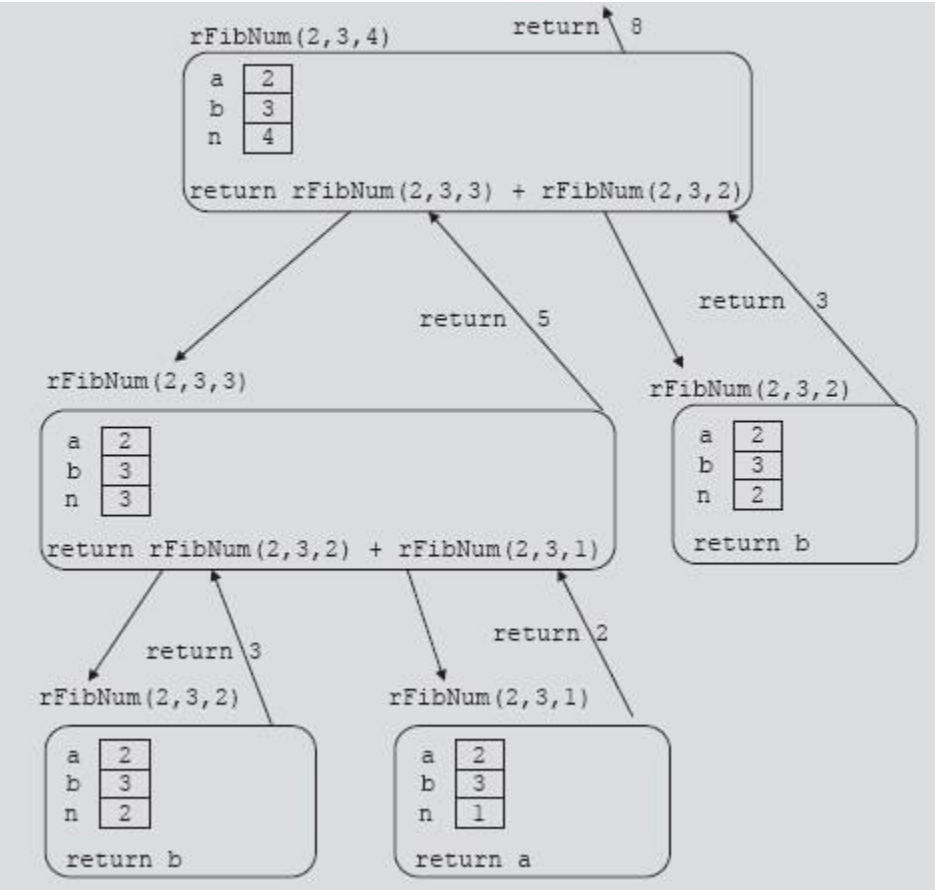


FIGURE 4-7 Execution of rFibNum(2, 3, 4)

Figure 4-7 reveals that the execution of the recursive version of the program to calculate a Fibonacci number is not as efficient as the execution of the nonrecursive version, even though the algorithm and the method implementing the algorithm might be simpler. In the recursive version, some values are calculated more than once. For example, to calculate **rFibNum(2, 3, 4)**, the value **rFibNum(2, 3, 2)** is calculated twice. So a recursive method might be easier to write, but might not execute as efficiently. The section ‘‘Recursion or Iteration?’’, presented later in this chapter, discusses these two alternatives.

The following C++ program uses the function **rFibNum**:

```
//************************************************************
// Given the first two numbers of a Fibonacci sequence, this //
program uses a recursive function to determine a specific //
number(s) of a Fibonacci sequence.
//************************************************************
#include <iostream> using
namespace std; int
rFibNum(int a, int b, int n);
```

```
int main()
{
        int firstFibNum;
        int secondFibNum; int
        nth;
        cout << "Enter the first Fibonacci number:
        "; cin >> firstFibNum; cout << endl;
```

```
        cout << "Enter the second Fibonacci number: "; cin
        >> secondFibNum;
        cout << endl;
        cout << "Enter the position of the desired Fibonacci number:
        "; cin >> nth; cout << endl;
        cout << "The Fibonacci number at position " << nth << " is: " << rFibNum(firstFibNum,
        secondFibNum, nth) << endl;
        return 0;
}

int rFibNum(int a, int b, int n)
{ if (n == 1) return a; else if (n == 2) return b; else return
        rFibNum(a, b, n - 1) + rFibNum(a, b, n - 2);
}
```

Sample Run: In this sample run, the user input is shaded.

Enter the first Fibonacci number: 2

Enter the second Fibonacci number: 5

Enter the position of the desired Fibonacci number: 6

The Fibonacci number at position 6 is: 31

4.1.6 Tower of Hanoi

In the nineteenth century, a game called the Tower of Hanoi became popular in Europe. This game represents work that is under way in the temple of Brahma. At the creation of the universe, priests in the temple of Brahma were supposedly given three diamond needles, with one needle containing 64 golden disks. Each golden disk is slightly smaller than the disk below it. The priests' task is to move all 64 disks from the first needle to the third needle. The rules for moving the disks are as follows:

1. Only one disk can be moved at a time.
2. The removed disk must be placed on one of the needles.
3. A larger disk cannot be placed on top of a smaller disk.

The priests were told that once they had moved all the disks from the first needle to the third needle, the

universe would come to an end.

Our objective is to write a program that prints the sequence of moves needed to transfer the disks from the first needle to the third needle. Figure 4-8 shows the Tower of Hanoi problem with three disks.

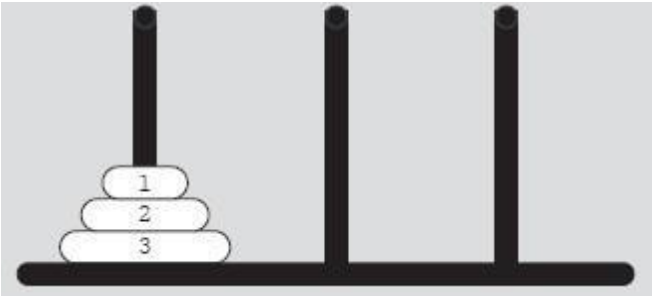


FIGURE 4-8 Tower of Hanoi problem with three disks

As before, we think in terms of recursion. Let us first consider the case when the first needle contains only one disk. In this case, the disk can be moved directly from needle 1 to needle 3. So let us consider the case when the first needle contains only two disks. In this case, first we move the first disk from needle 1 to needle 2, and then we move the second disk from needle 1 to needle 3. Finally, we move the first disk from needle 2 to needle 3. Next, we consider the case when the first needle contains three disks, and then generalize this to the case of 64 disks (in fact, to an arbitrary number of disks).

Suppose that needle 1 contains three disks. To move disk number 3 to needle 3, the top two disks must first be moved to needle 2. Disk number 3 can then be moved from needle 1 to needle 3. To move the top two disks from needle 2 to needle 3, we use the same strategy as before. This time we use needle 1 as the intermediate needle. Figure 4-9 shows a solution to the Tower of Hanoi problem with three disks.

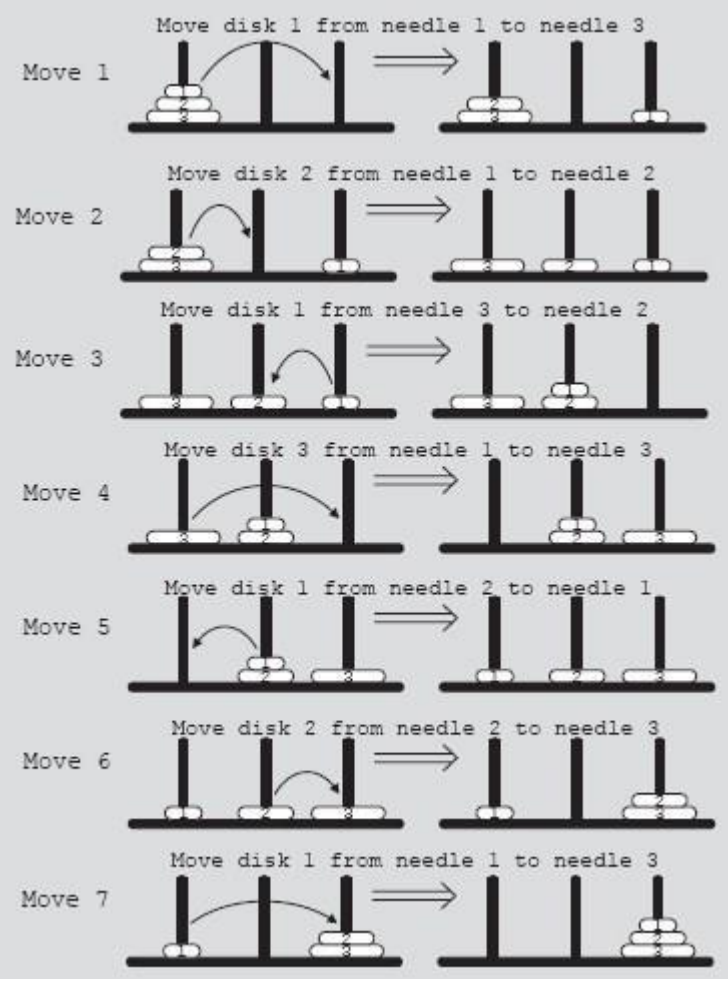


FIGURE 4-9 Solution to Tower of Hanoi problem with three disks

Let us now generalize this problem to the case of 64 disks. To begin, the first needle contains all 64 disks. Disk number 64 cannot be moved from needle 1 to needle 3 unless the top 63 disks are on the second needle. So first we move the top 63 disks from needle 1 to needle 2, and then we move disk number 64

from needle 1 to needle 3. Now the top 63 disks are all on needle 2. To move disk number 63 from needle 2 to needle 3, we first move the top 62 disks from needle 2 to needle 1, and then we move disk number 63 from needle 2 to needle 3. To move the remaining 62 disks, we use a similar procedure. This discussion translates into the following recursive algorithm given in pseudocode.

Suppose that needle 1 contains n disks, where n ≥ 1.

1. Move the top n - 1 disks from needle 1 to needle 2, using needle 3 as the intermediate needle.

2. Move disk number n from needle 1 to needle 3.
3. Move the top n - 1 disks from needle 2 to needle 3, using needle 1 as the intermediate needle.

This recursive algorithm translates into the following C++ function:

```
void moveDisks(int count, int needle1, int needle3, int needle2)
{
        if (count > 0)
        {
                moveDisks(count - 1, needle1, needle2, needle3);
                cout << "Move disk " << count << " from " << needle1<< " to " << needle3 << "." << endl;
                moveDisks(count - 1, needle2, needle3, needle1);
        }
}
```

4.1.6.1 Tower Of Hanoi: Analysis

Let us determine how long it would take to move all 64 disks from needle 1 to needle 3. If needle 1 contains 3 disks, then the number of moves required to move all 3 disks from needle 1 to needle 3 is 2^3 1 = 7. Similarly, if needle 1 contains 64 disks, then the number of moves required to move all 64 disks from needle 1 to needle 3 is 2^{64} - 1. Because $2^{10} = 1024 \approx 1000 = 10^3$, we have

$2^{64} = 2^4 \times 2^{60} \approx 2^4 \times 10^{18} = 1.6 \times 10^{19}$

The number of seconds in one year is approximately 3.2×10^7. Suppose the priests move one disk per second and they do not rest. Now:

$1.6 \times 10^{19} = 5 \times 3.2 \times 10^{18} = 5 \times (3.2 \times 10^7) \times 10^{11} = (3.2 \times 10^7) \times (5 \times 10^{11})$

The time required to move all 64 disks from needle 1 to needle 3 is roughly 5×10^{11} years. It is estimated that our universe is about 15 billion years old (1.5×10^{10}). Also, $5 \times 10^{11} = 50 \times 10^{10} \approx 33 \times (1.5 \times 10^{10})$. This calculation shows that our universe would last about 33 times as long as it already has.

Assume that a computer can generate 1 billion (10^9) moves per second. Then the number of moves that the computer can generate in one year is:

$(3.2 \times 10^7) \times 10^9 = 3.2 \times 10^{16}$

So the computer time required to generate 2^{64} moves is: 2^{64}

$\approx 1.6 \times 10^{19} = 1.6 \times 10^{16} \times 10^3 = (3.2 \times 10^{16}) \times 500$

Thus, it would take about 500 years for the computer to generate 2^{64} moves at the rate of 1 billion moves per second.

4.1.7 Converting a Number from Decimal to Binary

In this example, we design a program that uses recursion to convert a nonnegative integer in decimal format—that is, base 10—into the equivalent binary number—that is, base 2. First we define some terms.

Let x be an integer. We call the remainder of x after division by 2 the rightmost bit of x. Thus, the rightmost bit of 33 is 1 because 33 % 2 is 1, and the rightmost bit of 28 is 0 because 28 % 2 is 0. (Recall that in C++, % is the mod operator; it produces the remainder of the integer division.)

We first illustrate the algorithm to convert an integer in base 10 to the equivalent number in binary format with the help of an example.

Suppose we want to find the binary representation of 35. First, we divide 35 by 2. The quotient is 17 and the remainder—that is, the rightmost bit of 35—is 1. Next, we divide 17 by 2. The quotient is 8 and the remainder—that is, the rightmost bit of 17—is 1. Next, we divide 8 by 2. The quotient is 4 and the remainder—that is, the rightmost bit of 8—is 0. We continue this process until the quotient becomes 0.

The rightmost bit of 35 cannot be printed until we have printed the rightmost bit of 17. The rightmost bit of 17 cannot be printed until we have printed the rightmost bit of 8, and so on. Thus, the binary representation of 35 is the binary representation of 17 (that is, the quotient of 35 after division by 2), followed by the rightmost bit of 35.

Thus, to convert an integer num in base 10 into the equivalent binary number, we first convert the quotient num / 2 into an equivalent binary number, and then append the rightmost bit of num to the binary representation of num / 2.

This discussion translates into the following recursive algorithm, where binary(num) denotes the binary representation of num:

1. binary(num) = num if num = 0.
2. binary(num) = binary(num / 2) followed by num % 2 if num > 0.

The following recursive function implements this algorithm:

```
void decToBin(int num, int base)
{
        if (num > 0)
```

```
        {
                decToBin(num / base, base);
                cout << num % base;
        }
}
```

Figure 4-10 traces the execution of the following statement:

decToBin(13, 2);

where num is 13 and base is 2.

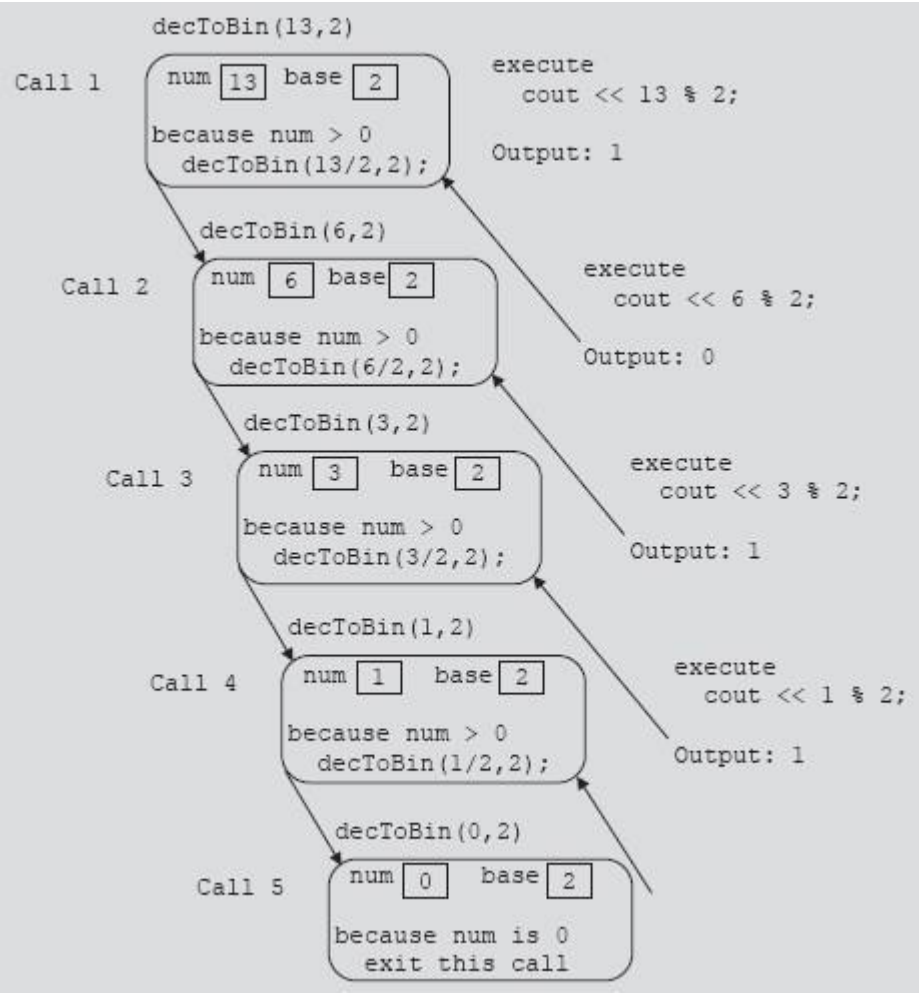


FIGURE 4-10 Execution of decToBin(13, 2)

Because the if statement in call 5 fails, this call does not print anything. The first output is produced by call 4, which prints 1; the second output is produced by call 3, which prints 1; the third output is produced by call 2, which prints 0; and the fourth output is produced by call 1, which prints 1. Thus, the output of the statement:

decToBin(13, 2);

is:

1101

The following C++ program tests the function decToBin:

//*** //

Program: Decimal to binary

// This program uses recursion to find the binary //

representation of a nonnegative integer.

```
//*********************************************************
#include <iostream> using
namespace std; void
decToBin(int num, int base); int
main()
{
```

```
        int decimalNum;
        int base; base =
        2;
        cout << "Enter number in decimal: ";
        cin >> decimalNum; cout
        << endl;
        cout << "Decimal " << decimalNum << " =
        "; decToBin(decimalNum, base); cout << "
        binary" << endl; return 0;
}

void decToBin(int num, int base)
{
        if (num > 0)
        {
                decToBin(num / base, base);
                cout << num % base;
        }
}
```

Sample Run: In this sample run, the user input is shaded.

Enter a number in decimal: 57

Decimal 57 = 111001 binary

4.3 Recursion or Iteration?

Often there are two ways to solve a particular problem—recursion or iteration. The programs in the previous chapters used a loop to repeat a set of statements to perform certain calculations. In other words, the programs in the previous chapters used an iterative control structure to repeat a set of statements. Formally, iterative control structures use a looping structure, such as while, for, or do... while, to repeat a set of statements.

This chapter began by designing a recursive method to find the factorial of a nonnegative integer. Using an iterative control structure, we can easily write an algorithm to find the factorial of a nonnegative

integer. Given our familiarity with iterative techniques, the iterative solution will seem simpler than the recursive solution. The only reason we gave a recursive solution to the factorial problem was to illustrate how recursion works using a simple example.

In this chapter, we also used recursion to determine the largest element in a list, determining a Fibonacci number. Using an iterative control structure, we can also write an algorithm to find the largest number in an array. Similarly, an algorithm that uses an iterative control structure can be designed to find the Fibonacci number.
The obvious question becomes, which approach is better? There is no general answer, but there are some guidelines. In addition to the nature of the solution, efficiency is the other key factor in determining the better approach.

When a function is called, memory space is allocated for its formal parameters and local variables. When the function terminates, that memory space is deallocated. In this chapter, while tracing the execution of recursive methods, we saw that every recursive call had its own set of parameters and local variables. That is, every recursive call required the system to allocate memory space for its formal parameters and local variables, and then deallocate the memory space when the function exited. Even though we don't need to write program statements to allocate and deallocate memory, overhead is associated with executing a recursive function, both in terms of memory space and execution time. Therefore, a recursive function executes more slowly than its iterative counterpart. On slower computers, especially those with limited memory space, the (relatively slow) execution of a recursive function is noticeable. Clearly, a recursive function is less efficient than a corresponding iterative function in terms of execution time and memory usage.

Efficiency is not determined solely by execution time and memory usage. Chances are that you have never been concerned with either execution time or memory usage as you have been writing C++ programs. Efficient use of a programmer's time also is an important consideration. Chances are that you have considered carefully what you can do to minimize the time required to produce C++ programs. Often this is entirely appropriate. As a professional programmer, your time typically is far more expensive than the cost of the computer you use to produce programs. Of course, if you are developing software that will be used many times a day by large numbers of users, then execution time and memory usage become important considerations.

Today's computers are fast and have abundant memory. Therefore, the additional execution time and the

memory consumed by a recursive function might not be noticeable. Given the ever-increasing speed and memory capacity of today's computers, the choice between iteration and recursion depends more and more on how the programmer envisions the solution to the problem, and less on execution time and memory usage. In rare instances where execution time must be minimized or the demand for memory is inordinately high, iteration might be better than recursion even when the recursive solution is more obvious. Fortunately, any program that can be written recursively also can be written iteratively.

As a general rule, if you think an iterative solution is at least as obvious and easy to construct as a recursive solution, choose the iterative solution. On the other hand, if the recursive solution is more obvious and easier to construct, such as the solution to the Towers of Hanoi problem, choose the recursive solution.

If you question the existence of problems for which a recursive solution is more obvious and easier to construct than an iterative solution, attempt to solve the Towers of Hanoi problem without using recursion. You'll quickly gain an increased appreciation for recursion. The ability to write recursive solutions is an important programming skill.

4.3 Recursion and Backtracking: 8-Queens Puzzle

This section describes a problem-solving and algorithm design technique called backtracking. Let us consider the following 8-queens puzzle: Place 8 queens on a chess-board (8 by 8 square board) so that no two queens can attack each other. For any two queens to be nonattacking, they cannot be in the same row, same column, or same diagonals. Figure 4-11 gives one possible solution to the 8-queens puzzle.

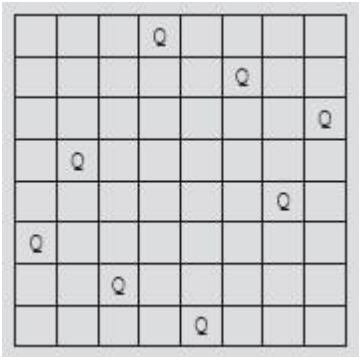


FIGURE 4-11 A solution to the 8-queens puzzle

In 1850, the 8-queens puzzle was considered by the great C. F. Gauss, who was unable to obtain a complete solution. The term backtrack was first coined by D. H. Lehmer in 1950. In 1960, R. J. Walker gave an algorithmic account of backtracking. A general description of backtracking with a variety of applications was presented by S. Golomb and L. Baumert.

4.3.1 Backtracking

The backtracking algorithm attempts to find solutions to a problem by constructing partial solutions and making sure that any partial solution does not violate the problem requirements. The algorithm tries to extend a partial solution toward completion. However, if it is determined that the partial solution would not lead to a solution, that is, the partial solution would end in a dead end, then the algorithm backs up by removing the most recently added part and trying other possibilities.

4.3.2 n-Queens Puzzle

In backtracking, the solution of the n-queens puzzle, because each queen must be placed in a different row, can be represented as an n-tuple $(x_1, x_2, \ldots, x_n)$, where x_i is an integer such that $1 \le x_i \le n$. In this tuple, x_i specifies the column number, where to place the ith queen in the ith row. Therefore, for the 8queens puzzle the solution is an 8-tuple $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$, where x_i is the column where to place the ith queen in the ith row. For example, the solution in Figure 4-11 can be represented as the 8-tuple (4,6,8,2,7,1,3,5). That is, the first queen is placed in the first row and fourth column, the second queen is placed in the second row and sixth column, and so on. Clearly, each x_i is an integer such that $1 \le x_i \le 8$.

Let us again consider the 8-tuple $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$, where x_i is an integer such that $1 \le x_i \le 8$. Because each x_i has 8 choices, there are 8^8 such tuples, and so possibly 8^8 solutions. However, because no

two queens can be placed in the same row, no two elements of the 8-tuple (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) are the same. From this, it follows that the number of 8-tuples possibly representing solutions are 8!.

The solution that we develop can, in fact, be applied to any number of queens. Therefore, to illustrate the backtracking technique, we consider the 4-queens puzzle. That is, you are given a 4 by 4 square board (see Figure 4-12) and you are to place 4 queens on the board so that no two queens attack each other.

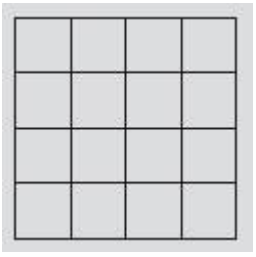


FIGURE 4-12 Square board for the 4-queens puzzle

We start by placing the first queen in the first row and first column as shown in Figure 4-13(a). (A cross in a box means no other queen can be placed in that box.)

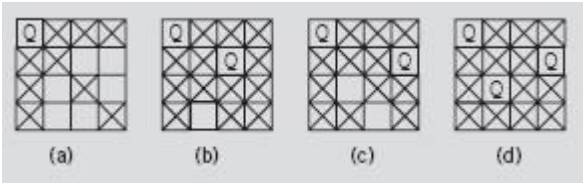


FIGURE 4-13 Finding a solution to the 4-queens puzzle

After placing the first queen, we try to place the second queen in the second row. Clearly, the first square in the second row where the second queen can be placed is the third column. So we place the second queen in that column; see Figure 4-13(b).

Next, we try to place the third queen in the third row. We find that the third queen cannot be placed in the third row and so we arrive at a dead end. At this point we backtrack to the previous board configuration and place the second queen in the fourth column; see Figure 4-13(c). Next, we try to place the third queen in the third row. This time, we successfully place the third queen in the second column of the third row; see Figure 4-13(d). After placing the third queen in the third row, when we try to place the fourth queen, we discover that the fourth queen cannot be placed in the fourth row.

We backtrack to the third row and try placing the queen in any other column. Because no other column is available for queen three, we backtrack to row two and try placing the second queen in any other column,

which cannot be done. We, therefore, backtrack to the first row and place the first queen in the next column. After placing the first queen in the second column, we place the remaining queens in the successive rows. This time we obtain the solution, as shown by Figure 4-14.

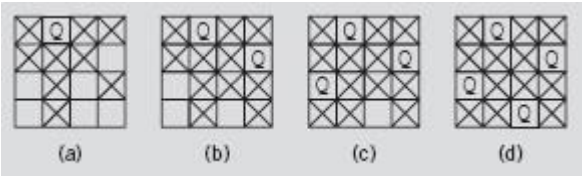


FIGURE 4-14 A solution to the 4-queens puzzle

4.3.3 Backtracking and the 4-Queens Puzzle

Suppose that the rows of the square board of the 4-queens puzzle are numbered 0 through 3 and the columns are numbered 0 through 3. (Recall that in C++, an array index starts at 0.)

For the 4-queens puzzle, we start by placing the first queen in the first row and first column, thus generating the tuple (0). We then place the second queen in the third column of the second row and so generate the tuple (0,2). When we try to place the third queen in the third row, we determined that the third queen cannot be placed in the third row and so we back up to the partial solution (0,2), remove 2 from the tuple and then generate the tuple (0,3), that is, the third queen is placed in the fourth column of the second row. With the partial solution (0,3), we then try to place the third queen in the third row and generate the tuple (0,3,1). Next, with the partial solution (0,3,1), when we try to place the fourth queen in the fourth row, it is determined that it cannot be done and so the partial solution (0,3,1) ends up in a dead end.

From the partial solution (0,3,1), the backtracking algorithm, in fact, backs up to placing the first queen and so removes all the elements of the tuple. The algorithm then places the first queen in the second column of the first row and thus generates the partial solution (1). In this case, the sequence of partial solution generated is, (1), (1,3), (1,3,0), and (1,3,0,2), which represents a solution to the 4-queens puzzle. The solutions generated by the backtracking algorithm can be represented by a tree, as shown in Figure 4-15.

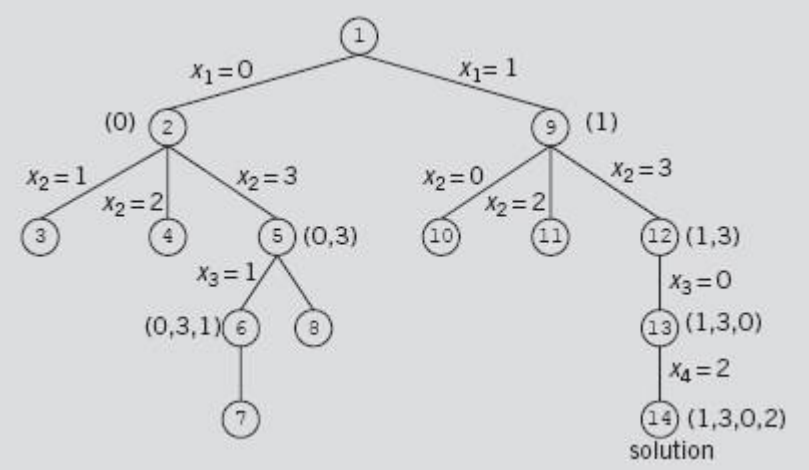


FIGURE 4-15 4-queens tree

4.3.4 8-Queens Puzzle

Let us now consider the 8-queens puzzle. Like the 4-queens puzzle, no two queens can be in the same row, same columns, and same diagonals. Determining whether two queens are in the same row or column is easy because we can check their row and column position. Let us describe how to determine whether two queens are in the same diagonal or not.

Consider the 8 by 8 square board shown in Figure 4-16. The rows are numbered 0 through 7; the columns are numbered 0 through 7. (Recall that, in C++, the array index starts at 0.)

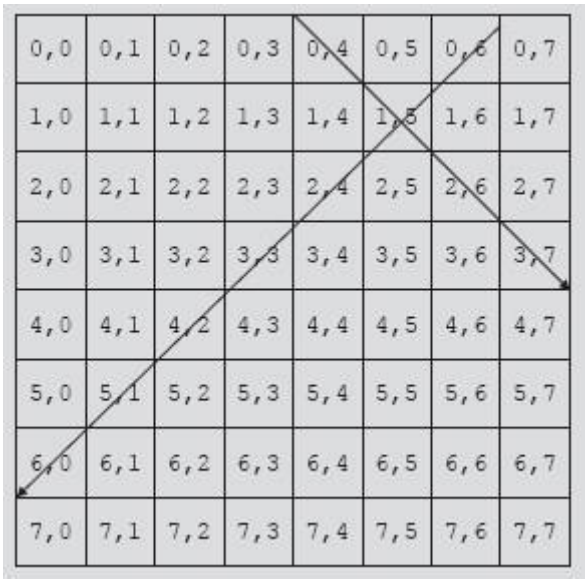


FIGURE 4-16 8 x 8 square board

Consider the diagonal from upper left to lower right, as indicated by the arrow. The positions of the squares on this diagonal are (0,4), (1,5), (2,6), and (3,7). Notice that for these entries **rowPosition –**

columnPosition is – 4. For example, 0 – 4 = 1 – 5 = 2 – 6 = 3 – 7 = – 4. It can be shown that for each square on a diagonal from upper left to lower right, **rowPosition – columnPosition** is the same.

Now consider the diagonal from upper right to lower left as indicated by the arrow. The positions of the squares on this diagonal are (0,6), (1,5), (2,4), (3,3), (4,2), (5,1), and (6,0). Here **rowPosition + columnPosition** = 6. It can be shown that for each square on a diagonal from upper right to lower left, **rowPosition + columnPosition** is the same.

We can use these results to determine whether two queens are on the same diagonal or not. Suppose a queen is at position (i, j), (row i and column j), and another queen is at position (k, l) , (row k and column l). These queens are on the same diagonal if either i + j = k + l or i - j = k – l. The first equation implies that j – l = i – k and the second equation implies that j – l = k – i. From this it follows that the two queens are on the same diagonal if |j – l| = |i – k|, where |j – l| is the absolute value of j – l and so on.

Because a solution to the 8-queens puzzle is represented as an 8-tuple, we use the array **queensInRow** of size 8, where **queensInRow[k]** specifies the column position of the kth queen in row k. For example, **queensInRow[0]** = 3 means the first queen is placed in column 3 (which is the fourth column) of row 0 (which is the first row).

Suppose that we place the first k-1 queens in the first k – 1 rows. Next we try to place the kth queen in a column of the kth row.Wewrite the function **canPlaceQueen(k,i)**,which returns true if the kth queen can be placed in the ith column of row k; otherwise, it returns false.

The first k-1 queens are in the first k-1 rows and we are trying to place the kth queen in the kth row. The kth row, therefore, must first be empty. It thus follows that the kth queen can be placed in column i of row k, provided no other queen is in column i and no queens are on the diagonals on which square (k, i) lies. The general algorithm for the function **canPlaceQueen(k, i)** is as follows:

```
for (int j = 0; j < k; j++)
        if((queensInRow[j] == i)                          //there is already a queen in column i
            || (abs(queensInRow[j] - i) == abs(j-k)))     //there is already
```

```
                                                    //a queen in one of the diagonals
                                    //on which square (k,i) lies return false;
        return true;
```

The for loop checks whether there is already a queen either in column i or in one of the diagonals on which square (k, i) lies. If it finds a queen at any of such positions, the for loop returns the value false. Otherwise, the value true is returned.

The following class defines the n-queens puzzle as an ADT:

```
//*************************************************************
// This class specifies the functions to solve the n-queens puzzle.
//************************************************************* class
nQueensPuzzle
{
        public:
                nQueensPuzzle(int queens = 8);
                //constructor
                //Postcondition: noOfSolutions = 0; noOfQueens = queens;
                // queensInRow is a pointer to the array to store the
                // n-tuple

                bool canPlaceQueen(int k, int i);
                //Function to determine whether a queen can be placed //in
                row k and column i.
                //Postcondition: returns true if a queen can be placed in
                // row k and column i; otherwise it returns false

                void queensConfiguration(int k);
                //Function to determine all solutions to the n-queens //puzzle
                using backtracking.
                //The function is called with the value 0.
                //Postcondition: All n-tuples representing solutions of //
                n-queens puzzle are generated and printed.
```

```
        void printConfiguration();
        //Function to output an n-tuple containing a solution //to
        the n-queens puzzle.

        int solutionsCount();
        //Function to return the total number of solutions.
        //Postcondition: The value of noOfSolution is returned.

    private: int
        noOfSolutions;
        int noOfQueens;
        int *queensInRow;
};
```

The definitions of the member functions of the class nQueensPuzzle are given next:

```
nQueensPuzzle::nQueensPuzzle(int  queens)
{
    noOfQueens = queens;
    queensInRow = new int[noOfQueens];
    noOfSolutions = 0;
}

bool nQueensPuzzle::canPlaceQueen(int k, int i)
{
    for  (int  j  =  0;  j  <  k;  j++)  if
        ((queensInRow[j] == i)
            || (abs(queensInRow[j] - i) == abs(j-k))) return
                false;
    return true;
}
```

Using recursion, the function queensConfiguration implements the backtracking technique to determine

all solutions to the n-queens puzzle. The parameter k specifies the queen to be placed in the kth row. Its definition is straightforward and is given next.

```
void nQueensPuzzle::queensConfiguration(int k)
{
        for (int i = 0; i < noOfQueens; i++)
        {
                if (canPlaceQueen(k, i))
                {
                        queensInRow[k] = i;                     //place the kth queen in column i
                        if (k == noOfQueens - 1)                //all the queens are placed
                                printConfiguration();           //print the n-tuple
                        else queensConfiguration(k + 1);        //determine the place
                                                                //for the (k+1)th queen
                }
        }
}

void nQueensPuzzle::printConfiguration()
{
        noOfSolutions++; cout
        << "(";
        for (int i = 0; i < noOfQueens - 1; i++) cout
                << queensInRow[i] << ", ";
        cout << queensInRow[noOfQueens - 1] << ")" << endl;
}

int nQueensPuzzle::solutionsCount()
```

```
{
        return noOfSolutions;
}
```

We leave it as an exercise for you to write a program to test the n-queens puzzle class for various board sizes.

4.4 Recursion, Backtracking, and Sudoku

In the previous section, we used recursion and backtracking to solve the 8-queens problem. In this section, we introduce the well-known sudoku problem, which can be solved using recursion and backtracking. This problem involves filling numbers 1 to 9 in a partially filled, 9 x 9 grid with constraints described in this section. Figure 4-17(a) shows a partially filled grid.

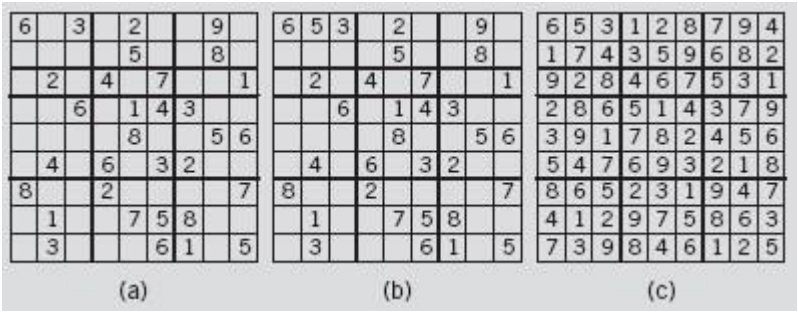


FIGURE 4-17 Sudoku problem and its solution

The sudoku grid is a 9 x 9 grid consisting of nine rows, nine columns, and nine 3 x 3 smaller grids. In Figure 4-17, the nine 3 x 3 smaller grids are separated by darker horizontal and vertical lines. The first 3 x 3 smaller grid is in rows 1 to 3 and columns 1 to 3, the second 3 x 3 smaller grid is in rows 1 to 3 and columns 4 to 6, and so on.

The objective is to fill the entire grid with numbers 1 to 9 such that each number appears exactly once in each row, each column, and each 3 x 3 smaller grid. For example, the solution to the sudoku problem in Figure 4-17(a) is shown in Figure 4-17(c).

Next, we describe, a simple recursive backtracking algorithm to find the solution to the sudoku problem in Figure 4-17(a).

Starting at the first row, we find the first empty grid slot. For example, in Figure 4-17(a), the first empty grid slot is in the second row and second column. Next we find the first number, between 1 and 9, that can be placed in this slot. (Note that before placing a number in an empty slot, we must check that the number does not appear in the row, column, and the 3 x 3 grid containing the slot.) For example, in Figure 4-17(a), the first number that can be placed in the second row and second column is 5; see Figure 4-17(b). After placing a number in the first empty slot, we find the next empty slot and try to place a number in that slot. If a number cannot be placed in a slot, then we must backtrack to the previous slot, where the number was placed, place a different number, and continue. If we arrive at a slot where no number can be placed, then the sudoku problem has no solutions.

The following class implements the sudoku problem as an ADT:

```
//*************************************************************
// This class specifies the functions to solve a sudoku problem.
//*************************************************************
class sudoku
{
    public:
        sudoku();
        //default constructor
        //Postcondition: grid is initialized to 0

        sudoku(int g[][9]);
        //constructor
        //Postcondition: grid = g

        void initializeSudokuGrid();
        //Function to prompt the user to specify the numbers of the //partially
        filled grid.
        //Postcondition: grid is initialized to the numbers //
        specified by the user.

        void initializeSudokuGrid(int g[][9]);
      //Function to initialize grid to g
```

```
//Postcondition: grid = g;

void printSudokuGrid();
//Function to print the sudoku grid.

bool solveSudoku();
//Funtion to solve the sudoku problem.
//Postcondition: If a solution exits, it returns true, // otherwise it returns false.

bool findEmptyGridSlot(int &row, int &col);
//Function to determine if the grid slot specified by
//row and col is empty.
//Postcondition: Returns true if grid[row][col] = 0;

bool canPlaceNum(int row, int col, int num); //Function to determine if num can be placed in
//grid[row][col]
//Postcondition: Returns true if num can be placed in // grid[row][col], otherwise it returns false.

bool numAlreadyInRow(int row, int num);
//Function to determine if num is in grid[row][] //Postcondition: Returns true if num is in grid[row][],
// otherwise it returns false.
```

```
        bool numAlreadyInCol(int col, int num); //Function
        to determine if num is in grid[row][] //Postcondition:
        Returns true if num is in grid[row][], // otherwise it
        returns false.

        bool numAlreadyInBox(int smallGridRow, int smallGridCol,int num);
        //Function to determine if num is in the small grid
        //Postcondition: Returns true if num is in small grid,
        // otherwise it returns false.

    private:
        int grid[9][9];
};
```

Note that we use the digit 0 to specify an empty grid. For example, the partially filled sudoku grid in Figure 4-17(a) is entered and stored as:

```
6 0 3 0 2 0 0 9 0
0 0 0 0 5 0 0 8 0
0 2 0 4 0 7 0 0 1
0 0 6 0 1 4 3 0 0
0 0 0 0 8 0 0 5 6
0 4 0 6 0 3 2 0 0
8 0 0 2 0 0 0 0 7
0 1 0 0 7 5 8 0 0
0 3 0 0 0 6 1 0 5
```

Next we write the definition of the function solveSudoku.

The function solveSudoku uses recursion and backtracking to find a solution, if one exists, of the partially filled sudoku grid. The general algorithm is as follows:

1. Find the position of the first empty slot in the partially filled grid. If the grid has no empty slots, return true and print the solution.
2. Suppose the variables row and col specify the position of the empty grid position.

```
for (int digit = 1; digit <= 9; digit++)
{
        if (grid[row][col] <> digit)
        { grid[row][col] = digit; recursively fill the updated grid; if the
                grid is filled successfully, return true, otherwise remove
                the assigned digit from grid[row][col] and try another
                digit.
        }
```

If all the digits have been tried and nothing worked, return false.

The definition of the function is as follows:

```
bool sudoku::solveSudoku()
{
        int row, col;

        if (findEmptyGridSlot(row, col))
        {
                for (int num = 1; num <= 9; num++)
                {
                        if (canPlaceNum(row, col, num))
                        {
                                grid[row][col] = num;
                                if (solveSudoku())      return          //recursive call
                                        true;
                                grid[row][col] = 0;
                        }
                }
                return false;                                           //backtrack
        }
```

```
        else

            return true;                                    //there are no empty slots
}
```

We leave it is as an exercise for you to write the definitions of the remaining functions of the class sudoku and program that solves sudoku problems.

Key Terms

1. The process of solving a problem by reducing it to smaller versions of itself is called recursion.
2. A recursive definition defines a problem in terms of smaller versions of itself.
3. Every recursive definition has one or more base cases.
4. A recursive algorithm solves a problem by reducing it to smaller versions of itself.
5. Every recursive algorithm has one or more base cases.
6. The solution to the problem in a base case is obtained directly.
7. A function is called recursive if it calls itself.
8. Recursive algorithms are implemented using recursive functions.
9. Every recursive function must have one or more base cases.
10. The general solution breaks the problem into smaller versions of itself.
11. The general case must eventually be reduced to a base case.
12. The base case stops the recursion. 13. While tracing a recursive function:

- Logically, you can think of a recursive function as having an unlimited number of copies of itself. • Every call to a recursive function—that is, every recursive call—has its own code and its own set of parameters and local variables.
- After completing a particular recursive call, control goes back to the calling environment, which is the previous call. The current (recursive) call must execute completely before control goes back to the previous call. The execution in the previous call begins from the point immediately following the recursive call.

14. A function is called directly recursive if it calls itself.
15. A function that calls another function and eventually results in the original function call is said to be indirectly recursive.
16. A recursive function in which the last statement executed is the recursive call is called a tail recursive function.
17. To design a recursive function, you must do the following:

- Understand the problem requirements.
- Determine the limiting conditions. For example, for a list the limiting condition is the number of elements in the list.
- Identify the base cases and provide a direct solution to each base case.
- Identify the general cases and provide a solution to each general case in terms of smaller versions of itself.

Review Questions

1. Mark the following statements as true or false.

a. Every recursive definition must have one or more base cases.

b. Every recursive function must have one or more base cases.

c. The general case stops the recursion.

d. In the general case, the solution to the problem is obtained directly.

e. A recursive function always returns a value.

2. What is a base case?

3. What is a recursive case?

4. What is direct recursion?

5. What is indirect recursion?

6. What is tail recursion?

7. Consider the following recursive function:

```
int mystery(int number)                                    //Line 1
{                                                          //Line 2 if
        (number == 0)                                      //Line 3
        return number;                                    //Line 4 else
                                                          //Line 5
         return(number + mystery(number - 1));            //Line 6
}                                                          //Line 7
```

a. Identify the base case.

b. Identify the general case.

c. What valid values can be passed as parameters to the function mystery?

d. If mystery(0) is a valid call, what is its value? If not, explain why.

e. If mystery(5) is a valid call, what is its value? If not, explain why.

f. If mystery(-3) is a valid call, what is its value? If not, explain why.

8. Consider the following recursive function:

```
void funcRec(int u, char v)                                          //Line 1
{                                                                    //Line 2 if
        (u == 0)                                                     //Line 3
        cout << v;                                                   //Line 4 else if (u
        == 1)                                                        //Line 5
                cout << static_cast<char>(static_cast<int>(v) + 1);  //Line 6 else
                                                                     //Line 7
        funcRec(u - 1, v);                                           //Line 8
}                                                                    //Line 9
```

Answer the following questions: a.

Identify the base case.

b. Identify the general case.

c. What is the output of the following statement?

```
funcRec(5, 'A');
```

9. Consider the following recursive function: void

```
exercise(int x)
{
        if (x > 0 && x < 10)
        {
                cout << x << " ";
                exercise(x + 1);
        }
}
```

What is the output of the following statements? a.

exercise(0);

b. exercise(5);

c. exercise(10);

d. exercise(-5);

10. Consider the following function:

```
int test(int x, int y)
{ if (x == y) return x;
        else if (x > y) return (x
                + y);
        else return test(x + 1, y - 1);
}
```

What is the output of the following statements? a.

```
cout << test(5, 10) << endl;
```

b. `cout << test(3, 9) << endl;`

11. Consider the following function:

```
int func(int x)
{ if (x == 0) return 2;
        else if (x == 1) return
                3;
        else return (func(x - 1) + func(x - 2));
}
```

What is the output of the following statements? a.

```
cout << func(0) << endl;
```

b. `cout << func(1) << endl;`

c. `cout << func(2) << endl;`

d. `cout << func(5) << endl;`

12. Suppose that intArray is an array of integers, and length specifies the number of elements in intArray. Also, suppose that low and high are two integers such that 0 <= low < length, 0 <= high < length, and low < high. That is, low and high are two indices in intArray. Write a recursive definition that reverses the elements in intArray between low and high.

13. Write a recursive algorithm to multiply two positive integers m and n using repeated addition. Specify the base case and the recursive case.

14. Consider the following problem: How many ways can a committee of four people be selected from a group of 10 people? There are many other similar problems, where you are asked to find the number of ways to select a set of items from a given set of items. The general problem can be stated as follows:

Find the number of ways r different things can be chosen from a set of n items, where r and n are nonnegative integers and r ≤ n. Suppose C(n, r) denotes the number of ways r different things can be chosen from a set of n items. Then C(n, r) is given by the following formula:

$$C(n,r) = \frac{n!}{r!(n-r)!}$$

where the exclamation point denotes the factorial function.Moreover, C(n, 0) = C(n, n) = 1. It is also known that C(n, r) = C(n – 1, r – 1) + C(n – 1, r).

a. Write a recursive algorithm to determine C(n, r). Identify the base case(s) and the general case(s).

b. Using your recursive algorithm, determine C(5, 3) and C(9, 4).

5 TOPIC 5: STACKS



LEARNING OUTCOMES:

After studying this topic you should be able to:

- Learn about stacks
- Examine various stack operations
- Learn how to implement a stack as an array
- Learn how to implement a stack as a linked list
- Discover stack applications
- Learn how to use a stack to remove recursion

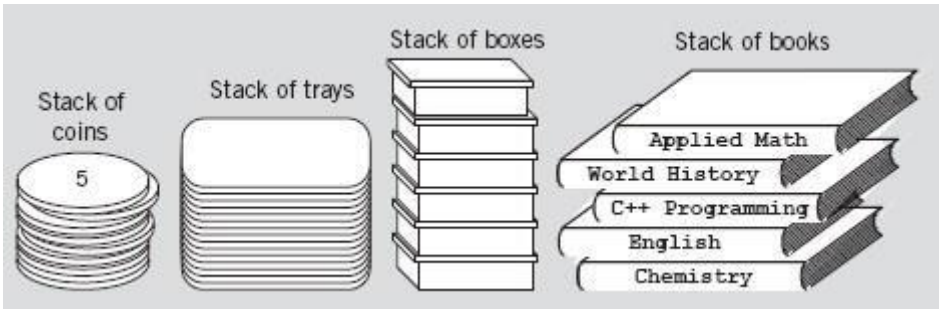
This chapter discusses a very useful data structure called a stack. It has numerous applications in computer science.

5.1 Stacks

Suppose that you have a program with several functions. To be specific, suppose that you have the functions A, B, C, and D in your program. Now suppose that function A calls function B, function B calls function C, and function C calls function D. When function D terminates, control goes back to function C; when function C terminates, control goes back to function B; and when function B terminates, control goes back to function A. During program execution, how do you think the computer keeps track of the function calls? What about recursive functions? How does the computer keep track of the recursive calls? In Chapter 6, we designed a recursive function to print a linked list backward. What if you want to write a nonrecursive algorithm to print a linked list backward?

This section discusses the data structure called the stack, which the computer uses to implement function calls. You can also use stacks to convert recursive algorithms into nonrecursive algorithms, especially recursive algorithms that are not tail recursive. Stacks have numerous other applications in computer science. After developing the tools necessary to implement a stack, we will examine some applications of stacks.

A stack is a list of homogenous elements in which the addition and deletion of elements occurs only at one end, called the top of the stack. For example, in a cafeteria, the second tray in a stack of trays can be removed only if the first tray has been removed. For another example, to get to your favorite computer science book, which is underneath your math and history books, you must first remove the math and



history books. After removing these books, the computer science book becomes the top book—that is, the top element of the stack. Figure 5-1 shows some examples of stacks.

FIGURE 5-1 Various examples of stacks

The elements at the bottom of the stack have been in the stack the longest. The top element of the stack is the last element added to the stack. Because the elements are added and removed from one end (that is, the top), it follows that the item that is added last will be removed first. For this reason, a stack is also called a Last In First Out (LIFO) data structure.

Stack: A data structure in which the elements are added and removed from one end only; a Last In First Out (LIFO) data structure.

Now that you know what a stack is, let us see what kinds of operations can be performed on a stack. Because new items can be added to the stack, we can perform the add operation, called push, to add an element onto the stack. Similarly, because the top item can be retrieved and/or removed from the stack, we can perform the operation top to retrieve the top element of the stack, and the operation pop to remove the top element from the stack.

The push, top, and pop operations work as follows: Suppose there are boxes lying on the floor that need to be stacked on a table. Initially, all of the boxes are on the floor and the stack is empty. (See Figure 52.)



FIGURE 5-2 Empty stack

First we push box A onto the stack. After the push operation, the stack is as shown in Figure 5-3(a).

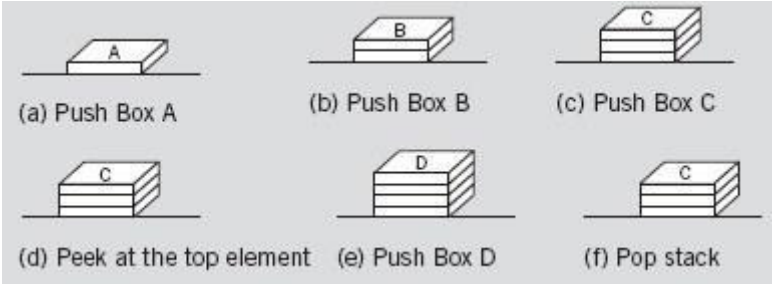


FIGURE 5-3 Stack operations

We then push box B onto the stack. After this push operation, the stack is as shown in Figure 5-3(b). Next, we push box C onto the stack. After this push operation, the stack is as shown in Figure 5-3(c).
Next, we look at the top element of the stack. After this operation, the stack is unchanged and shown in Figure 5-3(d). We then push box D onto the stack. After this push operation, the stack is as shown in Figure 5-3(e). Next, we pop the stack. After the pop operation, the stack is as shown in Figure 5-3(f).

An element can be removed from the stack only if there is something in the stack, and an element can be added to the stack only if there is room. The two operations that immediately follow from push, top, and pop are isFullStack (checks whether the stack is full) and isEmptyStack (checks whether the stack is empty). Because a stack keeps changing as we add and remove elements, the stack must be empty before we first start using it. Thus, we need another operation, called initializeStack, which initializes the stack to an empty state. Therefore, to successfully implement a stack, we need at least these six operations, which are described next. (We might also need other operations on a stack, depending on the specific implementation.)

- initializeStack—Initializes the stack to an empty state.
- isEmptyStack—Determines whether the stack is empty. If the stack is empty, it returns the value true; otherwise, it returns the value false.
- isFullStack—Determines whether the stack is full. If the stack is full, it returns the value true; otherwise, it returns the value false.
- push—Adds a new element to the top of the stack. The input to this operation consists of the stack and the new element. Prior to this operation, the stack must exist and must not be full.
- top—Returns the top element of the stack. Prior to this operation, the stack must exist and must not be empty.
- pop—Removes the top element of the stack. Prior to this operation, the stack must exist and must not be empty.

The following abstract class stackADT defines these operations as an ADT:

```
//**********************************************************
// This class specifies the basic operations on a stack.
//**********************************************************
* template <class Type> class stackADT
{ public: virtual void initializeStack() = 0;
                //Method to initialize the stack to an empty state.
                //Postcondition: Stack is empty.

                virtual bool isEmptyStack() const = 0;
                //Function to determine whether the stack is empty.
                //Postcondition: Returns true if the stack is empty, //
                otherwise returns false.

                virtual bool isFullStack() const = 0;
                //Function to determine whether the stack is full.
                //Postcondition: Returns true if the stack is full, //
                otherwise returns false.

                virtual void push(const Type& newItem) = 0; //Function
                to add newItem to the stack.
                //Precondition: The stack exists and is not full.
                //Postcondition: The stack is changed and newItem is added //
                to the top of the stack.

                virtual Type top() const = 0;
                //Function to return the top element of the stack. //Precondition:
                The stack exists and is not empty.
                //Postcondition: If the stack is empty, the program //
                terminates; otherwise, the top element of the stack //
                is returned.
```

```
        virtual void pop() = 0;
        //Function to remove the top element of the stack.
        //Precondition: The stack exists and is not empty.
        //Postcondition: The stack is changed and the top element //
        is removed from the stack.
};
```

Figure 5-4 shows the UML class diagram of the class stackADT.

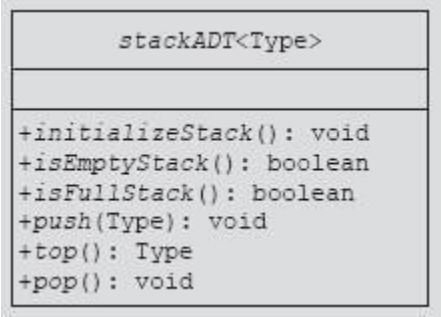


FIGURE 5-4 UML class diagram of the class stackADT

We now consider the implementation of our abstract stack data structure. Functions such as push and pop that are required to implement a stack are not available to C++ programmers. We must write the functions to implement the stack operations.

Because all the elements of a stack are of the same type, a stack can be implemented as either an array or a linked structure. Both implementations are useful and are discussed in this chapter.

5.1.1 Implementation of Stacks as Arrays

Because all the elements of a stack are of the same type, you can use an array to implement a stack. The first element of the stack can be put in the first array slot, the second element of the stack in the second array slot, and so on. The top of the stack is the index of the last element added to the stack.

In this implementation of a stack, stack elements are stored in an array, and an array is a random access data structure; that is, you can directly access any element of the array. However, by definition, a stack is a data structure in which the elements are accessed (popped or pushed) at only one end—that is, a Last In First Out data structure. Thus, a stack element is accessed only through the top, not through the bottom

or middle. This feature of a stack is extremely important and must be recognized in the beginning.

To keep track of the top position of the array, we can simply declare another variable, called stackTop.

The following class, stackType, implements the functions of the abstract class stackADT. By using a pointer, we can dynamically allocate arrays, so we will leave it for the user to specify the size of the array (that is, the stack size). We assume that the default stack size is 100. Because the class stackType has a pointer member variable (the pointer to the array to store the stack elements), we must overload the assignment operator and include the copy constructor and destructor. Moreover, we give a generic definition of the stack. Depending on the specific application, we can pass the stack element type when we declare a stack object.

```
//*************************************************************
// This class specifies the basic operation on a stack as an array.
//*************************************************************
******* template <class Type>
class stackType: public stackADT<Type>
{
        public:
                const stackType<Type>& operator=(const stackType<Type>&); //Overload
                the assignment operator.

                void initializeStack();
                //Function to initialize the stack to an empty state.
                //Postcondition: stackTop = 0;

                bool isEmptyStack() const;
                //Function to determine whether the stack is empty.
                //Postcondition: Returns true if the stack is empty, //
                otherwise returns false.

                bool isFullStack() const;
```

```
//Function to determine whether the stack is full.
//Postcondition: Returns true if the stack is full, // otherwise returns
false.

void push(const Type& newItem);
//Function to add newItem to the stack.
//Precondition: The stack exists and is not full.
//Postcondition: The stack is changed and newItem is
// added to the top of the stack.

Type top() const;
//Function to return the top element of the stack. //Precondition: The stack exists
and is not empty.
//Postcondition: If the stack is empty, the program // terminates;
otherwise, the top element of the stack // is returned.

void pop();
//Function to remove the top element of the stack.
//Precondition: The stack exists and is not empty.
//Postcondition: The stack is changed and the top element is // removed from the
stack.

stackType(int stackSize = 100);
//Constructor
//Create an array of the size stackSize to hold //the stack elements.
The default stack size is 100.
//Postcondition: The variable list contains the base address
// of the array, stackTop = 0, and maxStackSize = stackSize

stackType(const stackType<Type>& otherStack);
//Copy constructor
                ~stackType();
```

```
        //Destructor
        //Remove all the elements from the stack.
        //Postcondition: The array (list) holding the stack //
        elements is deleted.

    private:
        int maxStackSize; //variable to store the maximum stack size
        int stackTop; //variable to point to the top of the stack Type
        *list; //pointer to the array that holds the stack elements

        void copyStack(const stackType<Type>& otherStack); //Function
        to make a copy of otherStack.
        //Postcondition: A copy of otherStack is created and assigned //
        to this stack.
};
```

Figure 5-5 shows the UML class diagram of the class stackType.

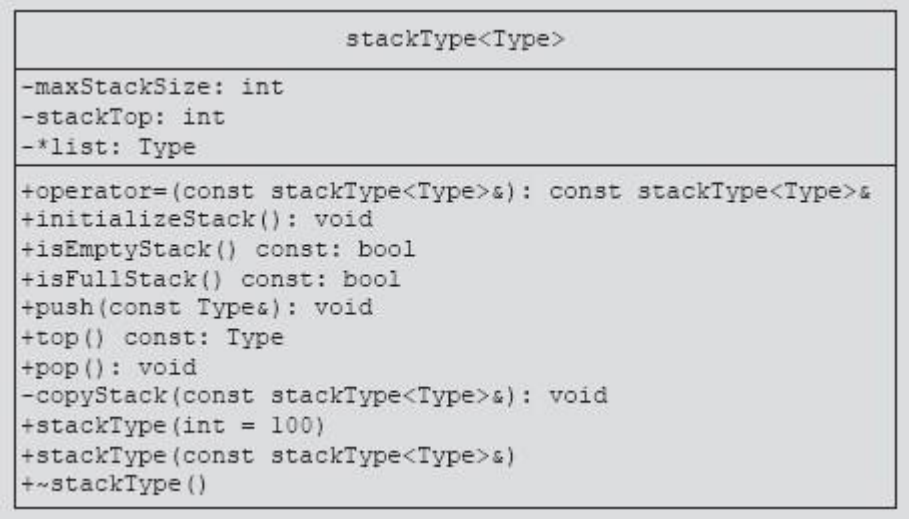


FIGURE 5-5 UML class diagram of the class stackType

Figure 5-6 shows this data structure, wherein stack is an object of type stackType. Note that stackTop can range from 0 to maxStackSize. If stackTop is nonzero, then stackTop - 1 is the index of the top element of the stack. Suppose that maxStackSize = 100.

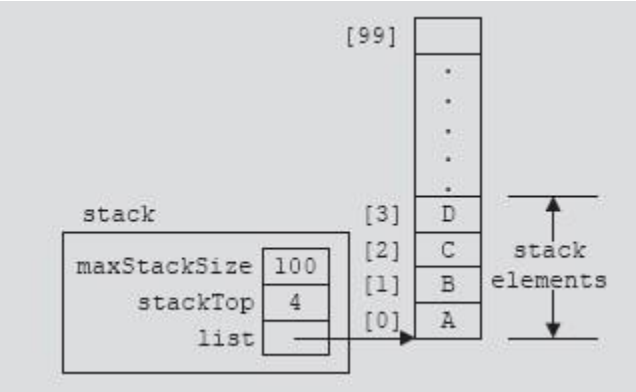


FIGURE 5-6 Example of a stack

Note that the pointer list contains the base address of the array (holding the stack elements)—that is, the address of the first array component. Next we discuss how to implement the member functions of the class stackType.

5.1.2 Initialize Stack

Let us consider the initializeStack operation. Because the value of stackTop indicates whether the stack is empty, we can simply set stackTop to 0 to initialize the stack. (See Figure 5-7.)

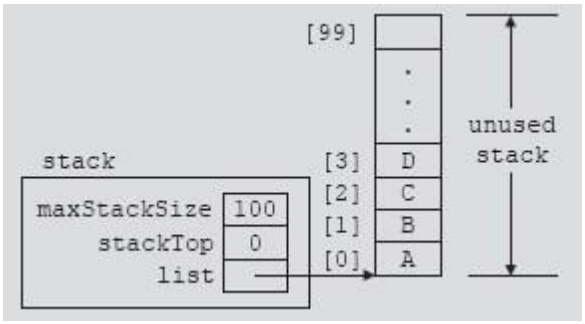


FIGURE 5-7 Empty stack

The definition of the function initializeStack is as follows:

```
template <class Type>
void stackType<Type>::initializeStack()
{
        stackTop = 0;
}                                                    //end initializeStack
```

5.1.3 Empty Stack

We have seen that the value of stackTop indicates whether the stack is empty. If stackTop is 0, the stack is empty; otherwise, the stack is not empty. The definition of the function isEmptyStack is as follows:

```
template <class Type>
bool stackType<Type>::isEmptyStack() const
{
        return(stackTop == 0);
}                                                    //end isEmptyStack
```

5.1.4 Full Stack

Next, we consider the operation isFullStack. It follows that the stack is full if stackTop is equal to maxStackSize. The definition of the function isFullStack is as follows:

```
template <class Type>
bool stackType<Type>::isFullStack() const
{
        return(stackTop == maxStackSize);
```

} //end isFullStack

5.1.5 Push

Adding, or pushing, an element onto the stack is a two-step process. Recall that the value of stackTop indicates the number of elements in the stack, and stackTop - 1 gives the position of the top element of the stack. Therefore, the push operation is as follows:

1. Store the newItem in the array component indicated by stackTop.
2. Increment stackTop.

Figure 5-8(a) shows the stack before pushing 'y' into the stack. Figure 5-8(b) shows the stack after pushing 'y' into the stack.

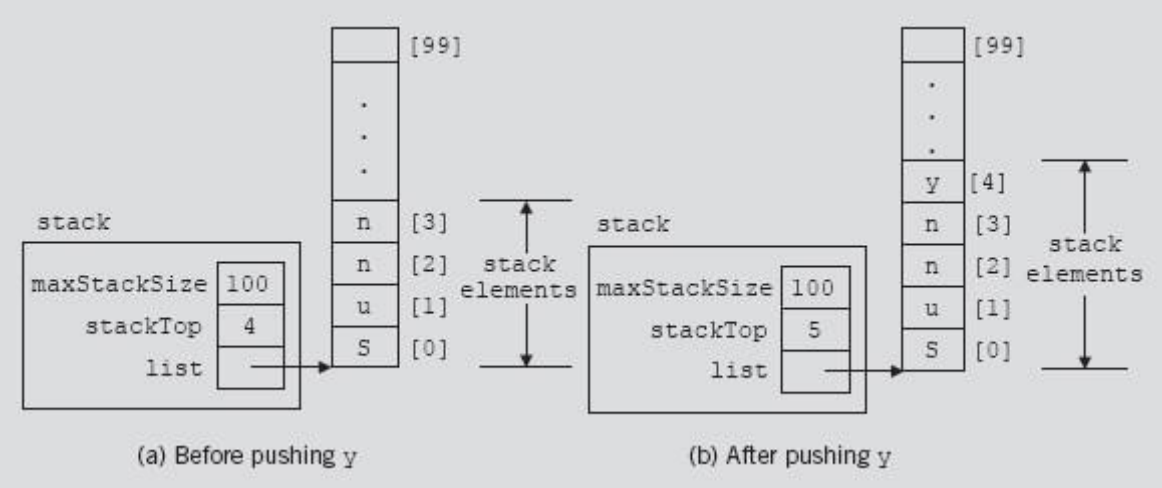


FIGURE 5-8 Stack before and after the push operation

The definition of the function push is as follows:

```
template <class Type>
void stackType<Type>::push(const Type& newItem)
{
    if (!isFullStack())
    {
        list[stackTop] = newItem;          //add newItem at the top stackTop++;
                                           //increment stackTop
    }
    else
```

```
cout << "Cannot       add to a full stack." << endl;
    }                                                    //end push
```

If we try to add a new item to a full stack, the resulting condition is called an overflow. Error checking for an overflow can be handled in different ways. One way is as shown previously. Or, we can check for an overflow before calling the function push, as shown next (assuming stack is an object of type stackType).

```
if (!stack.isFullStack()) stack.push(newItem);
```

5.1.6 Return the Top Element

The operation top returns the top element of the stack. Its definition is as follows:

```
template <class Type>
Type stackType<Type>::top() const
{
        assert(stackTop != 0);          //if stack is empty, terminate the
                                        //program
        return list[stackTop - 1];      //return the element of the stack
                                        //indicated by stackTop - 1
}                                       //end top
```

5.1.7 Pop

To remove, or pop, an element from the stack, we simply decrement stackTop by 1. Figure 5-9(a) shows the stack before popping 'D' from the stack. Figure 5-9(b) shows the stack after popping 'D' from the stack.

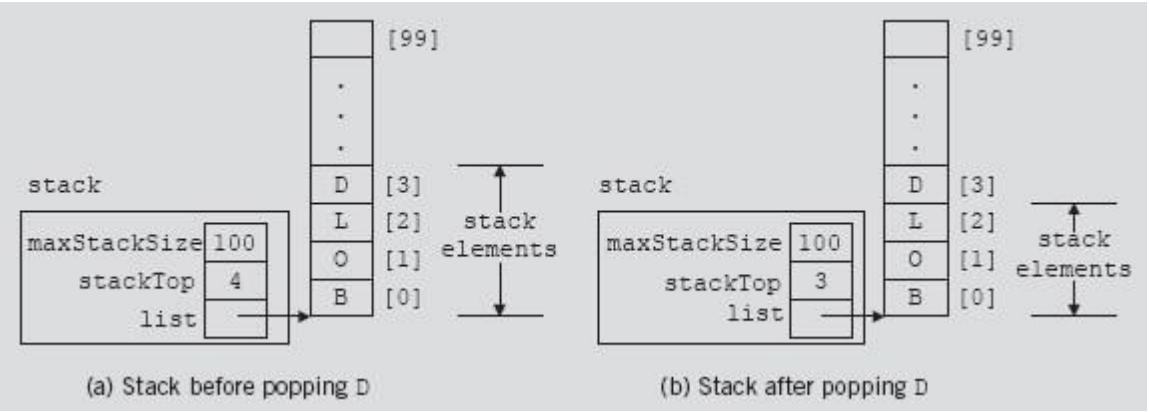


FIGURE 5-9 Stack before and after the pop operation

The definition of the function pop is as follows:

```
template <class Type>
void stackType<Type>::pop()
{
        if (!isEmptyStack())
                stackTop--;  //decrement stackTop else cout << "Cannot
```

```
        remove from an empty stack." << endl;
}                                                   //end pop
```

If we try to remove an item from an empty stack, the resulting condition is called an underflow. Error checking for an underflow can be handled in different ways. One way is as shown previously. Or, we can check for an underflow before calling the function pop, as shown next (assuming stack is an object of type stackType).

```
                    if (!stack.isEmptyStack()) stack.pop();
```

5.1.8 Copy Stack

The function copyStack makes a copy of a stack. The stack to be copied is passed as a parameter to the function copyStack. We will, in fact, use this function to implement the copy constructor and overload the assignment operator. The definition of this function is as follows:

```
template <class Type>
void stackType<Type>::copyStack(const stackType<Type>& otherStack)
{
        delete [] list;
        maxStackSize = otherStack.maxStackSize; stackTop
        = otherStack.stackTop;

        list = new Type[maxStackSize];
                                                //copy otherStack into this stack
        for (int j = 0; j < stackTop; j++)
                list[j] = otherStack.list[j];

}                                               //end copyStack
```

5.1.9 Constructor and Destructor

The functions to implement the constructor and the destructor are straightforward. The constructor with parameters sets the stack size to the size specified by the user, sets stackTop to 0, and creates an appropriate array in which to store the stack elements. If the user does not specify the size of the array in which to store the stack elements, the constructor uses the default value, which is 100, to create an array of size 100. The destructor simply deallocates the memory occupied by the array (that is, the stack) and sets stackTop to 0. The definitions of the constructor and destructor are as follows:

```
template <class Type>
stackType<Type>::stackType(int stackSize)
{
```

```
if (stackSize <= 0)
{
        cout << "Size of the array to hold the stack must "<< "be positive." << endl;
        cout << "Creating an array of size 100." << endl; maxStackSize = 100;
}
else
        maxStackSize = stackSize;                    //set the stack size to
                                                     //the value specified by
                                                     //the parameter stackSize
stackTop = 0;                                        //set stackTop to 0
list = new Type[maxStackSize];                       //create the array to
                                                     //hold the stack elements
```

```
}                                                   //end constructor

template <class Type>
stackType<Type>::~stackType() //destructor
{
        delete [] list;                             //deallocate the memory occupied
                                                    //by the array
}                                                   //end destructor
```

5.1.10 Copy Constructor

The copy constructor is called when a stack object is passed as a (value) parameter to a function. It copies the values of the member variables of the actual parameter into the corresponding member variables of the formal parameter. Its definition is as follows:

```
template <class Type>
stackType<Type>::stackType(const stackType<Type>& otherStack)
{
        list = NULL;
        copyStack(otherStack);
}                                                   //end copy constructor
```

5.1.11 Overloading the Assignment Operator (=)

Recall that for classes with pointer member variables, the assignment operator must be explicitly overloaded. The definition of the function to overload the assignment operator for the class stackType is as follows:

```
template <class Type>
const stackType<Type>& stackType<Type>::operator=(const stackType<Type>& otherStack)
{
        if (this != &otherStack)                    //avoid self-copy
```

```
            copyStack(otherStack);
        return *this;
}                                                   //end operator=
```

5.1.12 Stack Header File

Now that you know how to implement the stack operations, you can put the definitions of the class and the functions to implement the stack operations together to create the stack header file. For the sake of completeness, we next describe the header file. Suppose that the name of the header file containing the definition of the class stackType is myStack.h. We will refer to this header file in any program that uses a stack.

```
//Header file: myStack.h
#ifndef H_StackType
#define H_StackType
```

```
#include <iostream>
#include <cassert>

#include "stackADT.h" using
namespace std;
//Place the definition of the class template stackType, as given //previously
in this chapter, here.
//Place the definitions of the member functions as discussed here. #endif
```

The analysis of the stack operations is similar to the operations of the class arrayListType. We, therefore, provide only a summary in Table 5-1.

TABLE 5-1: Time complexity of the operations of the class stackType on a stack with n

Function	Time complexity
`isEmptyStack`	$O(1)$
`isFullStack`	$O(1)$
`initializeStack`	$O(1)$
constructor	$O(1)$
`top`	$O(1)$
`push`	$O(1)$
`pop`	$O(1)$
`copyStack`	$O(n)$
destructor	$O(1)$
copy constructor	$O(n)$
Overloading the assignment operator	$O(n)$

elements

EXAMPLE 5-1

Before we give a programming example, let us first write a simple program that uses the class stackType and tests some of the stack operations. Among others, we will test the assignment operator and the copy

constructor. The program and its output are as follows:

```
//*************************************************************
// This program tests various operations of a stack.
//*************************************************************
#include <iostream>
#include "myStack.h"
using namespace std;
```

```
void testCopyConstructor(stackType<int> otherStack); int
main()
{
        stackType<int> stack(50);
        stackType<int> copyStack(50);
        stackType<int>
        dummyStack(100);
        stack.initializeStack();
        stack.push(23); stack.push(45);
        stack.push(38);
        copyStack = stack;                          //copy stack into copyStack

        cout << "The elements of copyStack: ";
        while (!copyStack.isEmptyStack())           //print copyStack
        {
                cout << copyStack.top() << " ";
                copyStack.pop();
        }
        cout << endl;

        copyStack = stack;
        testCopyConstructor(stack);                 //test the copy constructor

        if (!stack.isEmptyStack()) cout << "The original stack is
                not empty." << endl
                        << "The top element of the original stack: "
                        << copyStack.top() << endl;

        dummyStack = stack;                         //copy stack into dummyStack
        cout << "The elements of dummyStack: ";
        while (!dummyStack.isEmptyStack())          //print dummyStack
        {
                cout << dummyStack.top() << " ";
```

```
            dummyStack.pop();
        }

        cout << endl;

        return 0;
}

void testCopyConstructor(stackType<int> otherStack)
{
```

```
        if     (!otherStack.isEmptyStack())          cout     <<
                  "otherStack is not empty." << endl
                        << "The top element of otherStack: "
                        << otherStack.top() << endl;
}
```

Sample Run:

The elements of copyStack: 38 45 23 otherStack
is not empty.
The top element of otherStack: 38 The
original stack is not empty.
The top element of the original stack: 38
The elements of dummyStack: 38 45 23

It is recommended that you do a walk-through of this program.

5.2 Linked Implementation of Stacks

Because an array size is fixed, in the array (linear) representation of a stack, only a fixed number of elements can be pushed onto the stack. If in a program the number of elements to be pushed exceeds the size of the array, the program might terminate in an error. We must overcome this problem.

We have seen that by using pointer variables we can dynamically allocate and deallocate memory, and by using linked lists we can dynamically organize data (such as an ordered list). Next, we will use these concepts to implement a stack dynamically.

Recall that in the linear representation of a stack, the value of stackTop indicates the number of elements in the stack, and the value of stackTop - 1 points to the top item in the stack. With the help of stackTop, we can do several things: Find the top element, check whether the stack is empty, and so on.

Similar to the linear representation, in a linked representation stackTop is used to locate the top element in the stack. However, there is a slight difference. In the former case, stackTop gives the index of the array. In the latter case, stackTop gives the address (memory location) of the top element of the stack.

The following class implements the functions of the abstract class stackADT:

```
//*************************************************************
// This class specifies the basic operation on a stack as a linked list.
//*************************************************************
//Definition of the node template
<class Type>
struct nodeType
{
        Type info;
        nodeType<Type> *link;
};

template <class Type>
class linkedStackType: public stackADT<Type>
```

```
{
    public:
        const linkedStackType<Type>& operator=(const linkedStackType<Type>&); //Overload
        the assignment operator.

        bool isEmptyStack() const;
        //Function to determine whether the stack is empty.
        //Postcondition: Returns true if the stack is empty; //
        otherwise returns false.

        bool isFullStack() const;
        //Function to determine whether the stack is full.
        //Postcondition: Returns false.

        void initializeStack();
        //Function to initialize the stack to an empty state.
        //Postcondition: The stack elements are removed;
        // stackTop = NULL;

        void push(const Type& newItem);
        //Function to add newItem to the stack.
        //Precondition: The stack exists and is not full.
        //Postcondition: The stack is changed and newItem is
        // added to the top of the stack.

        Type top() const;
        //Function to return the top element of the stack. //Precondition:
        The stack exists and is not empty.
        //Postcondition: If the stack is empty, the program
        // terminates; otherwise, the top element of //
        the stack is returned.

        void pop();
```

```
//Function to remove the top element of the stack.
//Precondition: The stack exists and is not empty.
//
//Postcondition: The stack is changed and the top // element is removed from the stack.

    linkedStackType(); //Default constructor
    //Postcondition: stackTop = NULL;

    linkedStackType(const linkedStackType<Type>& otherStack);
    //Copy constructor

    ~linkedStackType();
    //Destructor
    //Postcondition: All the elements of the stack are removed.
```

```
private:
        nodeType<Type> *stackTop;           //pointer to the stack

        void copyStack(const linkedStackType<Type>& otherStack); //Function
        to make a copy of otherStack.
        //Postcondition: A copy of otherStack is created and //
        assigned to this stack.
};
```

We leave the UML class diagram of the class linkedStackType as an exercise for you. (See Exercise 12 at the end of this chapter.)

EXAMPLE 5-2

Suppose that stack is an object of type linkedStackType. Figure 5-10(a) shows an empty stack and Figure 5-10(b) shows a nonempty stack.

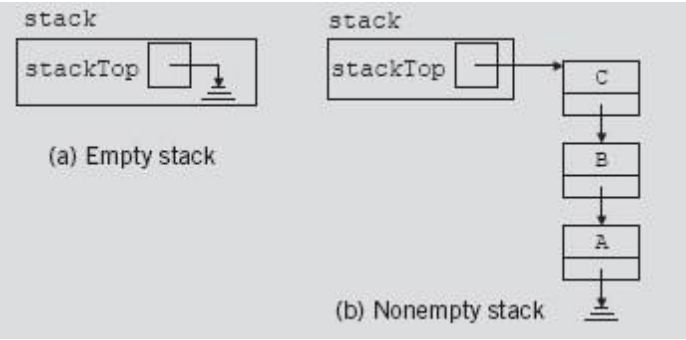


FIGURE 5-10 Empty and nonempty linked stacks

In Figure 5-10(b), the top element of the stack is C; that is, the last element pushed onto the stack is C. Next, we discuss the definitions of the functions to implement the operations of a linked stack.

5.3.1 Default Constructor

The first operation that we consider is the default constructor. The default constructor initializes the stack to an empty state when a stack object is declared. Thus, this function sets stackTop to NULL. The definition of this function is as follows:

```
template <class Type>
linkedStackType<Type>::linkedStackType()
{
```

```
        stackTop = NULL;
}
```

5.3.2 Empty Stack and Full Stack

The operations isEmptyStack and isFullStack are quite straightforward. The stack is empty if stackTop is NULL. Also, because the memory for a stack element is allocated and deallocated dynamically, the stack is never full. (The stack is full only if we run out of memory.) Thus, the function isFullStack always returns the value false. The definitions of the functions to implement these operations are as follows:

```
template <class Type>
bool linkedStackType<Type>::isEmptyStack() const
{
        return(stackTop == NULL);
}                                                   //end isEmptyStack

template <class Type>
bool linkedStackType<Type>::isFullStack() const
{ return false;
}                                                   //end isFullStack
```

Recall that in the linked implementation of stacks, the function isFullStack does not apply because logically the stack is never full. However, you must provide its definition because it is included as an abstract function in the parent class stackADT.

5.3.3 Initialize Stack

The operation initializeStack reinitializes the stack to an empty state. Because the stack might contain some elements and we are using a linked implementation of a stack, we must deallocate the memory occupied by the stack elements and set stackTop to NULL. The definition of this function is as follows:

```
template <class Type>
void linkedStackType<Type>:: initializeStack()
{
        nodeType<Type> *temp;               //pointer to delete the node
        while (stackTop != NULL)            //while there are elements in the stack
```

```
        {
                temp = stackTop;                //set temp to point to the current node
                stackTop = stackTop->link;      //advance stackTop to the next node delete
                temp;                           //deallocate memory occupied by temp
        }
}                                               //end initializeStack
```

Next, we consider the push, top, and pop operations. From Figure 5-10(b), it is clear that the newElement will be added (in the case of push) at the beginning of the linked list pointed to by stackTop. In the case of pop, the node pointed to by stackTop will be removed. In both cases, the value of the pointer stackTop is updated. The operation top returns the info of the node to which stackTop is pointing.

5.3.4 Push

Consider the stack shown in Figure 5-11.

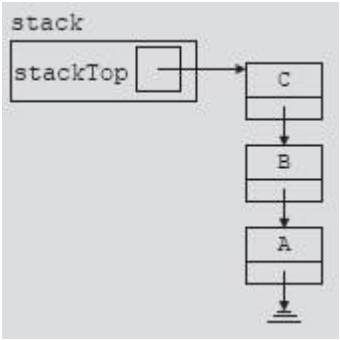


FIGURE 5-11 Stack before the push operation

Figure 5-12 shows the steps of the push operation. (Assume that the new element to be pushed is 'D'.)

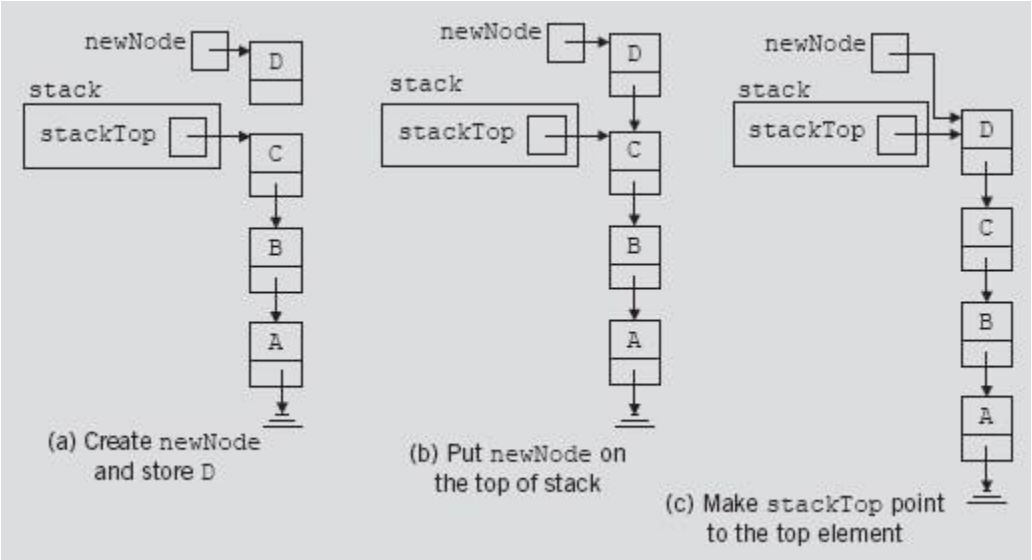


FIGURE 5-12 Push operation

As shown in Figure 5-12, to push 'D' into the stack, first we create a new node and store 'D' into it. Next, we put the new node on top of the stack. Finally, we make stackTop point to the top element of the stack. The definition of the function push is as follows:

```
template <class Type>
void linkedStackType<Type>::push(const Type& newElement)
{
        nodeType<Type> *newNode;            //pointer to create the new node
        newNode = new nodeType<Type>;  //create the node
        newNode->info = newElement;   //store newElement in the node newNode->link =
        stackTop;   //insert newNode before stackTop
```

```
        stackTop = newNode;                 //set stackTop to point to the top node
}                                           //end push
```

We do not need to check whether the stack is full before we push an element onto the stack because in this implementation, logically, the stack is never full.

5.3.5 Return the Top Element

The operation to return the top element of the stack is quite straightforward. Its definition is as follows:

```
template <class Type>
Type linkedStackType<Type>::top() const
{
        assert(stackTop != NULL);      //if stack is empty, terminate the program
        return stackTop->info;         //return the top element
}                                      //end top
```

5.3.6 Pop

Now we consider the pop operation, which removes the top element of the stack. Consider the stack shown in Figure 5-13.

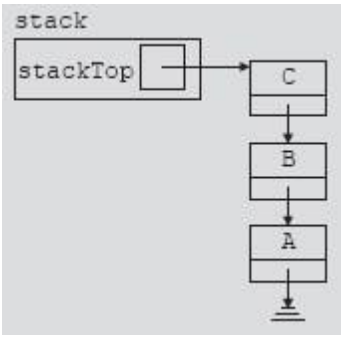


FIGURE 5-13 Stack before the pop operation

Figure 5-14 shows the pop operation.

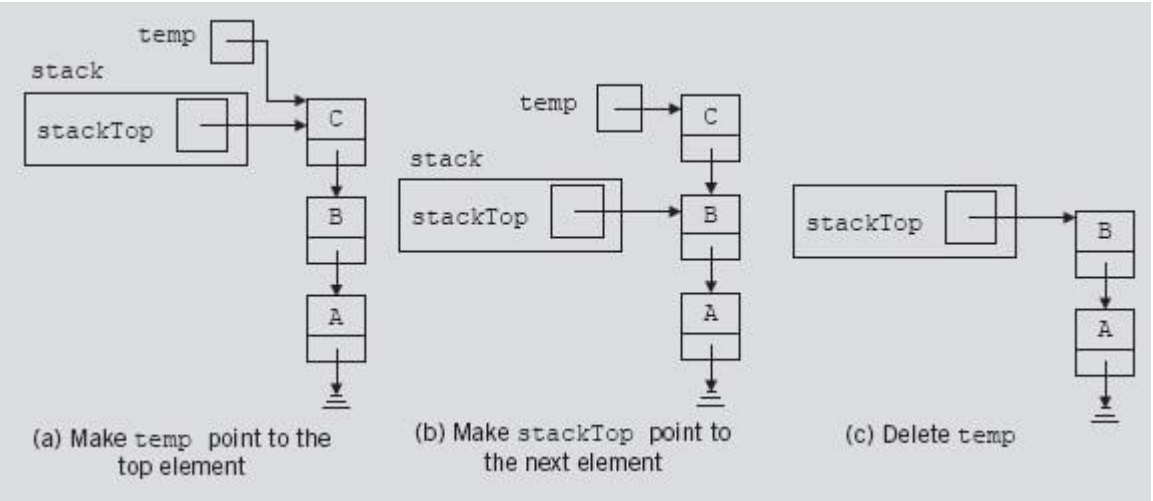


FIGURE 5-14 Pop operation

As shown in Figure 5-14, first we make a pointer temp point to the top of the stack.Next we make stackTop point to the next element of the stack, which will become the top element of the stack. Finally, we delete temp. The definition of the function pop is as follows:

```
template <class Type>
void linkedStackType<Type>::pop()
{
        nodeType<Type> *temp;                    //pointer to deallocate memory if
        (stackTop != NULL)
        {
                temp = stackTop;                 //set temp to point to the top node
                stackTop = stackTop->link;    //advance stackTop to the next node delete temp;
                //delete the top node
        }
        else cout << "Cannot remove from an empty stack." << endl;
}                                                //end pop
```

5.3.7 Copy Stack

The function copyStack makes an identical copy of a stack. Its definition is similar to the definition of copyList for linked lists, given in Chapter 3. The definition of the function copyStack is as follows:

```
template <class Type>
void linkedStackType<Type>::copyStack(const linkedStackType<Type>& otherStack)
{
        nodeType<Type> *newNode, *current, *last;
        if (stackTop != NULL)                    //if stack is nonempty, make it empty
                initializeStack(); if
        (otherStack.stackTop == NULL)
        stackTop = NULL;
        else
        {
```

```
current = otherStack.stackTop;          //set current to point
                                        //to the stack to be copied
                                        //copy the stackTop element of the stack
stackTop = new nodeType<Type>;          //create the node
stackTop->info = current->info;         //copy the info
stackTop->link = NULL;                  //set the link field to NULL
last = stackTop;                        //set last to point to the node
current = current->link;                //set current to point to the
                                        //next node
                                        //copy the remaining stack
while (current != NULL)
{
        newNode = new
        nodeType<Type>; newNode->info
        = current->info; newNode->link =
        NULL; last->link = newNode;
```

```
                last = newNode;
                current = current->link;
            }                                   //end while
        }                                       //end else
}                                               //end copyStack
```

5.3.8 Constructors and Destructors

We have already discussed the default constructor. To complete the implementation of the stack operations, next we give the definitions of the functions to implement the copy constructor and the destructor, and to overload the assignment operator. (These functions are similar to those discussed for linked lists in Chapter 3.)

```
//copy constructor template
<class Type>
linkedStackType<Type>::linkedStackType(const linkedStackType<Type>& otherStack)
{
    stackTop = NULL;
    copyStack(otherStack);
}           //end copy constructor

//destructor template
<class Type>
linkedStackType<Type>::~linkedStackType()
{
    initializeStack();
}           //end destructor
```

5.3.9 Overloading the Assignment Operator (=)

The definition of the function to overload the assignment operator for the class linkedStackType is as follows:

```
template <class Type>
const linkedStackType<Type>& linkedStackType<Type>::operator= (const
linkedStackType<Type>& otherStack)
{
        if (this != &otherStack)                    //avoid self-copy copyStack(otherStack);
        return *this;
}                                                   //end operator=
```

Table 5-2 summarizes the time complexity of the operations to implement a linked stack.

TABLE 5-2: Time complexity of the operations of the class linkedStackType on a stack

Function	Time complexity
`isEmptyStack`	$O(1)$
`isFullStack`	$O(1)$
`initializeStack`	$O(n)$
constructor	$O(1)$
`top`	$O(1)$
`push`	$O(1)$
`pop`	$O(1)$
`copyStack`	$O(n)$
destructor	$O(n)$
copy constructor	$O(n)$
Overloading the assignment operator	$O(n)$

with n elements

The definition of a stack, and the functions to implement the stack operations discussed previously, are generic. Also, as in the case of an array representation of a stack, in the linked representation of a stack, we put the definition of the stack and the functions to implement the stack operations together in a (header) file. A client's program can include this header file via the include statement.

The program in Example 5-3 illustrates how a linkedStack object is used in a program.

EXAMPLE 5-3

We assume that the definition of the class linkedStackType and the functions to implement the stack operations are included in the header file "linkedStack.h".

```
//***********************************************************
// This program tests various operations of a linked stack.
//***********************************************************
#include <iostream> #include
"linkedStack.h" using
namespace std;
void testCopy(linkedStackType<int> OStack); int
```

```
main()
{
        linkedStackType<int> stack; linkedStackType<int>
        otherStack;
        linkedStackType<int> newStack;
```

```
        //Add elements into stack
    stack.push(34); stack.push(43);
    stack.push(27);

        //Use the assignment operator to copy the elements
        //of stack into newStack newStack
    = stack;

    cout << "After the assignment operator, newStack: "<< endl;

        //Output the elements of newStack while
    (!newStack.isEmptyStack())
    {
        cout << newStack.top() << endl; newStack.pop();
    }

        //Use the assignment operator to copy the elements
        //of stack into otherStack otherStack
    = stack;

    cout << "Testing the copy constructor." << endl;
    testCopy(otherStack);
    cout << "After the copy constructor, otherStack: " << endl;

    while (!otherStack.isEmptyStack())
    {
        cout << otherStack.top() << endl; otherStack.pop();
    }
    return 0;
}

//Function to test the copy constructor
```

```
void testCopy(linkedStackType<int> OStack)
{
        cout << "Stack in the function testCopy:" << endl; while
        (!OStack.isEmptyStack())
        {
                cout << OStack.top() << endl; OStack.pop();
        }
}
```

Sample Run:

```
After the assignment operator, newStack: 27
43
```

34

Testing the copy constructor.

Stack in the function testCopy:

27

43

34

After the copy constructor, otherStack:

27

43

34

5.3 Stack as Derived from the class unorderedLinkedList

If we compare the push function of the stack with the insertFirst function discussed for general lists in Chapter 3, we see that the algorithms to implement these operations are similar. A comparison of other functions, such as initializeStack and nitializeList, isEmptyList and isEmptyStack, and so on, suggests that the class linkedStackType can be derived from the class linkedListType. Moreover, the functions pop and isFullStack can be implemented as in the previous section. Note that the class linkedListType is an abstract class and does not implement all the operations. However, the class unorderedLinkedList is derived from the the class linkedListType and provides the definitions of the abstract functions of the class linkedListType. Therefore, we can derive the class linkedStackType from the class unorderedLinkedList.

Next, we define the class linkedStackType that is derived from the class unorderedLinkedList. The definitions of the functions to implement the stack operations are also given.

```
#include <iostream>
#include
"unorderedLinkedList.h" using
namespace std; template <class
Type>
class linkedStackType: public unorderedLinkedList<Type>
{ public:
                void initializeStack(); bool
```

```
                isEmptyStack() const; bool
                isFullStack() const;
                void push(const Type& newItem);
                Type top() const;
                void pop();
};

template <class Type>
void linkedStackType<Type>::initializeStack()
{ unorderedLinkedList<Type>::initializeList();
}

template <class Type>
bool linkedStackType<Type>::isEmptyStack() const
{
```

```
        return unorderedLinkedList<Type>::isEmptyList();
}

template <class Type>
bool linkedStackType<Type>::isFullStack() const
{ return false;
}

template <class Type>
void linkedStackType<Type>::push(const Type& newElement)
{
        unorderedLinkedList<Type>::insertFirst(newElement);
}

template <class Type>
Type linkedStackType<Type>::top() const
{
        return unorderedLinkedList<Type>::front();
}

template <class Type>
void linkedStackType<Type>::pop()
{
        nodeType<Type> *temp;

        temp = first;
        first = first->link; delete
        temp;
}
```

5.4 Application of Stacks: Postfix Expressions Calculator

The usual notation for writing arithmetic expressions (the notation we learned in elementary school) is called infix notation, in which the operator is written between the operands. For example, in the expression a + b, the operator + is between the operands a and b. In infix notation, the operators have precedence. That is, we must evaluate expressions from left to right, and multiplication and division have higher precedence than addition and subtraction. If we want to evaluate the expression in a different order, we must include parentheses. For example, in the expression a + b * c, we first evaluate * using the operands b and c, and then we evaluate + using the operand a and the result of b * c.

In the early 1920s, the Polish mathematician Jan Lukasiewicz discovered that if operators were written before the operands (prefix or Polish notation; for example, + a b), the parentheses can be omitted. In the late 1950s, the Australian philosopher and early computer scientist Charles L. Hamblin proposed a scheme in which the operators follow the operands (postfix operators), resulting in the Reverse Polish notation. This has the advantage that the operators appear in the order required for computation.

For example, the expression:

a + b * c

in a postfix expression is:

a b c * +

The following example shows various infix expressions and their equivalent postfix expressions.

EXAMPLE 5-4

Infix expression	Equivalent postfix expression
$a + b$	$a\ b\ +$
$a + b * c$	$a\ b\ c\ * +$
$a * b + c$	$a\ b\ *\ c +$
$(a + b) * c$	$a\ b + c\ *$
$(a - b) * (c + d)$	$a\ b - c\ d + *$
$(a + b) * (c - d / e) + f$	$a\ b + c\ d\ e / - * f +$

Shortly after Lukasiewicz's discovery, it was realized that postfix notation had important applications in computer science. In fact, many compilers use stacks to first translate infix expressions into some form of postfix notation and then translate this postfix expression into machine code. Postfix expressions can be evaluated using the following algorithm:

Scan the expression from left to right. When an operator is found, back up to get the required number of operands, perform the operation, and continue.

Consider the following postfix expression:

6 3 + 2 * =

Let us evaluate this expression using a stack and the previous algorithm. Figure 5-15 shows how this expression gets evaluated.

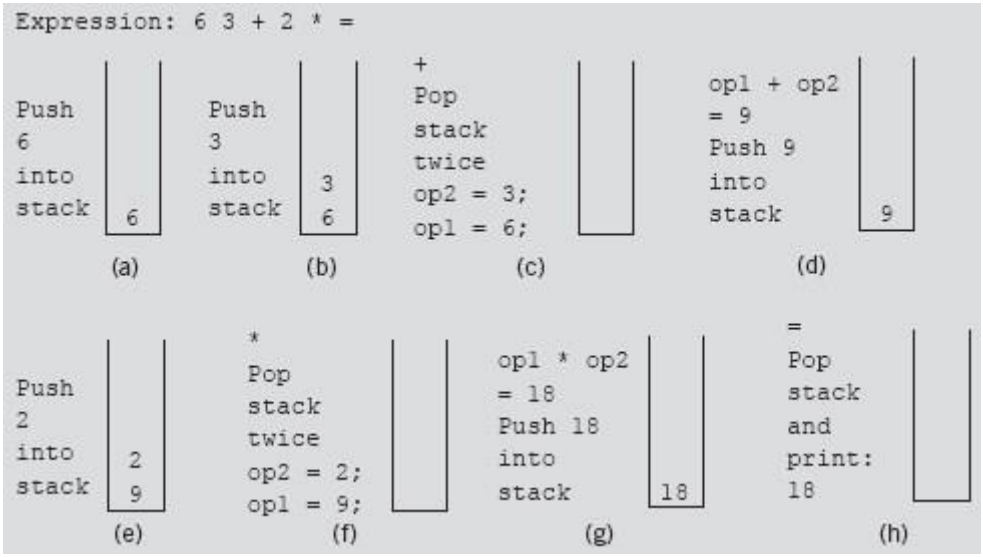


FIGURE 5-15 Evaluating the postfix expression: 6 3 + 2 * =

Read the first symbol, 6, which is a number. Push the number onto the stack; see Figure 5-15(a). Read the next symbol, 3, which is a number. Push the number onto the stack; see Figure 5-15(b). Read the next symbol, +, which is an operator. Because an operator requires two operands to be evaluated, pop the

stack twice; see Figure 5-15(c). Perform the operation and put the result back onto the stack; see Figure 5-15(d).

Read the next symbol, 2, which is a number. Push the number onto the stack; see Figure 5-15(e). Read the next symbol, *, which is an operator. Because an operator requires two operands to be evaluated, pop the stack twice; see Figure 5-15(f). Perform the operation and put the result back onto the stack; see Figure 5-15(g).

Scan the next symbol, =, which is the equal sign, indicating the end of the expression. Therefore, print the result. The result of the expression is in the stack, so pop and print; see Figure 5-15(h).

The value of the expression 6 3 + 2 * = 18.

From this discussion, it is clear that when we read a symbol other than a number, the following cases arise:

1. The symbol we read is one of the following: +, -, *, /, or =.
   a. If the symbol is +, -, *, or /, the symbol is an operator and so we must evaluate it. Because an operator requires two operands, the stack must have at least two elements; otherwise, the expression has an error.
   b. If the symbol is = (an equal sign), the expression ends and we must print the answer. At this step, the stack must contain exactly one element; otherwise, the expression has an error.
2. The symbol we read is something other than +, -, *, /, or ¼. In this case, the expression contains an illegal operator.

It is also clear that when an operand (number) is encountered in an expression, it is pushed onto the stack because the operator comes after the operands.

Consider the following expressions:

i. 7 6 + 3 ; 6 - = ii.

14 + 2 3 * =

iii. 14 2 3 + =

Expression (i) has an illegal operator, expression (ii) does not have enough operands for +, and expression (iii) has too many operands. In the case of expression (iii), when we encounter the equal sign (=), the stack will have two elements and this error cannot be discovered until we are ready to print the value of the expression.

To make the input easier to read, we assume that the postfix expressions are in the following form:

#6 #3 + #2 * =

The symbol # precedes each number in the expression. If the symbol scanned is #, the next input is a number (that is, an operand). If the symbol scanned is not #, it is either an operator (might be illegal) or an equal sign (indicating the end of the expression). Furthermore, we assume that each expression contains only the +, -, *, and / operators.

This program outputs the entire postfix expression together with the answer. If the expression has an error, the expression is discarded. In this case, the program outputs the expression together with an appropriate error message. Because an expression might contain an error, we must clear the stack before processing the next expression. Also, the stack must be initialized; that is, the stack must be empty.

5.3.10 Main Algorithm

Pursuant to the previous discussion, the main algorithm in pseudocode is as follows:

```
Read the first character while
not the end of input data
{
        a. initialize the stack
        b. process the expression
        c. output result
        d. get the next expression
}
```

To simplify the complexity of the function main, we write four functions— evaluateExpression, evaluateOpr, discardExp, and printResult. The function evaluateExpression, if possible, evaluates the expression and leaves the result in the stack. If the postfix expression is error free, the function printResult outputs the result. The function evaluateOpr evaluates an operator, and the function discardExp discards the current expression if there is any error in the expression.

5.3.11 Function evaluateExpression

The function evaluateExpression evaluates each postfix expression. Each expression ends with the symbol =. The general algorithm in pseudocode is as follows:

```
while (ch is not = '=')                    //process each expression = marks the end of an expression
{
        switch (ch)
```

```
{
        case '#':
                read a number output the
                number; push the number onto
                the stack; break;

        default:
                assume that ch is an operation
                evaluate the operation;
}                                       //end switch

if no error was found, then
{
        read next ch;
        output ch;
```

```
        }
        else
        Discard the expression
}                                                   //end while
```

From this algorithm, it follows that this method has five parameters—a parameter to access the input file, a parameter to access the output file, a parameter to access the stack, a parameter to pass a character of the expression, and a parameter to indicate whether there is an error in the expression. The definition of this function is as follows:

```
void evaluateExpression(ifstream& inpF, ofstream& outF, stackType<double>&
stack,
char& ch, bool& isExpOk)
{
        double num;
        outF << ch;

        while (ch != '=')
        {
                switch (ch)
                {
                        case '#':
                                inpF >> num; outF
                                << num << " ";
                                if (!stack.isFullStack())
                                stack.push(num); else
                                {
                                        cout << "Stack overflow. "<< "Program terminates!" << endl;
                                        exit(0);                        //terminate the program
                                }
                                break; default:
                                evaluateOpr(outF, stack, ch, isExpOk);
                }                                                       //end switch
```

```
            if (isExpOk)                                    //if no error
            { inpF >> ch; outF
                    << ch; if
                    (ch != '#')
                            outF << " ";
            }
            else discardExp(inpF, outF, ch);
        }                                                   //end while (!= '=')
}                                                           //end evaluateExpression
```

Note that the funtion exit terminates the program.

5.3.12 Function evaluateOpr

This function (if possible) evaluates an expression. Two operands are needed to evaluate an operation and operands are saved in the stack. Therefore, the stack must contain at least two numbers. If the stack contains fewer than two numbers, the expression has an error. In this case, the entire expression is discarded and an appropriate message is printed. This function also checks for any illegal operations. In pseudocode, this function is as follows:

```
if stack is empty
{
        error in the expression set
        expressionOk to false
}
else
{
        retrieve the top element of stack into op2 pop
        stack
        if stack is empty
        {
                error in the expression
                set expressionOk to false
        }
        else
        {
                retrieve the top element of stack into op1 pop
                stack
                //If the operation is legal, perform the operation and
                //push the result onto the stack; //otherwise, report
                the error. switch (ch)
                {
                        case '+': //add the operands: op1 +
                                op2 stack.push(op1 + op2);
```

```
                    break;

                case '-': //subtract the operands: op1 - op2
                stack.push(op1 - op2); break; case '*':
                //multiply the operands: op1 * op2
                stack.push(op1 * op2); break; case '/': //If
                (op2 != 0), op1 / op2 stack.push(op1 /
                op2); break; otherwise operation is illegal
                {
                    output an appropriate message;
                    set expressionOk to false
                }
            } //end switch
}
```

Following this pseudocode, the definition of the function evaluateOpr is as follows:

```
void evaluateOpr(ofstream& out, stackType<double>& stack, char& ch, bool& isExpOk)
{
	double op1, op2;
	if (stack.isEmptyStack())
	{
		out << " (Not enough operands)";
		isExpOk = false;
	}
	else
	{
		op2 = stack.top(); stack.pop();
		if (stack.isEmptyStack())
		{
			out << " (Not enough operands)";
			isExpOk = false;
		}
		else
		{
			op1 = stack.top(); stack.pop();
			switch (ch)
			{
				case '+':
					stack.push(op1 + op2);
				break; case '-':
					stack.push(op1 - op2); break;
				case '*':
					stack.push(op1 * op2);
				break; case '/':
					if (op2 != 0) stack.push(op1 /
						op2);
```

```
                    else
                    {
                        out << " (Division by 0)"; isExpOk
                        = false;
                    }
                    break;

                default: out << " (Illegal
                    operator)"; isExpOk =
                    false;
            }                                   //end switch
        }                                       //end else
    }                                           //end else
}                                               //end evaluateOpr
```

5.3.13 Function discardExp

This function is called whenever an error is discovered in the expression. It reads and writes the input data only until the input is '=', the end of the expression. The definition of this function is as follows:

```
void discardExp(ifstream& in, ofstream& out, char& ch)
{
        while (ch != '=')
        { in.get(ch);
                out << ch;
        }
}                       //end discardExp
```

5.3.14 Function printResult

If the postfix expression contains no errors, the function printResult prints the result; otherwise, it outputs an appropriate message. The result of the expression is in the stack and the output is sent to a file. Therefore, this function must have access to the stack and the output file. Suppose that no errors were encountered by the method evaluateExpression. If the stack has only one element, the expression is error free and the top element of the stack is printed. If either the stack is empty or it has more than one element, there is an error in the postfix expression. In this case, this method outputs an appropriate error message. The definition of this method is as follows:

```
void printResult(ofstream& outF, stackType<double>& stack, bool isExpOk)
{ double result;
        if (isExpOk)                                    //if no error, print the result
        {
                if (!stack.isEmptyStack())
                { result = stack.top();
                        stack.pop();
```

```
                    if (stack.isEmptyStack()) outF << result << endl; else
                outF << " (Error: Too many operands)" << endl; } //end if
                else outF << " (Error in the expression)" << endl;
        }
        else outF << " (Error in the expression)" << endl;
                outF << "_____"<< endl << endl;
}                                                       //end printResult
```

5.5 Program Listing

```
//*********************************************************** //
This program uses a stack to evaluate postfix expressions.
//***********************************************************
#include <iostream>
#include <iomanip>
```

```
#include <fstream>
#include "mystack.h"
using namespace std;
void evaluateExpression(ifstream& inpF, ofstream& outF, stackType<double>& stack
        , char& ch, bool& isExpOk); void
evaluateOpr(ofstream& out, stackType<double>& stack,
char& ch, bool& isExpOk);
void discardExp(ifstream& in, ofstream& out, char& ch);
void printResult(ofstream& outF, stackType<double>& stack, bool isExpOk); int
main()
{
        bool expressionOk;
        char ch;
        stackType<double> stack(100);
        ifstream infile;
        ofstream outfile;
        infile.open("RpnData.txt");
        if (!infile)
        {
                cout << "Cannot open the input file. "<< "Program terminates!" << endl; return
                1;
        }

        outfile.open("RpnOutput.txt");

        outfile << fixed << showpoint;
        outfile << setprecision(2);

        infile >> ch; while
        (infile)
        {
                stack.initializeStack();
                expressionOk = true; outfile
```

```
                << ch;

                evaluateExpression(infile, outfile, stack, ch,expressionOk); printResult(outfile,
                stack, expressionOk);
                infile >> ch; //begin processing the next expression
        } //end while

        infile.close();
        outfile.close();

        return 0;
} //end main
```

```
//Place the definitions of the function evaluateExpression,
//evaluateOpr, discardExp, and printResult as described //previously
here.
```

Sample Run:

Input File

```
#35 #27 + #3 * =
#26 #28 + #32 #2 ; - #5 / =
#23 #30 #15 * / =
#2 #3 #4 + =
#20 #29 #9 * ; =
#25 #23 - + =
#34 #24 #12 #7 / * + #23 - =
```

Output

#35.00 #27.00 + #3.00 * = 186.00

#26.00 #28.00 + #32.00 #2.00 ; (Illegal operator) - #5 / = (Error in the expression)

#23.00 #30.00 #15.00 * / = 0.05

#2.00 #3.00 #4.00 + = (Error: Too many operands)

#20.00 #29.00 #9.00 * ; (Illegal operator) = (Error in the expression)

#25.00 #23.00 - + (Not enough operands) = (Error in the expression)

#34.00 #24.00 #12.00 #7.00 / * + #23.00 - = 52.14

5.6 Removing Recursion: Nonrecursive Algorithm to Print a Linked List Backward

In this section, you will learn how a stack can be used to design a nonrecursive algorithm to print a linked list backward. Consider the linked list shown in Figure 5-16.

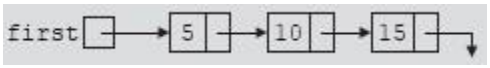


FIGURE 5-16 Linked list

To print the list backward, first we need to get to the last node of the list, which we can do by traversing the linked list starting at the first node. However, once we are at the last node, how do we get back to the previous node, especially given that links go in only one direction? You can again traverse the linked list with the appropriate loop termination condition, but this approach might waste a considerable amount of computer time, especially if the list is very large. Moreover, if we do this for every node in the list, the program might execute very slowly. Next, we show how to use a stack effectively to print the list backward.

After printing the info of a particular node, we need to move to the node immediately behind this node. For example, after printing 15, we need to move to the node with info 10. Thus, while initially traversing the list to move to the last node, we must save a pointer to each node. For example, for the list in Figure 5-16, we must save a pointer to each of the nodes with info 5 and 10. After printing 15, we go back to the node with info 10; after printing 10, we go back to the node with info 5. From this, it follows that we must save pointers to each node in a stack, so as to implement the Last In First Out principle.

Because the number of nodes in a linked list is usually not known, we will use the linked implementation of a stack. Suppose that stack is an object of type linkedListType, and current is a pointer of the same type as the pointer first. Consider the following statements:

```
current = first;      //Line 1 while (current != NULL)     //Line 2
{          //Line 3  stack.push(current);       //Line 4 current =
           current->link;    //Line 5
}                                                          //Line 6
```

After the statement in Line 1 executes, current points to the first node. (See Figure 5-17)

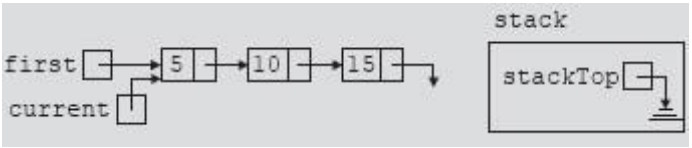


FIGURE 5-17 List after the statement current = first; executes

Because current is not NULL, the statements in Lines 4 and 5 execute. (See Figure 5-18.)

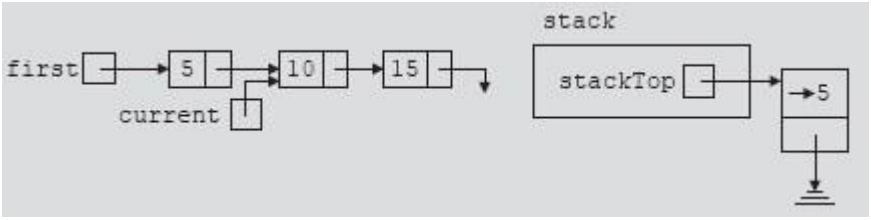


FIGURE 5-18 List and stack after the statements stack.push(current); and current = current->link; execute

Because current is not NULL, statements in Lines 4 and 5 execute. In fact, statements in Lines 4 and 5 execute until current beomes NULL. When current is NULL, Figure 5-19 results.

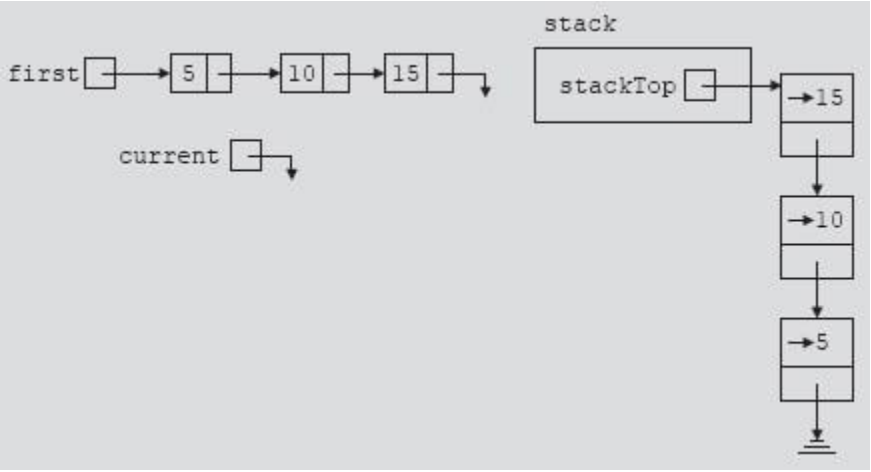

stack
first
5
10
15
stackTop
15
current
10
5

FIGURE 5-19 List and stack after the while statement executes

After the statement in Line 4 executes, the loop condition, in Line 2, is evaluated again. Because current is NULL, the loop condition evaluates to false and the while loop, in Line 2, terminates. From Figure 5-19, it follows that a pointer to each node in the linked list is saved in the stack. The top element of the stack contains a pointer to the last node in the list, and so on. Let us now execute the following statements:

```
while (!stack.isEmptyStack())                    //Line 7
{                                                //Line 8 current
        = stack.top();              //Line 9
        stack.pop();                             //Line 10 cout
        << current->info << " ";            //Line 11
}                                                //Line 12
```

The loop condition in Line 7 evaluates to true because the stack is nonempty. Therefore, the statements in Lines 9, 10, and 11 execute. After the statement in Line 9 executes, current points to the last node. The statement in Line 10 removes the top element of the stack; see Figure 5-20.

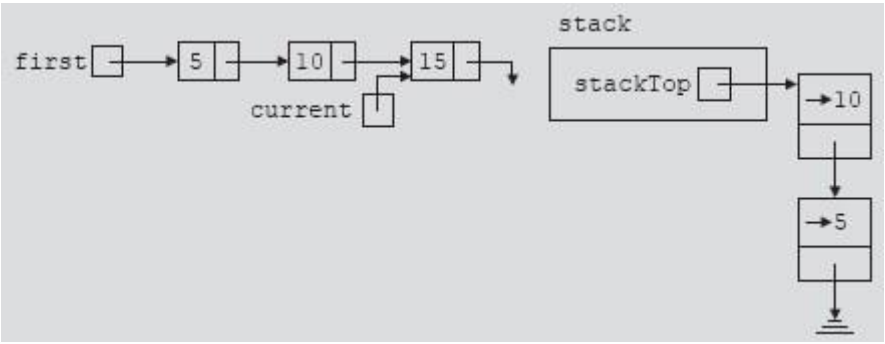


FIGURE 5-20 List and stack after the statements current = stack.top(); and stack.pop(); Execute

The statement in Line 11 outputs current->info, which is 15.

Because stack is nonempty, the body of the while loop executes again. In fact, for the linked list in Figure 5-20, the body of the while loop executes two more times; the first time it prints 10, and the second time it prints 5. Ater printing 5, the stack becomes empty and the while loop terminates. It follows that the while loop in Line 7 produces the following output: 15 10 5

5.7 STL class stack

The previous sections discussed the data structure stack in detail. Because a stack is an important data structure, the Standard Template Library (STL) provides a class to implement a stack in a program. The name of the class defining a stack is stack; the name of the header file containing the definition of the class stack is stack. The implementation of the class stack provided by the STL is similar to the one described in this chapter. Table 5-3 defines the various operations supported by the stack container class.

TABLE 5-3: Operations on a stack object

Operation	Effect
`size`	Returns the actual number of elements in the stack.
`empty`	Returns `true` if the stack is empty, and `false` otherwise.
`push(item)`	Inserts a copy of item into the stack.
`top`	Returns the top element of the stack, but does not remove the top element from the stack. This operation is implemented as a value-returning function.
`pop`	Removes the top element of the stack.

In addition to the operations size, empty, push, top, and pop, the stack container class provides relational operators to compare two stacks. For example, the relational operator == can be used to determine whether two stacks are identical.

The program in Example 5-5 illustrates how to use the stack container class.

EXAMPLE 5-5

```
//***********************************************************
// This program illustrates how to use the STL class stack in //
a program.
//***********************************************************
#include <iostream>                                          //Line 1 #include
<stack>                                                      //Line 2 using
namespace std;                                               //Line 3

int main()                                                          //Line 4
{                                                                   //Line 5 stack<int>
        intStack;                                                   //Line 6 intStack.push(16);
                                                                    //Line 7 intStack.push(8);
                                                                    //Line 8 intStack.push(20);
                                                                    //Line 9
        intStack.push(3);                                           //Line 10

        cout << "Line 11: The top element of intStack: "
```

```
        << intStack.top() << endl;                                  //Line 11

intStack.pop();                                                     //Line 12

cout << "Line 13: After the pop operation, the "
        << " top element of intStack: "
        << intStack.top() << endl;                                  //Line 13

cout << "Line 14: intStack elements: ";                             //Line 14
```

```
    while (!intStack.empty())                                          //Line 15
    {       //Line 16 cout << intStack.top() << " ";      //Line 17 intStack.pop();       //Line
             18
    }                                                                  //Line 19

    cout << endl;                                                      //Line 20 return 0;
                                                                       //Line 21
}                                                                      //Line 22
```

Sample Run:

Line 11: The top element of intStack: 3

Line 13: After the pop operation, the top element of intStack: 20 Line

14: intStack elements: 20 8 16

The preceding output is self-explanatory. The details are left as an exercise for you.

Key Terms

1. A stack is a data structure in which the items are added and deleted from one end only.
2. A stack is a Last In First Out (LIFO) data structure.
3. The basic operations on a stack are as follows: Push an item onto the stack, pop an item from the stack, retrieve the top element of the stack, initialize the stack, check whether the stack is empty, and check whether the stack is full.
4. A stack can be implemented as an array or a linked list.
5. The middle elements of a stack should not be accessed directly.
6. Stacks are restricted versions of arrays and linked lists.
7. Postfix notation does not require the use of parentheses to enforce operator precedence.
8. In postfix notation, the operators are written after the operands.
9. Postfix expressions are evaluated according to the following rules:
    a. Scan the expression from left to right.
    b. If an operator is found, back up to get the required number of operands, evaluate the operator, and continue.
10. The STL class stack can be used to implement a stack in a program.

Review Exercise

1. Consider the following statements:

```
stackType<int> stack;
int x, y;
```

Show what is output by the following segment of code:

```
x = 4; y = 0; stack.push(7);
stack.push(x);
stack.push(x + 5); y =
stack.top(); stack.pop();
stack.push(x + y);
stack.push(y - 2);
stack.push(3); x =
stack.top(); stack.pop();
cout << "x = " << x <<
endl; cout << "y = " << y
<< endl;
while (!stack.isEmptyStack())
{
        cout << stack.top() << endl; stack.pop();
}
```

2. Consider the following statements:

```
stackType<int> stack;
int x;
```

Suppose that the input is:

```
14 45 34 23 10 5 -999
```

Show what is output by the following segment of code:

```
stack.push(5);
cin >> x; while
(x != -999)
{
```

```
        if (x % 2 == 0)
        {
                if (!stack.isFullStack()) stack.push(x);
        }
        else
                cout << "x = " << x << endl;
        cin >> x;
}
cout << "Stack Elements: ";
```

```
while (!stack.isEmptyStack())
{
        cout << " " << stack.top(); stack.pop();
}
cout << endl;
```

3. Evaluate the following postfix expressions:

a. 8 2 + 3 * 16 4 / - =

b. 12 25 5 1 / / * 8 7 + - =

c. 70 14 4 5 15 3 / * - - / 6 + =

d. 3 5 6 * + 13 - 18 2 / + =

4. Convert the following infix expressions to postfix notations:

a. (A + B) * (C + D) - E

b. A - (B + C) * D + E / F

c. ((A + B) / (C - D) + E) * F - G

d. A + B * (C + D) - E / F * G + H

5. Write the equivalent infix expression for the following postfix expressions: a.

A B * C +

b. A B + C D - *

c. A B – C – D *

6. What is the output of the following program?

```
#include <iostream>
#include <string> #include
"myStack.h"
using namespace std;

template <class Type>
void mystery(stackType<Type>& s, stackType<Type>& t); int
main()
{
```

```
stackType<string> s1; stackType<string>
s2;

string list[] = {"Winter", "Spring", "Summer", "Fall","Cold", "Warm", "Hot"};
for (int i = 0; i < 7; i++) s1.push(list[i]);
mystery(s1, s2);

while (!s2.isEmptyStack())
{ cout << s2.top() << " ";
        s2.pop();
}
cout << endl;
```

```
}

template <class Type>
void mystery(stackType<Type>& s, stackType<Type>& t)
{
        while (!s.isEmptyStack())
        {
                t.push(s.top());
                s.pop();
        }
}
```

7. What is the effect of the following statements? If a statement is invalid, explain why it is invalid. The classes stackADT, stackType, and linkedStackType are as defined in this chapter. a. stackADT<int> newStack;

b. stackType<double> sales(-10);

c. stackType<string> names;

d. linkedStackType<int> numStack(50);

8. What is the output of the following program?

```
#include <iostream>
#include <string> #include
"myStack.h" using
namespace std;
void mystery(stackType<int>& s, stackType<int>& t); int
main()
{
        int list[] = {5, 10, 15, 20, 25};

        stackType<int> s1; stackType<int>
        s2;

        for (int i = 0; i < 5; i++)
```

```
            s1.push(list[i]);
        mystery(s1, s2);
        while (!s2.isEmptyStack())
        { cout << s2.top() << " ";
            s2.pop();
        }
        cout << endl;
}

void mystery(stackType<int>& s, stackType<int>& t)
{
        while (!s.isEmptyStack())
        {
```

```
            t.push(2 * s.top());
            s.pop();
        }
}
```

9. What is the output of the following program segment?

```
linkedStackType<int> myStack;
myStack.push(10);
myStack.push(20);
myStack.pop();
cout << myStack.top() << endl; myStack.push(25);
myStack.push(2 * myStack.top()); myStack.push(-10);
myStack.pop();

linkedStackType<int> tempStack;
tempStack = myStack;
while (!tempStack.isEmptyStack())
{
        cout << tempStack.top() << " "; tempStack.pop();
}

cout << endl;

cout << myStack.top() << endl;
```

10. Write the definition of the function template printListReverse that uses a stack to print a linked list in reverse order. Assume that this function is a member of the class linkedListType, designed in Chapter 3.

11. Write the definition of the function template second that takes as a parameter a stack object and returns the second element of the stack. The original stack remains unchanged.

12. Draw the UML class diagram of the class linkedStackType.

13. Write the definition of the function template clear that takes as a parameter a stack object of the type stack (STL class) and removes all the elements from the stack.

6 TOPIC 6: QUEUES



LEARNING OUTCOMES:

After studying this topic you should be able to:

- Learn about queues
- Examine various queue operations
- Learn how to implement a queue as an array
- Learn how to implement a queue as a linked list
- Discover queue applications
- Become aware of the STL class queue

This chapter discusses another important data structure, called a queue. The notion of a queue in computer science is the same as the notion of the queues to which you are accustomed in everyday life. There are queues of customers in a bank or in a grocery store and queues of cars waiting to pass through a tollbooth. Similarly, because a computer can send a print request faster than a printer can print, a queue of documents is often waiting to be printed at a printer. The general rule to process elements in a queue is that the customer at the front of the queue is served next and that when a new customer arrives, he or she stands at the end of the queue. That is, a queue is a First In First Out data structure.

Queues have numerous applications in computer science. Whenever a system is modelled on the First In First Out principle, queues are used. At the end of this section, we will discuss one of the most widely used applications of queues, computer simulation. First, however, we need to develop the tools necessary to implement a queue. The next few sections discuss how to design classes to implement queues as an ADT.

A queue is a set of elements of the same type in which the elements are added at one end, called the back or rear, and deleted from the other end, called the front. For example, consider a line of customers in a bank, wherein the customers are waiting to withdraw/deposit money or to conduct some other business. Each new customer gets in the line at the rear. Whenever a teller is ready for a new customer, the customer at the front of the line is served.

The rear of the queue is accessed whenever a new element is added to the queue, and the front of the queue is accessed whenever an element is deleted from the queue. As in a stack, the middle elements of the queue are inaccessible, even if the queue elements are stored in an array.

Queue: A data structure in which the elements are added at one end, called the rear, and deleted from the other end, called the front; a First In First Out (FIFO) data structure.

6.1 Queue Operations

From the definition of queues, we see that the two key operations are add and delete. We call the add operation addQueue and the delete operation deleteQueue. Because elements can be neither deleted from an empty queue nor added to a full queue, we need two more operations to successfully implement the addQueue and deleteQueue operations: isEmptyQueue (checks whether the queue is empty) and isFullQueue (checks whether a queue is full).

We also need an operation, initializeQueue, to initialize the queue to an empty state. Moreover, to retrieve the first and last elements of the queue, we include the operations front and back as described in the following list. Some of the queue operations are as follows:

- initializeQueue—Initializes the queue to an empty state.
- isEmptyQueue—Determines whether the queue is empty. If the queue is empty, it returns the value true; otherwise, it returns the value false.
- isFullQueue—Determines whether the queue is full. If the queue is full, it returns the value true; otherwise, it returns the value false.
- front—Returns the front, that is, the first element of the queue. Prior to this operation, the queue must exist and must not be empty.
- back—Returns the last element of the queue. Prior to this operation, the queue must exist and must not be empty.
- addQueue—Adds a new element to the rear of the queue. Prior to this operation, the queue must exist and must not be full.
- deleteQueue—Removes the front element from the queue. Prior to this operation, the queue must exist and must not be empty.

As in the case of a stack, a queue can be stored in an array or in a linked structure. We will consider both implementations. Because elements are added at one end and removed from the other end, we need two pointers to keep track of the front and rear of the queue, called queueFront and queueRear.

The following abstract class queueADT defines these operations as an ADT:

```
//**********************************************************
// This class specifies the basic operations on a queue.
//**********************************************************  template
<class Type>
class queueADT
{ public:
        virtual bool isEmptyQueue() const = 0;
        //Function to determine whether the queue is empty.
        //Postcondition: Returns true if the queue is empty,
        // otherwise returns false.

        virtual bool isFullQueue() const = 0;
        //Function to determine whether the queue is full.
```

```
//Postcondition: Returns true if the queue is full, //
otherwise returns false.

virtual void initializeQueue() = 0;
//Function to Initialize the queue to an empty state.
//Postcondition: The queue is empty.

virtual Type front() const = 0;
//Function to return the first element of the queue.
//Precondition: The queue exists and is not empty.
//Postcondition: If the queue is empty, the program //
terminates; otherwise, the first element of the queue //
is returned.
```

```
        virtual Type back() const = 0;
        //Function to return the last element of the queue. //Precondition:
        The queue exists and is not empty.
        //Postcondition: If the queue is empty, the program //
        terminates; otherwise, the last element of the queue //
        is returned.

        virtual void addQueue(const Type& queueElement) = 0; //Function
        to add queueElement to the queue.
        //Precondition: The queue exists and is not full.
        //Postcondition: The queue is changed and queueElement is //
        added to the queue.

        virtual void deleteQueue() = 0;
        //Function to remove the first element of the queue.
        //Precondition: The queue exists and is not empty.
        //Postcondition: The queue is changed and the first element //
        is removed from the queue.
};
```

We leave it as an exercise for you to draw the UML diagram of the class queueADT.

6.2 Implementation of Queues as Arrays

Before giving the definition of the class to implement a queue as an ADT, we need to decide how many member variables are needed to implement the queue. Of course, we need an array to store the queue elements, the variables queueFront and queueRear to keep track of the first and last elements of the queue, and the variable maxQueueSize to specify the maximum size of the queue. Thus, we need at least four member variables.

Before writing the algorithms to implement the queue operations, we need to decide how to use queueFront and queueRear to access the queue elements. How do queueFront and queueRear indicate that the queue is empty or full? Suppose that queueFront gives the index of the first element of the queue, and queueRear gives the index of the last element of the queue. To add an element to the queue, first we

advance queueRear to the next array position and then add the element to the position that queueRear is pointing to. To delete an element from the queue, first we retrieve the element that queueFront is pointing to and then advance queueFront to the next element of the queue. Thus, queueFront changes after each deleteQueue operation and queueRear changes after each addQueue operation.

Let us see what happens when queueFront changes after a deleteQueue operation and queueRear changes after an addQueue operation. Assume that the array to hold the queue elements is of size 100.

Initially, the queue is empty. After the operation:

```
addQueue(Queue,'A');
```

the array is as shown in Figure 6-1.

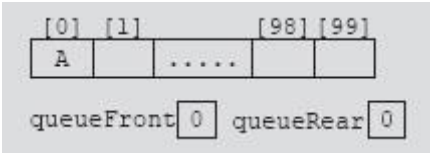


FIGURE 6-1 Queue after the first addQueue operation

After two more addQueue operations:

addQueue(Queue,'B'); addQueue(Queue,'C');

the array is as shown in Figure 6-2.

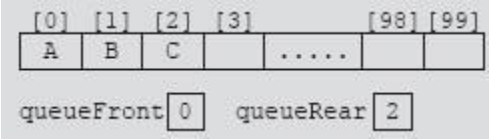


FIGURE 6-2 Queue after two more addQueue operations

Now consider the deleteQueue operation:

deleteQueue();

After this operation, the array containing the queue is as shown in Figure 6-3.

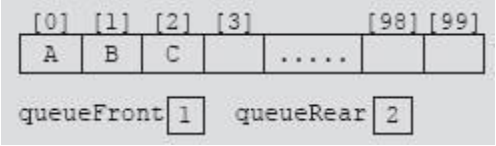


FIGURE 6-3 Queue after the deleteQueue operation

Will this queue design work? Suppose A stands for adding (that is, addQueue) an element to the queue, and D stands for deleting (that is, deleteQueue) an element from the queue. Consider the following sequence of operations:

AAADADADADADADADA...

This sequence of operations would eventually set the index queueRear to point to the last array position, giving the impression that the queue is full. However, the queue has only two or three elements and the front of the array is empty. (See Figure 6-4.)

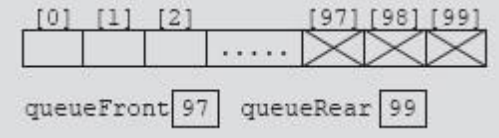


FIGURE 6-4 Queue after the sequence of operations AAADADADADADA...

One solution to this problem is that when the queue overflows to the rear (that is, queueRear points to the last array position), we can check the value of the index queueFront. If the value of queueFront indicates that there is room in the front of the array, then when queueRear gets to the last array position, we can slide all of the queue elements toward the first array position. This solution is good if the queue size is very small; otherwise, the program might execute more slowly.

Another solution to this problem is to assume that the array is circular—that is, the first array position immediately follows the last array position. (See Figure 6-5.)

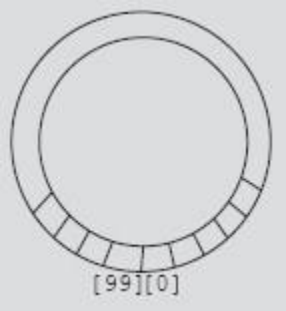


FIGURE 6-5 Circular queue

We will consider the array containing the queue to be circular, although we will draw the figures of the array holding the queue elements as before.

Suppose that we have the queue as shown in Figure 6-6(a).

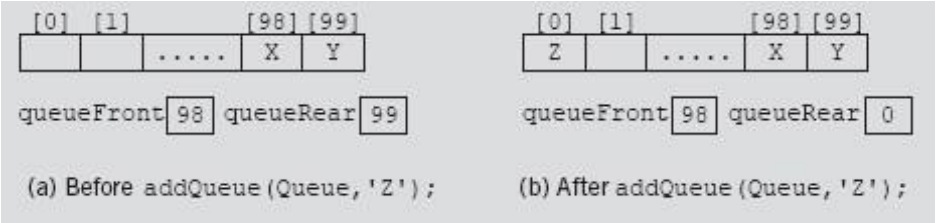


FIGURE 6-6 Queue before and after the add operation

After the operation addQueue(Queue,'Z');, the queue is as shown in Figure 6-6(b).

Because the array containing the queue is circular, we can use the following statement to advance queueRear (queueFront) to the next array position:

queueRear = (queueRear + 1) % maxQueueSize;

If queueRear < maxQueueSize - 1, then queueRear + 1 <= maxQueueSize - 1, so (queueRear + 1) % maxQueueSize = queueRear + 1. If queueRear == maxQueueSize - 1 (that is, queueRear points to the last array position), queueRear + 1 == maxQueueSize, so (queueRear + 1) % maxQueueSize = 0. In this case, queueRear will be set to 0, which is the first array position.

This queue design seems to work well. Before we write the algorithms to implement the queue operations,

consider the following two cases.

Case 1: Suppose that after certain operations, the array containing the queue is as shown in Figure 6-7(a).

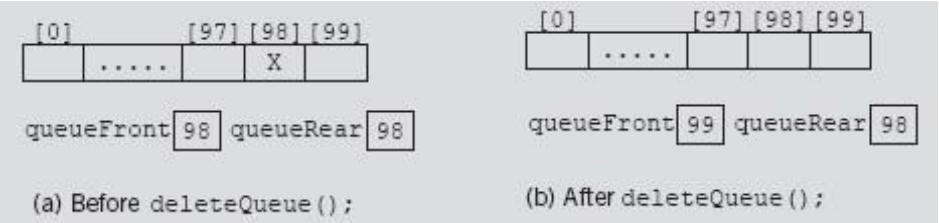


FIGURE 6-7 Queue before and after the delete operation

After the operation deleteQueue();, the resulting array is as shown in Figure 6-7(b).

Case 2: Let us now consider the queue shown in Figure 6-8(a).

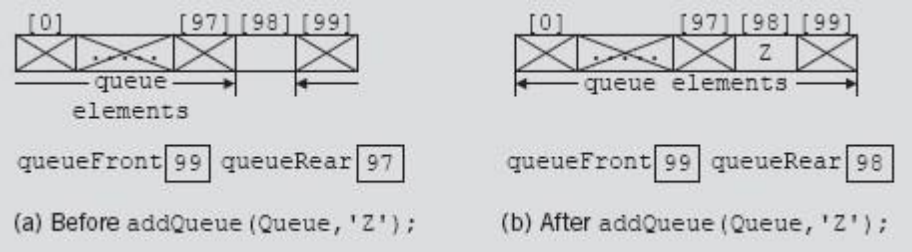


FIGURE 6-8 Queue before and after the add operation

After the operation addQueue(Queue,'Z');, the resulting array is as shown in Figure 6-8(b).

The arrays in Figures 6-7(b) and 6-8(b) have identical values for queueFront and queueRear. However, the resulting array in Figure 6-7(b) represents an empty queue, whereas the resulting array in Figure 68(b) represents a full queue. This latest queue design has brought up another problem of distinguishing between an empty and a full queue.

This problem has several solutions. One solution is to keep a count. In addition to the member variables queueFront and queueRear, we need another variable, count, to implement the queue. The value of count is incremented whenever a new element is added to the queue, and it is decremented whenever an element is removed from the queue. In this case, the function initializeQueue initializes count to 0. This solution is very useful if the user of the queue frequently needs to know the number of elements in the queue.

Another solution is to let queueFront indicate the index of the array position preceding the first element of the queue, rather than the index of the (actual) first element itself. In this case, assuming queueRear still indicates the index of the last element in the queue, the queue is empty if queueFront == queueRear. In this solution, the slot indicated by the index queueFront (that is, the slot preceding the first true element) is reserved. The queue will be full if the next available space is the special reserved slot indicated by queueFront. Finally, because the array position indicated by queueFront is to be kept empty, if the array size is, say, 100, then 99 elements can be stored in the queue. (See Figure 6-9.)

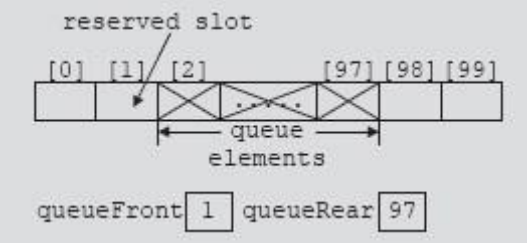


FIGURE 6-9 Array to store the queue elements with a reserved slot

Let us implement the queue using the first solution. That is, we use the variable count to indicate whether the queue is empty or full.

The following class implements the functions of the abstract class queueADT. Because arrays can be allocated dynamically, we will leave it for the user to specify the size of the array to implement the queue. The default size of the array is 100.

```
//*************************************************************
// This class specifies the basic operation on a queue as an array.
//*************************************************************  template
<class Type>
class queueType: public queueADT<Type>
{ public:
        const queueType<Type>& operator=(const queueType<Type>&); //Overload
        the assignment operator.

        bool isEmptyQueue() const;
        //Function to determine whether the queue is empty.
        //Postcondition: Returns true if the queue is empty,
        // otherwise returns false.

        bool isFullQueue() const;
        //Function to determine whether the queue is full.
        //Postcondition: Returns true if the queue is full, //
        otherwise returns false.

        void initializeQueue();
        //Function to initialize the queue to an empty state.
        //Postcondition: The queue is empty.

        Type front() const;
        //Function to return the first element of the queue. //Precondition:
        The queue exists and is not empty.
        //Postcondition: If the queue is empty, the program
        // terminates; otherwise, the first element of the //
        queue is returned.

        Type back() const;
        //Function to return the last element of the queue. //Precondition:
        The queue exists and is not empty.
```

```
//Postcondition: If the queue is empty, the program //
terminates; otherwise, the last element of the queue //
is returned.

void addQueue(const Type& queueElement);
//Function to add queueElement to the queue.
//Precondition: The queue exists and is not full.
//Postcondition: The queue is changed and queueElement is //
added to the queue.

void deleteQueue();
//Function to remove the first element of the queue.
//Precondition: The queue exists and is not empty.
//Postcondition: The queue is changed and the first element //
is removed from the queue.
```

```
        queueType(int queueSize = 100);
        //Constructor

        queueType(const queueType<Type>& otherQueue);
        //Copy constructor

        ~queueType();
        //Destructor

    private:
        int maxQueueSize;          //variable to store the maximum queue size
        int count;                 //variable to store the number of elements in the queue
        int queueFront;            //variable to point to the first element of the queue
        int queueRear;             //variable to point to the last element of the queue
        Type *list;                //pointer to the array that holds the queue elements
};
```

We leave the UML diagram of the class queueType as an exercise for you. (See Exercise 15 at the end of this chapter.)

Next, we consider the implementation of the queue operations.

6.2.1 Empty Queue and Full Queue

As discussed earlier, the queue is empty if count == 0, and the queue is full if count == maxQueueSize. So the functions to implement these operations are as follows:

```
template <class Type>
bool queueType<Type>::isEmptyQueue() const
{
    return (count == 0);
} //end isEmptyQueue
```

```
template <class Type>
bool queueType<Type>::isFullQueue() const
{
        return (count == maxQueueSize);
} //end isFullQueue
```

6.2.2 Initialize Queue

This operation initializes a queue to an empty state. The first element is added at the first array position. Therefore, we initialize queueFront to 0, queueRear to maxQueueSize - 1, and count to 0. See Figure 610.

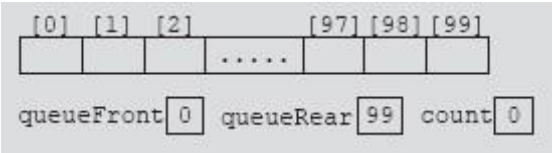


FIGURE 6-10 Empty queue

The definition of the function initializeQueue is as follows:

```
template <class Type>
void queueType<Type>::initializeQueue()
{
        queueFront = 0;
        queueRear = maxQueueSize - 1;
        count = 0;
} //end initializeQueue
```

6.2.3 Front

This operation returns the first element of the queue. If the queue is nonempty, the element of the queue indicated by the index queueFront is returned; otherwise, the program terminates.

```
template <class Type>
Type queueType<Type>::front() const
{
        assert(!isEmptyQueue()); return
        list[queueFront];
} //end front
```

6.2.4 Back

This operation returns the last element of the queue. If the queue is nonempty, the element of the queue indicated by the index queueRear is returned; otherwise, the program terminates.

```
template <class Type>
Type queueType<Type>::back() const
{
```

```
        assert(!isEmptyQueue()); return
        list[queueRear];
} //end back
```

6.2.5 Add Queue

Next, we implement the addQueue operation. Because queueRear points to the last element of the queue, to add a new element to the queue, we first advance queueRear to the next array position, and then add

the new element to the array position indicated by queueRear. We also increment count by 1. So the function addQueue is as follows:

```
template <class Type>
void queueType<Type>::addQueue(const Type& newElement)
{
        if (!isFullQueue())
        {
                queueRear = (queueRear + 1) % maxQueueSize; //use the
                //mod operator to advance queueRear
                //because the array is circular count++;
                list[queueRear] = newElement;
        }
        else cout << "Cannot add to a full queue." << endl;
} //end addQueue
```

6.2.6 Delete Queue

To implement the deleteQueue operation, we access the index queueFront. Because queueFront points to the array position containing the first element of the queue, to remove the first queue element, we decrement count by 1 and advance queueFront to the next queue element. So the function deleteQueue is as follows:

```
template <class Type>
void queueType<Type>::deleteQueue()
{
        if (!isEmptyQueue())
        { count--;
                queueFront = (queueFront + 1) % maxQueueSize; //use the
                //mod operator to advance queueFront
                //because the array is circular
        }
        else cout << "Cannot remove from an empty queue" << endl; }
```

```
//end deleteQueue
```

6.2.7 Constructors and Destructors

To complete the implementation of the queue operations, we next consider the implementation of the constructor and the destructor. The constructor gets the maxQueueSize from the user, sets the variable maxQueueSize to the value specified by the user, and creates an array of size maxQueueSize. If the user does not specify the queue size, the constructor uses the default value, which is 100, to create an array of size 100. The constructor also initializes queueFront and queueRear to indicate that the queue is empty. The definition of the function to implement the constructor is as follows:

```
template <class Type>
queueType<Type>::queueType(int queueSize)
{
        if (queueSize <= 0)
```

```
    {
        cout << "Size of the array to hold the queue must "<< "be positive." << endl; cout
        << "Creating an array of size 100." << endl;
        maxQueueSize = 100;
    }
    else

        maxQueueSize = queueSize;                    //set maxQueueSize to queueSize
    queueFront = 0;                                  //initialize queueFront
    queueRear = maxQueueSize - 1;        count       //initialize queueRear
    = 0;
    list = new Type[maxQueueSize];                   //create the array to
                                                     //hold the queue elements
}                                                    //end constructor
```

The array to store the queue elements is created dynamically. Therefore, when the queue object goes out of scope, the destructor simply deallocates the memory occupied by the array that stores the queue elements. The definition of the function to implement the destructor is as follows:

```
template <class Type>
queueType<Type>::~queueType()
{ delete [] list;
}
```

The implementation of the copy constructor and overloading the assignment operator are left as exercises for you. (The definitions of these functions are similar to those discussed for array-based lists and stacks.)

6.3 Linked Implementation of Queues

Because the size of the array to store the queue elements is fixed, only a finite number of queue elements can be stored in the array. Also, the array implementation of the queue requires the array to be treated in a special way together with the values of the indices queueFront and queueRear. The linked

implementation of a queue simplifies many of the special cases of the array implementation and, because the memory to store a queue element is allocated dynamically, the queue is never full. This section discusses the linked implementation of a queue.

Because elements are added at one end, queueRear, and removed from the other end, queueFront, we need to know the front of the queue and the rear of the queue. Thus, we need two pointers, queueFront and queueRear, to maintain the queue. The following class implements the functions of the abstract class queueADT:

```
//*************************************************************
// This class specifies the basic operations on a queue as a //
linked list.
//*************************************************************
//Definition of the node template
<class Type>
```

```
struct nodeType
{
        Type info;
        nodeType<Type> *link;
};

template <class Type>
class linkedQueueType: public queueADT<Type>
{
        public:
                const linkedQueueType<Type>& operator=(const linkedQueueType<Type>&); //Overload
                the assignment operator.

                bool isEmptyQueue() const;
                //Function to determine whether the queue is empty.
                //Postcondition: Returns true if the queue is empty,
                // otherwise returns false.

                bool isFullQueue() const;
                //Function to determine whether the queue is full.
                //Postcondition: Returns true if the queue is full, //
                otherwise returns false.

                void initializeQueue();
                //Function to initialize the queue to an empty state.
                //Postcondition: queueFront = NULL; queueRear = NULL

                Type front() const;
                //Function to return the first element of the queue. //Precondition:
                The queue exists and is not empty.
                //Postcondition: If the queue is empty, the program
                // terminates; otherwise, the first element of the //
                queue is returned.
```

ent of the queue. //Precondition: The queue exists and is not
Type back() const; empty.
/ //Postcondition: If the queue is empty, the program
/ // terminates; otherwise, the last element of the //
F queue is returned.
u
n void addQueue(const Type& queueElement);
c //Function to add queueElement to the queue.
t //Precondition: The queue exists and is not full.
i //Postcondition: The queue is changed and queueElement is //
o added to the queue.
n
t void deleteQueue();
o
r
e
t
u
r
n
t
h
e
l
a
s
t
e
l
e
m

```
        //Function to remove the first element of the queue.
        //Precondition: The queue exists and is not empty.
        //Postcondition: The queue is changed and the first element //
        is removed from the queue.

        linkedQueueType(); //Default
        constructor

        linkedQueueType(const linkedQueueType<Type>& otherQueue); //Copy
        constructor

        ~linkedQueueType();
        //Destructor

    private:
        nodeType<Type> *queueFront;        //pointer to the front of the queue nodeType<Type>
        *queueRear;   //pointer to the rear of the queue
};
```

The UML diagram of the class linkedQueueType is left as an exercise for you. (See Exercise 16 at the end of this chapter.)

Next, we write the definitions of the functions of the class linkedQueueType.

6.1.1 Empty and Full Queue

The queue is empty if queueFront is NULL. Memory to store the queue elements is allocated dynamically. Therefore, the queue is never full and so the function to implement the isFullQueue operation returns the value false. (The queue is full only if the program runs out of memory.)

```
template <class Type>
bool linkedQueueType<Type>::isEmptyQueue() const
{
```

```
    return(queueFront == NULL);
} //end

template <class Type>
bool linkedQueueType<Type>::isFullQueue() const
{ return false;
} //end isFullQueue
```

Note that in reality, in the linked implementation of queues, the function isFullQueue does not apply because logically the queue is never full. However, you must provide its definition because it is included as an abstract function in the parent class queueADT.

6.1.2 Initialize Queue

The operation initializeQueue initializes the queue to an empty state. The queue is empty if there are no elements in the queue. Note that the constructor initializes the queue when the queue object is declared. So this operation must remove all the elements, if any, from the queue. Therefore, this operation traverses the list containing the queue starting at the first node, and it deallocates the memory occupied by the queue elements. The definition of this function is as follows:

```
template <class Type>
void linkedQueueType<Type>::initializeQueue()
{
        nodeType<Type> *temp;
        while (queueFront!= NULL)                    //while there are elements left in the queue
        {
                temp = queueFront;            //set temp to point to the current node queueFront =
                queueFront->link;       //advance first to the next node
                delete temp;                         //deallocate memory occupied by temp
        }
        queueRear = NULL;                            //set rear to NULL
} //end initializeQueue
```

6.1.3 addQueue, front, back, and deleteQueue Operations

The addQueue operation adds a new element at the end of the queue. To implement this operation, we access the pointer queueRear.

If the queue is nonempty, the operation front returns the first element of the queue and so the element of the queue indicated by the pointer queueFront is returned. If the queue is empty, the function front terminates the program.

If the queue is nonempty, the operation back returns the last element of the queue and so the element

of the queue indicated by the pointer queueRear is returned. If the queue is empty, the function back terminates the program. Similarly, if the queue is nonempty, the operation deleteQueue removes the first element of the queue, and so we access the pointer queueFront.

The definitions of the functions to implement these operations are as follows:

```
template <class Type>
void linkedQueueType<Type>::addQueue(const Type& newElement)
{
        nodeType<Type> *newNode;
        newNode = new nodeType<Type>;          //create the node
        newNode->info = newElement;            //store the info
        newNode->link = NULL;                  //initialize the link field to
                                               NULL
```

```
        if (queueFront == NULL)                    //if initially the queue is
                                                   empty
        {
                queueFront = newNode;
                queueRear = newNode;
        }
        else                                       //add newNode at the end
        {
                 queueRear->link = newNode;
                 queueRear = queueRear->link;
        }
}                                                  //end addQueue

template <class Type>
Type linkedQueueType<Type>::front() const
{
        assert(queueFront    !=    NULL);
        return queueFront->info;
}                                                  //end front

template <class Type>
Type linkedQueueType<Type>::back() const
{
        assert(queueRear!= NULL);
        return queueRear->info;
}                                                  //end back

template <class Type>
void linkedQueueType<Type>::deleteQueue()
{
        nodeType<Type> *temp;
        if (!isEmptyQueue())
```

```
        {

                temp = queueFront;                  //make temp point to the first
                                                    node
                queueFront = queueFront->link;      //advance queueFront
                delete temp;                        //delete the first node
                if (queueFront == NULL)             //if after deletion the queue is
                                                    empty
                        queueRear = NULL;           //set queueRear to NULL
        }
        else cout << "Cannot remove from an empty queue" << endl;
}                                                   //end deleteQueue
```

The definition of the default constructor is as follows:

```
template<class Type>
```

```
linkedQueueType<Type>::linkedQueueType()
{
        queueFront = NULL;                              //set front to null
        queueRear = NULL;                               //set rear to null
}                                                       //end default constructor
```

When the queue object goes out of scope, the destructor destroys the queue; that is, it deallocates the memory occupied by the elements of the queue. The definition of the function to implement the destructor is similar to the definition of the function initializeQueue. Also, the functions to implement the copy constructor and overload the assignment operators are similar to the corresponding functions for stacks.

EXAMPLE 6-1

The following program tests various operations on a queue. It uses the class linkedQueueType to implement a queue.

```
//*************************************************************
* // This program illustrates how to use the class linkedQueueType // in a
program.
//*************************************************************
* #include <iostream> #include "linkedQueue.h" using namespace std;
int main()
{
        linkedQueueType<int> queue;
        int x, y;
        queue.initializeQueue();
        x = 4; y
        = 5;
        queue.addQueue(x);
        queue.addQueue(y); x =
        queue.front();
        queue.deleteQueue();
        queue.addQueue(x + 5);
        queue.addQueue(16);
        queue.addQueue(x);
```

```
        queue.addQueue(y - 3);
        cout << "Queue Elements:
        ";
        while (!queue.isEmptyQueue())
        {
                cout << queue.front() << " ";
                queue.deleteQueue();
        }
        cout << endl;
        return 0;
}
```

Sample Run:

Queue Elements: 5 9 16 4 2

6.1.4 Queue Derived from the class unorderedLinkedList

From the definitions of the functions to implement the queue operations, it is clear that the linked implementation of a queue is similar to the implementation of a linked list created in a forward manner (see Chapter 3). The addQueue operation is similar to the operation insertFirst. Likewise, the operations initializeQueue and initializeList, isEmptyQueue, and isEmptyList are similar. The deleteQueue operation can be implemented as before. The pointer queueFront is the same as the pointer first, and the pointer queueRear is the same as the pointer last. This correspondence suggests that we can derive the class to implement the queue from the class linkedListType (see Chapter 3). Note that the class linkedListType is an abstract and does not implement all the operations. However, the class unorderedLinkedList is derived from the the class linkedListType and provides the definitions of the abstract functions of the the class linkedListType. Therefore, we can derive the class linkedQueueType from the class unorderedLinkedList.

6.2 STL class queue (Queue Container Adapter)

The preceding sections discussed the data structure queue in detail. Because a queue is an important data structure, the Standard Template Library (STL) provides a class to implement queues in a program. The name of the class defining the queue is queue, and the name of the header file containing the definition of the class queue is queue. The class queue provided by the STL is implemented similar to the classes discussed in this chapter. Table 6-1 defines various operations supported by the queue container class.

TABLE 6-1: Operations on a queue object

Operation	Effect
`size`	Returns the actual number of elements in the queue.
`empty`	Returns **`true`** if the queue is empty, and **`false`** otherwise.
`push(item)`	Inserts a copy of `item` into the queue.
`front`	Returns the next—that is, first—element in the queue, but does not remove the element from the queue. This operation is implemented as a value-returning function.
`back`	Returns the last element in the queue, but does not remove the element from the queue. This operation is implemented as a value-returning function.
`pop`	Removes the next element in the queue.

In addition to the operations size, empty, push, front, back, and pop, the queue container class provides relational operators to compare two queues. For example, the relational operator == can be used to determine whether two queues are identical.

The program in Example 6-2 illustrates how to use the queue container class.

EXAMPLE 6-2

```
//*************************************************************
// This program illustrates how to use the STL class queue in a program.
//*************************************************************
```

```
#include <iostream>                                                          //Line 1
#include <queue>                                                            //Line 2
using namespace std;                                                        //Line 3
int main()                                                                  //Line 4
{                                                                           //Line 5
    queue<int> intQueue;                                                    //Line 6
    intQueue.push(26);                                                      //Line 7
    intQueue.push(18);                                                      //Line 8
    intQueue.push(50);                                                      //Line 9
    intQueue.push(33);                                                      //Line 10
    cout << "Line 11: The front element of intQueue: "<< intQueue.front() << endl;  //Line 11
    cout << "Line 12: The last element of intQueue: "<< intQueue.back() << endl;    //Line 12
    intQueue.pop();                                                         //Line 13
cout << "Line 14: After the pop operation, the "<< "front element of intQueue: "
        << intQueue.front() << endl;                                        //Line 14
    cout << "Line 15: intQueue elements: ";                                 //Line 15
    while (!intQueue.empty())                                               //Line 16
    {                                                                       //Line 17
        cout << intQueue.front() << " ";                                    //Line 18
        intQueue.pop();                                                     //Line 19
    }                                                                       //Line 20
    cout << endl;                                                           //Line 21
    return 0;                                                               //Line 22
}                                                                           //Line 23
```

Sample Run:

Line 11: The front element of intQueue: 26

Line 12: The last element of intQueue: 33

Line 14: After the pop operation, the front element of intQueue: 18

Line 15: intQueue elements: 18 50 33

The preceding output is self-explanatory. The details are left as an exercise for you.

6.4 Priority Queues

The preceding sections describe how to implement a queue in a program. The use of a queue structure ensures that the items are processed in the order they are received. For example, in a banking environment, the customers who arrive first are served first. However, there are certain situations when

this First In First Out rule needs to be relaxed somewhat. In a hospital environment, patients are, usually, seen in the order they arrive. Therefore, you could use a queue to ensure that the patients are seen in the order they arrive. However, if a patient arrives with severe or life-threatening symptoms, they are treated first. In other words, these patients take priority over the patients who can wait to be seen, such as those awaiting their routine annual checkup. For another example, in a shared environment, when print requests are sent to the printer, interactive programs take priority over batchprocessing programs.

There are many other situations where some priority is assigned to the customers. To implement such a data structure in a program, we use a special type of queue, called priority queues. In a priority queue, customers or jobs with higher priority are pushed to the front of the queue.

One way to implement a priority queue is to use an ordinary linked list, which keeps the items in order from the highest to lowest priority. However, an effective way to implement a priority queue is to use a treelike structure. In Chapter 8, we discuss a special type of sorting algorithm, called the heapsort, which uses a treelike structure to sort a list. After describing this algorithm, we discuss how to effectively implement a priority queue.

6.2.1 STL class priority_queue

The STL provides the class template priority_queue<elemType>, where the data type of the queue elements is specified by elemType. This class template is contained in the STL header file queue. You can specify the priority of the elements of a priority queue in various ways. The default priority criteria for the queue elements uses the lessthan operator, <. For example, a program that implements a priority queue of numbers could use the operator < to assign the priority to the numbers so that larger numbers are always at the front of the queue. If you design your own class to implement the queue elements, you can specify your priority rule by overloading the less-than operator, <, to compare the elements. You could also define a comparison function to specify the priority.

6.3 Application of Queues: Simulation

A technique in which one system models the behavior of another system is called simulation. For example, physical simulators include wind tunnels used to experiment with the design of car bodies and flight simulators are used to train airline pilots. Simulation techniques are used when it is too expensive or

dangerous to experiment with real systems. You can also design computer models to study the behavior of real systems. (We will describe some real systems modeled by computers shortly.) Simulating the behavior of an expensive or dangerous experiment using a computer model is usually less expensive than using the real system, and a good way to gain insight without putting human life in danger. Moreover, computer simulations are particularly useful for complex systems where it is difficult to construct a mathematical model. For such systems, computer models can retain descriptive accuracy. In mathematical simulations, the steps of a program are used to model the behavior of a real system. Let us consider one such problem.

The manager of a local movie theater is hearing complaints from customers about the time they have to wait in line to buy tickets. The theater currently has only one cashier. Another theater is preparing to open in the neighborhood and the manager is afraid of losing customers. The manager wants to hire enough cashiers so that a customer does not have to wait too long to buy a ticket, but does not want to hire extra cashiers on a trial basis and potentially waste time and money. One thing that the manager would like to

know is the average time a customer has to wait for service. The manager wants someone to write a program to simulate the behavior of the theater.

In computer simulation, the objects being studied are usually represented as data. For the theater problem, some of the objects are the customers and the cashier. The cashier serves the customers and we want to determine a customer's average waiting time. Actions are implemented by writing algorithms, which in a programming language are implemented with the help of functions. Thus, functions are used to implement the actions of the objects. In C++, we can combine the data and the operations on that data into a single unit with the help of classes. Thus, objects can be represented as classes. The member variables of the class describe the properties of the objects, and the function members describe the actions on that data. This change in simulation results can also occur if we change the values of the data or modify the definitions of the functions (that is, modify the algorithms implementing the actions). The main goal of a computer simulation is to either generate results showing the performance of an existing system or predict the performance of a proposed system.

In the theater problem, when the cashier is serving a customer, the other customers must wait. Because customers are served on a first-come, first-served basis and queues are an effective way to implement a First In First Out system, queues are important data structures for use in computer simulations. This section examines computer simulations in which queues are the basic data structure. These simulations model the behavior of systems, called queuing systems, in which queues of objects are waiting to be served by various servers. In other words, a queuing system consists of servers and queues of objects waiting to be served. We deal with a variety of queuing systems on a daily basis. For example, a grocery store and a banking system are both queuing systems. Furthermore, when you send a print request to a networked printer that is shared by many people, your print request goes in a queue. Print requests that arrived before your print request are usually completed before yours. Thus, the printer acts as the server when a queue of documents is waiting to be printed.

6.3.1 Designing a Queuing System

In this section, we describe a queuing system that can be used in a variety of applications, such as a bank, grocery store, movie theater, printer, or mainframe environment in which several people are trying to use the same processors to execute their programs. To describe a queuing system, we use the term server for the object that provides the service. For example, in a bank, a teller is a server; in a grocery store or movie

theater, a cashier is a server. We will call the object receiving the service the customer, and the service time — the time it takes to serve a customer—the transaction time.

Because a queuing system consists of servers and a queue of waiting objects, we will model a system that consists of a list of servers and a waiting queue holding the customers to be served. The customer at the front of the queue waits for the next available server. When a server becomes free, the customer at the front of the queue moves to the free server to be served.

When the first customer arrives, all servers are free and the customer moves to the first server. When the next customer arrives, if a server is available, the customer immediately moves to the available server; otherwise, the customer waits in the queue. To model a queuing system, we need to know the number of servers, the expected arrival time of a customer, the time between the arrivals of customers, and the number of events affecting the system.

Let us again consider the movie theater system. The performance of the system depends on how many servers are available, how long it takes to serve a customer, and how often a customer arrives. If it takes too long to serve a customer and customers arrive frequently, then more servers are needed. This system can be modeled as a time-driven simulation. In a time-driven simulation, the clock is implemented as a counter and the passage of, say, 1 minute can be implemented by incrementing the counter by 1. The simulation is run for a fixed amount of time. If the simulation needs to be run for 100 minutes, the counter starts at 1 and goes up to 100, which can be implemented by using a loop.

For the simulation described in this section, we want to determine the average wait time for a customer. To calculate the average wait time for a customer, we need to add the waiting time of each customer, and then divide the sum by the number of customers who have arrived. When a customer arrives, he or she goes to the end of the queue and the customer's waiting time starts. If the queue is empty and a server is free, the customer is served immediately and so this customer's waiting time is zero. On the other hand, if when the customer arrives and either the queue is nonempty or all the servers are busy, the customer must wait for the next available server and, therefore, this customer's waiting time starts. We can keep track of the customer's waiting time by using a timer for each customer. When a customer arrives, the timer is set to 0, which is incremented after each clock unit.
Suppose that, on average, it takes five minutes for a server to serve a customer. When a server becomes free and the waiting customer's queue is nonempty, the customer at the front of the queue proceeds to begin the transaction. Thus, we must keep track of the time a customer is with a server. When the customer arrives at a server, the transaction time is set to five and is decremented after each clock unit. When the transaction time becomes zero, the server is marked free. Hence, the two objects needed to implement a time-driven computer simulation of a queuing system are the customer and the server.

Before designing the main algorithm to implement the simulation, we design classes to implement each of the two objects: customer and server.

6.3.2 Customer

Every customer has a customer number, arrival time, waiting time, transaction time, and departure time. If we know the arrival time, waiting time, and transaction time, we can determine the departure time by adding these three times. Let us call the class to implement the customer object customerType. It follows that the class customerType has four member variables: the customerNumber, arrivalTime, waitingTime,

and transactionTime, each of type int. The basic operations that must be performed on an object of type customerType are as follows: Set the customer's number, arrival time, and waiting time; increment the waiting time by one clock unit; return the waiting time; return the arrival time; return the transaction time; and return the customer number. The following class, customerType, implements the customer as an ADT:

```
//*********************************************************
// class customerType
// This class specifies the members to implement a customer.
//********************************************************* class
customerType
{ public: customerType(int cN = 0, int arrvTime = 0, int wTime = 0, int tTime = 0);
                //Constructor to initialize the instance variables
                //according to the parameters
```

```
//If no value is specified in the object declaration, //the
default values are assigned.
//Postcondition: customerNumber = cN; arrivalTime = arrvTime;
// waitingTime = wTime; transactionTime = tTime

void setCustomerInfo(int cN = 0, int inTime = 0, int wTime = 0, int tTime = 0); //Function
to initialize the instance variables.
//Instance variables are set according to the parameters.
//Postcondition: customerNumber = cN; arrivalTime = arrvTime;
// waitingTime = wTime; transactionTime = tTime;

int getWaitingTime() const;
//Function to return the waiting time of a customer.
//Postcondition: The value of waitingTime is returned.

void setWaitingTime(int time);
//Function to set the waiting time of a customer.
//Postcondition: waitingTime = time;

void incrementWaitingTime();
//Function to increment the waiting time by one time unit.
//Postcondition: waitingTime++;

int getArrivalTime() const;
//Function to return the arrival time of a customer.
//Postcondition: The value of arrivalTime is returned.

int getTransactionTime() const;
//Function to return the transaction time of a customer.
//Postcondition: The value of transactionTime is returned.

int getCustomerNumber() const;
//Function to return the customer number.
```

```
        //Postcondition: The value of customerNumber is returned.

    private:
        int customerNumber;
        int arrivalTime; int
        waitingTime;
        int transactionTime;
};
```

Figure 6-11 shows the UML diagram of the class customerType.

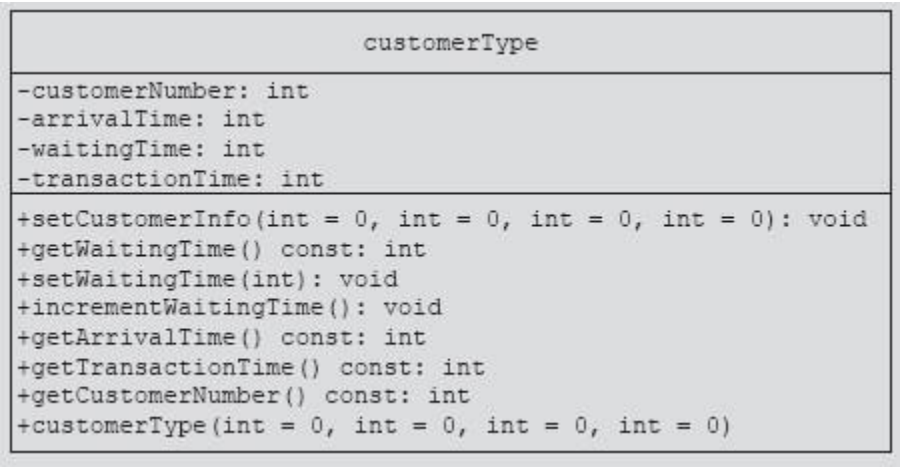


FIGURE 6-11 UML diagram of the class customerType

The definitions of the member functions of the class customerType follow easily from their descriptions. Next, we give the definitions of the member functions of the class customerType.

The function setCustomerInfo uses the values of the parameters to initialize customerNumber, arrivalTime, waitingTime, and transactionTime. Its definition is as follows:

```
void customerType::setCustomerInfo(int cN, int arrvTime, int wTime, int tTime)
{
        customerNumber = cN;
        arrivalTime = arrvTime;
        waitingTime = wTime;
        transactionTime = tTime;
}
```

The definition of the constructor is similar to the definition of the function setCustomerInfo. It uses the values of the parameters to initialize customerNumber, arrivalTime, waitingTime, and transactionTime. To make debugging easier, we use the function setCustomerInfo to write the definition of the constructor, which is given next.

```
customerType::customerType(int cN, int arrvTime, int wTime, int tTime)
{
        setCustomerInfo(cN, arrvTime, wTime, tTime);
}
```

The function getWaitingTime returns the current waiting time. The definition of the function getWaitingTime is as follows:

```
int customerType::getWaitingTime() const
{
        return waitingTime;
}
```

The function incrementWaitingTime increments the value of waitingTime. Its definition is as follows:

```
void customerType::incrementWaitingTime()
{
        waitingTime++;
}
```

The definitions of the functions setWaitingTime, getArrivalTime, getTransactionTime, and getCustomerNumber are left as an exercise for you.

6.5 Server

At any given time unit, the server is either busy serving a customer or is free.We use a string variable to set the status of the server. Every server has a timer and because the program might need to know which customer is served by which server, the server also stores the information of the customer being served. Thus, three member variables are associated with a server: the status, the transactionTime, and the currentCustomer. Some of the basic operations that must be performed on a server are as follows: Check whether the server is free; set the server as free; set the server as busy; set the transaction time (that is, how long it takes to serve the customer); return the remaining transaction time (to determine whether the server should be set to free); if the server is busy after each time unit, decrement the transaction time by one time unit; and so on. The following class, serverType, implements the server as an ADT:

```
//*********************************************************
// class serverType
// This class specifies the members to implement a server.
//*********************************************************
****** class serverType
{
        public:
                serverType(); //Default
                constructor
                //Sets the values of the instance variables to their default //values.
                //Postcondition: currentCustomer is initialized by its
                // default constructor; status = "free"; and the //
                transaction time is initialized to 0.
```

```
bool isFree() const;
//Function to determine if the server is free.
//Postcondition: Returns true if the server is free, //
otherwise returns false.

void setBusy();
//Function to set the status of the server to busy.
//Postcondition: status = "busy";

void setFree();
//Function to set the status of the server to "free."
//Postcondition: status = "free";
void setTransactionTime(int t);
```

```
//Function to set the transaction time according to the //parameter
t.
//Postcondition: transactionTime = t;

void setTransactionTime();
//Function to set the transaction time according to //the
transaction time of the current customer.
//Postcondition:
// transactionTime = currentCustomer.transactionTime;

int getRemainingTransactionTime() const; //Function
to return the remaining transaction time.
//Postcondition: The value of transactionTime is returned.

void decreaseTransactionTime();
//Function to decrease the transactionTime by 1 unit.
//Postcondition: transactionTime--;

void setCurrentCustomer(customerType cCustomer);
//Function to set the info of the current customer
//according to the parameter cCustomer.
//Postcondition: currentCustomer = cCustomer;

int getCurrentCustomerNumber() const;
//Function to return the customer number of the current //customer.
//Postcondition: The value of customerNumber of the //
current customer is returned.

int getCurrentCustomerArrivalTime() const;
//Function to return the arrival time of the current
//customer.
//Postcondition: The value of arrivalTime of the current //
customer is returned.
```

```
        int getCurrentCustomerWaitingTime() const;
        //Function to return the current waiting time of the
        //current customer.
        //Postcondition: The value of transactionTime is returned.

        int getCurrentCustomerTransactionTime() const;
        //Function to return the transaction time of the
        //current customer.
        //Postcondition: The value of transactionTime of the //
        current customer is returned.

private:
        customerType currentCustomer; string
        status;
```

```
        int transactionTime;
};
```

Figure 6-12 shows the UML diagram of the class serverType.

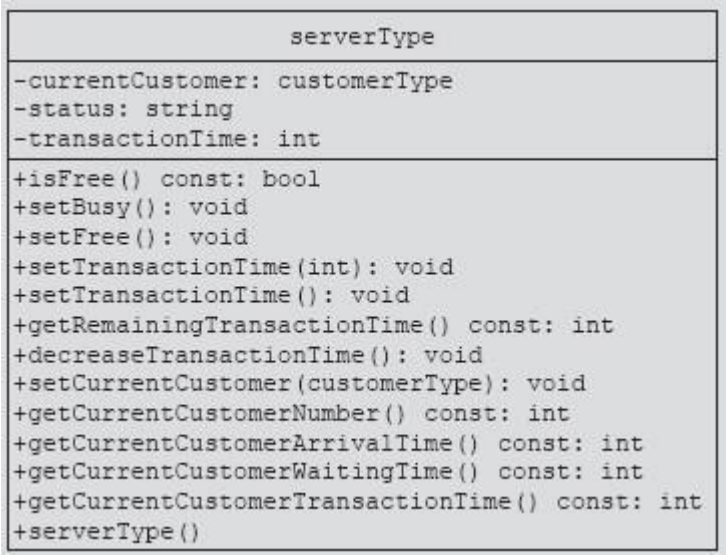


FIGURE 6-12 UML diagram of the class serverType

The definitions of some of the member functions of the class serverType are as follows:

```
serverType::serverType()
{ status = "free";
        transactionTime = 0;
}

bool serverType::isFree() const
{ return (status == "free");
}

void serverType::setBusy()
{ status = "busy";
}

void serverType::setFree()
{ status = "free";
}
```

```
void serverType::setTransactionTime(int t)
{ transactionTime = t;
}

void serverType::setTransactionTime()
{ int time;
        time = currentCustomer.getTransactionTime(); transactionTime
        = time;
}
```

```
void serverType::decreaseTransactionTime()
{
        transactionTime--;
}
```

We leave the definitions of the functions getRemainingTransactionTime, setCurrentCustomer, getCurrentCustomerNumber, getCurrentCustomerArrivalTime, getCurrentCustomerWaitingTime, and getCurrentCustomerTransactionTime as an exercise for you.

Because we are designing a simulation program that can be used in a variety of applications, we need to design two more classes: a class to create and process a list of servers and a class to create and process a queue of waiting customers. The next two sections describe each of these classes.

6.3.3 Server List

A server list is a set of servers. At any given time, a server is either free or busy. For the customer at the front of the queue, we need to find a server in the list that is free. If all the servers are busy, the customer must wait until one of the servers becomes free. Thus, the class that implements a list of servers has two member variables: one to store the number of servers and one to maintain a list of servers. Using dynamic arrays, depending on the number of servers specified by the user, a list of servers is created during program execution. Some of the operations that must be performed on a server list are as follows: Return the server number of a free server; when a customer gets ready to do business and a server is available, set the server to busy; when the simulation ends, some of the servers might still be busy, so return the number of busy servers; after each time unit, reduce the transactionTime of each busy server by one time unit; and if the transactionTime of a server becomes zero, set the server to free. The following class, serverListType, implements the list of servers as an ADT:

```
//*************************************************************
// class serverListType
// This class specifies the members to implement a list of servers.
//************************************************************* class
serverListType
```

```
{ public: serverListType(int num = 1);
                //Constructor to initialize a list of servers
                //Postcondition: numOfServers = num
                // A list of servers, specified by num, is created and //
                each server is initialized to "free".

                ~serverListType();
                //Destructor
                //Postcondition: The list of servers is destroyed.

                int getFreeServerID() const;
                //Function to search the list of servers.
                //Postcondition: If a free server is found, returns its ID; //
                otherwise, returns -1.
```

```
        int getNumberOfBusyServers() const;
        //Function to return the number of busy servers.
        //Postcondition: The number of busy servers is returned.

        void setServerBusy(int serverID, customerType cCustomer, int tTime); //Function
        to set a server busy.
        //Postcondition: The server specified by serverID is set to
        // "busy", to serve the customer specified by cCustomer,
        // and the transaction time is set according to the //
        parameter tTime.

        void setServerBusy(int serverID, customerType cCustomer); //Function
        to set a server busy.
        //Postcondition: The server specified by serverID is set to
        // "busy", to serve the customer specified by cCustomer.

        void updateServers(ostream& outFile); //Function
        to update the status of a server.
        //Postcondition: The transaction time of each busy server
        // is decremented by one unit. If the transaction time of
        // a busy server is reduced to zero, the server is set to
        // "free". Moreover, if the actual parameter corresponding
        // to outFile is cout, a message indicating which customer
        // has been served is printed on the screen, together with the
        // customer's departing time. Otherwise, the output is sent //
        to a file specified by the user.

    private:
        int numOfServers;
        serverType *servers;
};
```

Figure 6-13 shows the UML diagram of the class serverListType.

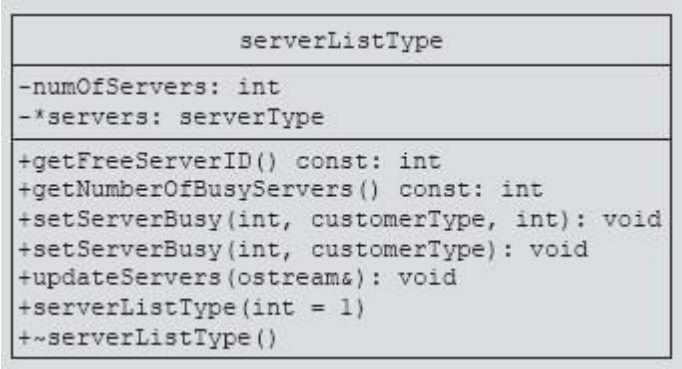


FIGURE 6-13 UML diagram of the class serverListType

Following are the definitions of the member functions of the class serverListType. The definitions of the constructor and destructor are straightforward.

serverListType::serverListType(int num)

```
{
        numOfServers = num;
        servers = new serverType[num];
}

serverListType::~serverListType()
{ delete [] servers;
}
```

The function getFreeServerID searches the list of servers. If a free server is found, it returns the server's ID; otherwise, the value -1 is returned, which indicates that all the servers are busy. The definition of this function is as follows:

```
int serverListType::getFreeServerID() const
{ int serverID = -1;
        for (int i = 0; i < numOfServers; i++)
                if (servers[i].isFree())
                { serverID = i;
                        break;
                } return
        serverID;
}
```

The function getNumberOfBusyServers searches the list of servers and determines the number of busy servers. The number of busy servers is returned. The definition of this function is as follows:

```
int serverListType::getNumberOfBusyServers() const
{ int busyServers = 0;
        for (int i = 0; i < numOfServers; i++) if
                (!servers[i].isFree())
                busyServers++;
        return busyServers;
}
```

The function setServerBusy sets a server to busy. This function is overloaded. The serverID of the server that is set to busy is passed as a parameter to this function. One function sets the server's transaction time according to the parameter tTime; the other function sets it by using the transaction time stored in the object cCustomer. The transaction time is later needed to determine the average waiting time. The definitions of these functions are as follows:

```
void serverListType::setServerBusy(int serverID, customerType cCustomer, int tTime)
{
        servers[serverID].setBusy();
        servers[serverID].setTransactionTime(tTime);
        servers[serverID].setCurrentCustomer(cCustomer);
}

void serverListType::setServerBusy(int serverID, customerType cCustomer)
```

```
{
        int time = cCustomer.getTransactionTime();
        servers[serverID].setBusy();
        servers[serverID].setTransactionTime(time);
        servers[serverID].setCurrentCustomer(cCustomer);
}
```

The definition of the function updateServers is quite straightforward. Starting at the first server, it searches the list of servers for busy servers. When a busy server is found, its transactionTime is decremented by 1. If the transactionTime reduces to zero, the server is set to free. If the transactionTime of a busy server reduces to zero, the transaction of the customer being served by the server is complete. If the actual parameter corresponding to outFile is cout, a message indicating which customer has been served is printed on the screen, together with the customer's departing time. Otherwise, the output is sent to a file specified by the user. The definition of this function is as follows:

```
void serverListType::updateServers(ostream& outF)
{ for (int i = 0; i < numOfServers; i++)
                if (!servers[i].isFree())
                {
                        servers[i].decreaseTransactionTime();
                        if (servers[i].getRemainingTransactionTime() == 0)
                        {
                                outF << "From server number " << (i + 1)
                                        << " customer number "
                                        << servers[i].getCurrentCustomerNumber()
                                        << "\n departed at clock unit "
                                        << servers[i].getCurrentCustomerArrivalTime()
                                        + servers[i].getCurrentCustomerWaitingTime()
                                        + servers[i].getCurrentCustomerTransactionTime()
                                        << endl; servers[i].setFree();
                        }
                }
}
```

6.3.4 Waiting Customers Queue

When a customer arrives, he or she goes to the end of the queue. When a server becomes available, the customer at the front of the queue leaves to conduct the transaction. After each time unit, the waiting time of each customer in the queue is incremented by 1. The ADT queueType designed in this chapter has all the operations needed to implement a queue, except the operation of incrementing the waiting time of each customer in the queue by one time unit. We will derive a class, waitingCustomerQueueType, from the class queueType and add the additional operations to implement the customer queue. The definition of the class waitingCustomerQueueType is as follows:

```
//*********************************************************
// class waitingCustomerQueueType
// This class extends the class queueType to implement a list //
of waiting customers.
```

```
//********************************************************** class
waitingCustomerQueueType: public queueType<customerType>
{ public:
        waitingCustomerQueueType(int size = 100);
        //Constructor
        //Postcondition: The queue is initialized according to the
        // parameter size. The value of size is passed to the //
        constructor of queueType.

        void updateWaitingQueue();
        //Function to increment the waiting time of each //customer
        in the queue by one time unit.
};
```

The definitions of the member functions are given next. The definition of the constructor is as follows:

```
waitingCustomerQueueType::waitingCustomerQueueType(int size):queueType<customerType>(size)
{
}
```

The function updateWaitingQueue increments the waiting time of each customer in the queue by one time unit. The class waitingCustomerQueueType is derived from the class queueType. Because the member variables of queueType are private, the function updateWaitingQueue cannot directly access the elements of the queue. The only way to access the elements of the queue is to use the deleteQueue operation.After incrementing the waiting time, the element can be put back into the queue by using the addQueue operation.

The addQueue operation inserts the element at the end of the queue. If we perform the deleteQueue operation followed by the addQueue operation for each element of the queue, eventually the front element again becomes the front element. Given that each deleteQueue operation is followed by an addQueue operation, how do we determine that all the elements of the queue have been processed? We cannot use the isEmptyQueue or isFullQueue operations on the queue because the queue will never be empty or full.

One solution to this problem is to create a temporary queue. Every element of the original queue is

removed, processed, and inserted into the temporary queue. When the original queue becomes empty, all of the elements in the queue are processed. We can then copy the elements from the temporary queue back into the original queue. However, this solution requires us to use extra memory space, which could be significant. Also, if the queue is large, extra computer time is needed to copy the elements from the temporary queue back into the original queue. Let us look into another solution.

In the second solution, before starting to update the elements of the queue, we can insert a dummy customer with a waiting time of, say -1. During the update process, when we arrive at the customer with the waiting time of -1, we can stop the update process without processing the customer with the waiting time of -1. If we do not process the customer with the waiting time -1, this customer is removed from the queue and after processing all the elements of the queue, the queue will contain no extra elements. This solution does not require us to create a temporary queue, so we do not need extra computer time to copy

the elements back into the original queue. We will use this solution to update the queue. Therefore, the definition of the function updateWaitingQueue is as follows:

```
void waitingCustomerQueueType::updateWaitingQueue()
{
        customerType cust;

        cust.setWaitingTime(-1); int
        wTime = 0;
        addQueue(cust);
        while (wTime != -1)
        {
                cust = front();
                deleteQueue();

                wTime = cust.getWaitingTime();
                if (wTime == -1) break;
                cust.incrementWaitingTime();  addQueue(cust);
        }
}
```

6.3.5 Main Program

To run the simulation, we first need to get the following information:

- The number of time units the simulation should run. Assume that each time unit is one minute.
- The number of servers.
- The amount of time it takes to serve a customer—that is, the transaction time. The
- approximate time between customer arrivals.

These pieces of information are called simulation parameters. By changing the values of these parameters, we can observe the changes in the performance of the system. We can write a function, setSimulationParameters, to prompt the user to specify these values. The definition of this function is as

follows:

```
void setSimulationParameters(int& sTime, int& numOfServers, int& transTime, int& tBetweenCArrival)
{
        cout << "Enter the simulation time:
        "; cin >> sTime; cout << endl;

        cout << "Enter the number of servers:
        "; cin >> numOfServers; cout << endl;

        cout << "Enter the transaction time: "; cin
        >> transTime;
```

```
    cout << endl;

    cout << "Enter the time between customers arrival:
    "; cin >> tBetweenCArrival; cout << endl;
}
```

When a server becomes free and the customer queue is nonempty, we can move the customer at the front of the queue to the free server to be served. Moreover, when a customer starts the transaction, the waiting time ends. The waiting time of the customer is added to the total waiting time. The general algorithm to start the transaction (supposing that serverID denotes the ID of the free server) is as follows:

1. Remove the customer from the front of the queue.

   ```
   customer = customerQueue.front();
   customerQueue.deleteQueue();
   ```

2. Update the total waiting time by adding the current customer's waiting time to the previous total waiting time.

   ```
   totalWait = totalWait + customer.getWaitingTime();
   ```

3. Set the free server to begin the transaction.

   ```
   serverList.setServerBusy(serverID, customer, transTime);
   ```

To run the simulation, we need to know the number of customers arriving at a given time unit and how long it takes to serve the customer. We use the Poisson distribution from statistics, which says that the probability of y events occurring at a given time is given by the formula:

$$P(y) = \frac{\lambda^y e^{-\lambda}}{y!}, y = 0, 1, 2, \ldots,$$

where λ is the expected value that y events occur at that time. Suppose that, on average, a customer arrives every four minutes. During this four-minute period, the customer can arrive at any one of the four minutes. Assuming an equal likelihood of each of the four minutes, the expected value that a customer arrives in each of the four minutes is, therefore, 1 / 4 = 0.25.

Next, we need to determine whether the customer actually arrives at a given minute. Now $P(0) = e^{-\lambda}$ is the probability that no event occurs at a given time. One of the basic assumptions of the Poisson distribution is that the probability of more than one outcome occurring in a short time interval is negligible. For simplicity, we assume that only one customer arrives at a given time unit. Thus, we use $e^{-\lambda}$ as the cutoff point to determine whether a customer arrives at a given time unit. Suppose that, on average, a customer arrives every four minutes. Then $\lambda = 0.25$.We can use an algorithm to generate a number between 0 and 1. If the value of the number generated is $> e^{-0.25}$, we can assume that the customer arrived at a particular time unit. For example, suppose that rNum is a random number such that $0 \leq$ rNum ≤ 1. If rNum$> e^{-0.25}$, the customer arrived at the given time unit.

We now describe the function runSimulation to implement the simulation. Suppose that we run the simulation for 100 time units and customers arrive at time units 93, 96, and 100. The average transaction time is 5 minutes—that is, 5 time units. For simplicity, assume that we have only one server and the server becomes free at time unit 97, and that all customers arriving before time unit 93 have been served. When the server becomes free at time unit 97, the customer arriving at time unit 93 starts the transaction. Because the transaction of the customer arriving at time unit 93 starts at time unit 97 and it takes 5 minutes to complete a transaction, when the simulation loop ends, the customer arriving at time unit 93 is still at the server. Moreover, customers arriving at time units 96 and 100 are in the queue. For simplicity, we assume that when the simulation loop ends, the customers at the servers are considered served. The general algorithm for this function is as follows:

1. Declare and initialize the variables such as the simulation parameters, customer number, clock, total and average waiting times, number of customers arrived, number of customers served, number of customers left in the waiting queue, number of customers left with the servers, waitingCustomersQueue, and a list of servers.
2. The main loop is as follows:

   ```
   for (clock = 1; clock <= simulationTime; clock++)
   {
   ```

   2.1. Update the server list to decrement the transaction time of each busy server by one time unit.

   2.2. If the customer's queue is nonempty, increment the waiting time of each customer by one time unit.

   2.3. If a customer arrives, increment the number of customers by 1 and add the new customer to the queue.

   2.4. If a server is free and the customer's queue is nonempty, remove a customer from the front of the queue and send the customer to the free server.

   ```
   }
   ```

3. Print the appropriate results. Your results must include the number of customers left in the queue, the number of customers still with servers, the number of customers arrived, and the number of customers who actually completed a transaction.

Once you have designed the function runSimulation, the definition of the function main is simple and

straightforward because the function main calls only the function runSimulation.

When we tested our version of the simulation program, we generated the following results. We assumed that the average transaction time is 5 minutes and that on average a customer arrives every 4 minutes, and we used a random number generator to generate a number between 0 and 1 to decide whether a customer arrived at a given time unit.

Sample Run:

Customer number 1 arrived at time unit 4

Customer number 2 arrived at time unit 8

From server number 1 customer number 1 departed

at clock unit 9

Customer number 3 arrived at time unit 9

Customer number 4 arrived at time unit 12

From server number 1 customer number 2 departed
at clock unit 14
From server number 1 customer number 3 departed
at clock unit 19
Customer number 5 arrived at time unit 21
From server number 1 customer number 4 departed
at clock unit 24
From server number 1 customer number 5 departed
at clock unit 29
Customer number 6 arrived at time unit 37
Customer number 7 arrived at time unit 38
Customer number 8 arrived at time unit 41
From server number 1 customer number 6 departed
at clock unit 42
Customer number 9 arrived at time unit 43
Customer number 10 arrived at time unit 44
From server number 1 customer number 7 departed
at clock unit 47
Customer number 11 arrived at time unit 49
Customer number 12 arrived at time unit 51
From server number 1 customer number 8 departed
at clock unit 52
Customer number 13 arrived at time unit 52
Customer number 14 arrived at time unit 53
Customer number 15 arrived at time unit 54
From server number 1 customer number 9 departed
at clock unit 57
Customer number 16 arrived at time unit 59
From server number 1 customer number 10 departed
at clock unit 62
Customer number 17 arrived at time unit 66
From server number 1 customer number 11 departed
at clock unit 67
Customer number 18 arrived at time unit 71

From server number 1 customer number 12 departed
at clock unit 72
From server number 1 customer number 13 departed
at clock unit 77
Customer number 19 arrived at time unit 78
From server number 1 customer number 14 departed
at clock unit 82
From server number 1 customer number 15 departed
at clock unit 87
Customer number 20 arrived at time unit 90
From server number 1 customer number 16 departed
at clock unit 92
Customer number 21 arrived at time unit 92

From server number 1 customer number 17 departed
at clock unit 97

The simulation ran for 100 time units
Number of servers: 1
Average transaction time: 5
Average arrival time difference between customers: 4
Total waiting time: 269
Number of customers that completed a transaction: 17
Number of customers left in the servers: 1
The number of customers left in queue: 3
Average waiting time: 12.81
********** END SIMULATION **********

Key Terms

1. A queue is a data structure in which the items are added at one end and removed from the other end.
2. A queue is a First In First Out (FIFO) data structure.
3. The basic operations on a queue are as follows: Add an item to the queue, remove an item from the queue, retrieve the first and last element of the queue, initialize the queue, check whether the queue is empty, and check whether the queue is full.
4. A queue can be implemented as an array or a linked list.
5. The middle elements of a queue should not be accessed directly.
6. If the queue is nonempty, the function front returns the front element of the queue and the function back returns the last element in the queue.
7. Queues are restricted versions of arrays and linked lists.

Review Questions

1. Consider the following statements:

```
queueType<int> queue;
int x, y;
```

Show what is output by the following segment of code:

```
x = 4; y
= 5;
queue.addQueue(x);
queue.addQueue(y); x =
queue.front();
queue.deleteQueue();
queue.addQueue(x + 5);
queue.addQueue(16);
queue.addQueue(x);
queue.addQueue(y - 3);

cout << "Queue Elements: ";
while (!queue.isEmptyQueue())
{
```

```
        cout << queue.front() << " ";
        queue.deleteQueue();
    }
    cout << endl;
```

2. Consider the following statements:

```
    stackType<int> stack;
    queueType<int> queue;
    int x;
```

Suppose the input is:

```
    15 28 14 22 64 35 19 32 7 11 13 30 -999
```

Show what is written by the following segment of

```
    code: stack.push(0);
    queue.addQueue(0);
    cin >> x; while (x
    != -999)
    {
        switch (x % 4)
        { case 0: stack.push(x);
            break; case 1:
                if (!stack.isEmptyStack())
                {
                    cout << "Stack Element = " << stack.top() << endl; stack.pop();
                }
                else cout << "Sorry, the stack is empty." << endl;
                break; case
            2:
                queue.addQueue(x);
                break; case
            3:
                if (!queue.isEmptyQueue())
```

```
                    {
                        cout << "Queue Element = " << queue.front() << endl;
                        queue.deleteQueue();
                    }
                    else cout << "Sorry, the queue is empty." << endl;
break; } //end switch cin >> x; } //end while cout << "Stack Elements: ";
while (!stack.isEmptyStack())
{ cout << stack.top() << " ";
        stack.pop();
}
cout  <<  endl;  cout  <<
"Queue Elements: ";
while (!queue.isEmptyQueue())
{
        cout << queue.front() << " ";
```

```
        queue.deleteQueue();
    }
    cout << endl;
```

3. What does the following function do?

```
void mystery(queueType<int>& q)
{
    stackType<int>  s;  while
    (!q.isEmptyQueue())
    {
        s.push(q.front());
        q.deleteQueue();
    }
    while (!s.isEmptyStack())
    {
        q.addQueue(2 * s.top());
        s.pop();
    }
}
```

4. What is the effect of the following statements? If a statement is invalid, explain why it is invalid. The classes queueADT, queueType, and linkedQueueType are as defined in this chapter. a. queueADT<int> newQueue;

b. queueType <double> sales(-10);

c. queueType <string> names;

d. linkedQueueType <int> numQueue(50);

5. What is the output of the following program segment?

```
linkedQueueType<int> queue;
queue.addQueue(10);
queue.addQueue(20); cout <<
queue.front() << endl;
```

```
queue.deleteQueue();
queue.addQueue(2 * queue.back());
queue.addQueue(queue.front());
queue.addQueue(5);
queue.addQueue(queue.back() - 2);

linkedQueueType<int> tempQueue;
tempQueue = queue;
while (!tempQueue.isEmptyQueue())
{
        cout << tempQueue.front() << " ";
        tempQueue.deleteQueue();
}
```

```
cout << endl;

cout << queue.front() << " " << queue.back() << endl;
```

6. Suppose that queue is a queueType object and the size of the array implementing queue is 100. Also, suppose that the value of queueFront is 50 and the value of queueRear is 99.

a. What are the values of queueFront and queueRear after adding an element to queue?

b. What are the values of queueFront and queueRear after removing an element from queue?

7. Suppose that queue is a queueType object and the size of the array implementing queue is 100. Also, suppose that the value of queueFront is 99 and the value of queueRear is 25.

a. What are the values of queueFront and queueRear after adding an element to queue?

b. What are the values of queueFront and queueRear after removing an element from queue?

8. Suppose that queue is a queueType object and the size of the array implementing queue is 100. Also, suppose that the value of queueFront is 25 and the value of queueRear is 75.

a. What are the values of queueFront and queueRear after adding an element to queue?

b. What are the values of queueFront and queueRear after removing an element from queue?

9. Suppose that queue is a queueType object and the size of the array implementing queue is 100. Also, suppose that the value of queueFront is 99 and the value of queueRear is 99.

a. What are the values of queueFront and queueRear after adding an element to queue?

b. What are the values of queueFront and queueRear after removing an element from queue?

10. Suppose that queue is implemented as an array with the special reserved slot, as described in this chapter. Also, suppose that the size of the array implementing queue is 100. If the value of queueFront is 50, what is the position of the first queue element?

11. Suppose that queue is implemented as an array with the special reserved slot, as described in this chapter. Suppose that the size of the array implementing queue is 100. Also, suppose that the value of queueFront is 74 and the value of queueRear is 99.

a. What are the values of queueFront and queueRear after adding an element to queue?

b. What are the values of queueFront and queueRear after removing an element from queue? Also, what is the position of the removed queue element?

12. Write a function template, reverseQueue, that takes as a parameter a queue object and uses a stack object to reverse the elements of the queue.

13. Add the operation queueCount to the class queueType (the array implementation of queues), which returns the number of elements in the queue.

Write the definition of the function template to implement this operation.

14. Draw the UML diagram of the class queueADT.

15. Draw the UML diagram of the class queueType.

16. Draw the UML diagram of the class linkedQueueType.

7 TOPIC 7: SEARCHING AND HASHING ALGORITHMS



LEARNING OUTCOMES:

After studying this topic you should be able to:

- Learn the various search algorithms
- Explore how to implement the sequential and binary search algorithms
- Discover how the sequential and binary search algorithms perform
- Become aware of the lower bound on comparison-based search algorithms
- Learn about hashing

Chapter 3 described how to organize data using linked lists. The most important operation performed on a list is the search algorithm. Using the search algorithm, you can do the following:

- Determine whether a particular item is in the list.
- If the data is specially organized (for example, sorted), find the location in the list where a new item can be inserted.
- Find the location of an item to be deleted.

The search algorithm's performance, therefore, is crucial. If the search is slow, it takes a large amount of computer time to accomplish your task; if the search is fast, you can accomplish your task quickly.

7.1 Search Algorithms

This chapter discusses other search algorithms and analyzes them. The analysis of algorithms enables programmers to decide which algorithm to use for a specific application. Before describing these algorithms, let us make the following observations.

Associated with each item in a data set is a special member that uniquely identifies the item in the data set. For example, if you have a data set consisting of student records, then the student ID uniquely

identifies each student in a particular school. This unique member of the item is called the key of the item. The keys of the items in the data set are used in such operations as searching, sorting, insertion, and deletion. For instance, when we search the data set for a particular item, we compare the key of the item for which we are searching with the keys of the items in the data set.

As previously remarked, in addition to describing searching algorithms, this chapter analyses these algorithms. In the analysis of an algorithm, the key comparisons refer to comparing the key of the search item with the key of an item in the list. Moreover, the number of key comparisons refers to the number of times the key of the item (in algorithms such as searching and sorting) is compared with the keys of the items in the list.

This chapter refers to this class(class arrayListType), for easy reference we give its definition, without documentation to save space, here:

```
template <class elemType> class
arrayListType
{ public:
```

```
        const arrayListType<elemType>& operator=(const arrayListType<elemType>&);
        bool isEmpty() const; bool isFull() const; int listSize() const; int maxListSize()
        const; void print() const; bool isItemAtEqual(int location, const elemType&
        item) const; void insertAt(int location, const elemType& insertItem); void
        insertEnd(const elemType& insertItem); void removeAt(int location); void
        retrieveAt(int location, elemType& retItem) const; void replaceAt(int location,
        const elemType& repItem); void clearList();
        int seqSearch(const elemType& item) const;
        void insert(const elemType& insertItem); void
        remove(const elemType& removeItem);
        arrayListType(int size = 100);
        arrayListType(const arrayListType<elemType>& otherList); ~arrayListType();

    protected:
        elemType *list;      //array to hold the list elements
        int length;          //to store the length of the list
        int maxSize;         //to store the maximum size of the list
};
```

7.2 Sequential Search

The sequential search (also called linear search) on linked lists was covered in Chapter 3. The sequential search works the same for both array-based and linked lists. The search always starts at the first element in the list and continues until either the item is found in the list or the entire list is searched.

Because we are interested in the performance of the sequential search (that is, the analysis of this type of search), for easy reference and for the sake of completeness, we give the sequential search algorithm for array-based lists. If the search item is found, its index (that is, its location in the array) is returned. If the search is unsuccessful, -1 is returned. Note that the following sequential search does not require the list elements to be in any particular order.

```
template <class elemType>
int arrayListType<elemType>::seqSearch(const elemType& item) const
```

```
{
        int loc;
        bool found = false;

        for (loc = 0; loc < length; loc++)
                if (list[loc] == item)
                {
                        found = true;
                        break;
                } if
        (found) return
        loc; else
                return -1;
}                                       //end seqSearch
```

7.1.1.1 Sequential Search Analysis

This section analyzes the performance of the sequential search algorithm in both the worst case and the average case.

The statements before and after the loop are executed only once and, hence, require very little computer time. The statements in the for loop are the ones that are repeated several times. For each iteration of the loop, the search item is compared with an element in the list, and a few other statements are executed, including some other comparisons. Clearly, the loop terminates as soon as the search itemis found in the list. Therefore, the execution of the other statements in the loop is directly related to the outcome of the key comparison. Also, different programmers might implement the same algorithm differently, although the number of key comparisons would typically be the same. The speed of a computer can also easily affect the time an algorithm takes to perform, but not the number of key comparisons.

Therefore, when analyzing a search algorithm, we count the number of key comparisons because this number gives us the most useful information. Furthermore, the criteria for counting the number of key comparisons can be applied equally well to other search algorithms.

Suppose that *L* is list of length *n*. We want to determine the number of key comparisons made by the sequential search when the *L* is searched for a given item.

If the search item is not in the list, we then compare the search item with every element in the list, making n comparisons. This is an unsuccessful case.

Suppose that the search item is in the list. Then the number of key comparisons depends on where in the list the search item is located. If the search item is the first element of *L*, we make only one key comparison. This is the best case. On the other hand, if the search item is the last element in the list, the algorithm makes *n* comparisons. This is the worst case. The best and worst cases are not likely to occur every time we apply the sequential search on *L*, so it would be more helpful if we could determine the average behavior of the algorithm. That is, we need to determine the average number of key comparisons the sequential search algorithm makes in the successful case.

To determine the average number of comparisons in the successful case of the sequential search algorithm:

1. Consider all possible cases.
2. Find the number of comparisons for each case.
3. Add the number of comparisons and divide by the number of cases.

If the search item, called the target, is the first element in the list, one comparison is required. If the target is the second element in the list, two comparisons are required. Similarly, if the target is the *k*th element in the list, *k* comparisons are required. We assume that the target can be any element in the list; that is, all list elements are equally likely to be the target. Suppose that there are *n* elements in the list. The following expression gives the average number of comparisons:

$$\frac{1+2+\ldots+n}{n}$$

It is known that

$$1+2+\cdots+n=\frac{n(n+1)}{2}$$

Therefore, the following expression gives the average number of comparisons made by the sequential search in the successful case:

$$\frac{1+2+\ldots+n}{n}=\frac{1}{n}\frac{n(n+1)}{2}=\frac{n+1}{2}$$

This expression shows that, on average, the sequential search searches half the list. It, thus, follows that if the list size is 1,000,000, on average, the sequential search makes 500,000 comparisons. As a result, the sequential search is not efficient for large lists.

7.3 Ordered Lists

A list is ordered if its elements are ordered according to some criteria. The elements of a list are usually in ascending order. Several operations that can be performed on an ordered list are similar to the operations performed on an arbitrary list. For example, determining whether the list is empty or full, determining the length of the list, printing the list, and clearing the list for an ordered list are the same operations as those on an unordered list. Therefore, to define an ordered list as an abstract data type (ADT), by using the mechanism of inheritance, we can derive the class to implement the ordered lists from the class arrayListType discussed in the previous section. Depending on whether a specific application of a list can be stored in either an array or a linked list, we define two classes.

The following class, orderedArrayListType, defines an ordered list stored in an array as an ADT:

```
template <class elemType>
class orderedArrayListType: public arrayListType<elemType>
{ public: orderedArrayListType(int size = 100);
                //constructor
        …
                //We will add the necessary members as needed. private:
                //We will add the necessary members as needed.
}
```

Chapter 3 defined the following class to implement ordered linked lists:

```
template <class elemType>
```

```
class orderedLinkedListType: public linkedListType<elemType>
{
        public:
        ...
}
```

7.4 Binary Search

As you can see, the sequential search is not efficient for large lists because, on average, the sequential search searches half the list. We therefore describe another search algorithm, called the binary search, which is very fast. However, a binary search can be performed only on ordered lists. We, therefore, assume that the list is ordered. In the next chapter, we describe several sorting algorithms.

The binary search algorithm uses the divide-and-conquer technique to search the list. First, the search item is compared with the middle element of the list. If the search item is found, the search terminates. If the search item is less than the middle element of the list, we restrict the search to the first half of the list; otherwise, we search the second half of the list.

Consider the sorted list of length = 12 in Figure 7-1.

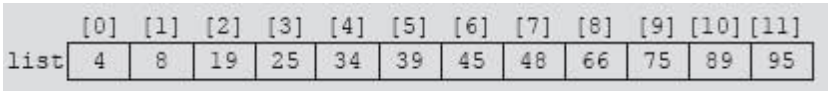


FIGURE 7-1 List of length 12

Suppose that we want to determine whether 75 is in the list. Initially, the entire list is the search list (see Figure 7-2).

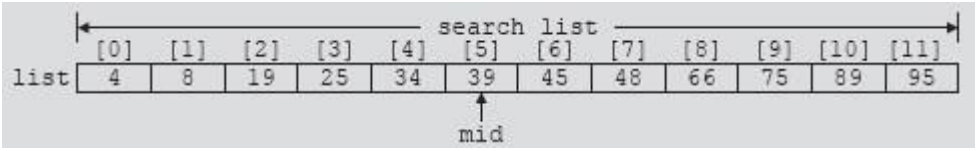


FIGURE 7-2 Search list, list[0]...list[11]

First, we compare 75 with the middle element in this list, list[5] (which is 39). Because 75 ≠ list[5] and 75 > list[5], we restrict our search to the list list[6]...list[11], as shown in Figure 7-3.

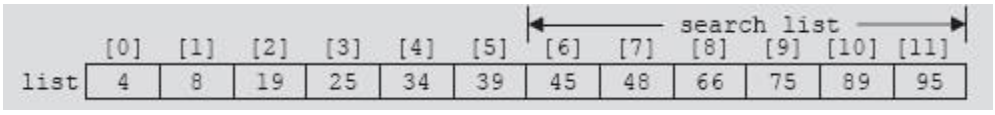


FIGURE 7-3 Search list, list[6]...list[11]

This process is now repeated on the list list[6]...list[11], which is a list of length = 6.

Because we need to determine the middle element of the list frequently, the binary search algorithm is typically implemented for array-based lists. To determine the middle element of the list, we add the starting index, first, and the ending index, last, of the search list and then divide by 2 to calculate its index.

That is, mid = (first + last) / 2.

Initially, first = 0 and last = length – 1 (this is because an array index in C++ starts at 0 and length denotes the number of elements in the list).

The following C++ function implements the binary search algorithm. If the item is found in the list, its location is returned; if the search item is not in the list, -1 is returned.

```
template<class elemType>
int orderedArrayListType<elemType>::binarySearch(const elemType& item) const
{
```

```
    int first = 0;      int
    last = length - 1;
    int mid;

    bool found = false;
    while (first <= last && !found)
    {
        mid = (first + last) / 2;

        if (list[mid] == item)
        found = true; else if
        (list[mid] > item) last =
        mid - 1; else
            first = mid + 1;
    }

    if (found) return
    mid; else
        return -1;
}                                   //end binarySearch
```

In the binary search algorithm, each time through the loop we make two key comparisons. The only exception is in the successful case; the last time through the loop only one key comparison is made.

Example 7-1 further illustrates how the binary search algorithm works.

EXAMPLE 7-1

Consider the list given in Figure 7-4.

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
list	4	8	19	25	34	39	45	48	66	75	89	95

FIGURE 7-4 Sorted list for a binary search

The number of elements in this list is 12, so length = 12. Suppose that we are searching for item 89. Table 7-1 shows the values of first, last, and mid each time through the loop. It also shows the number of times the item is compared with an element in the list each time through the loop.

TABLE 7-1: Values of first, last, and mid and the number of comparisons for search item 89

Iteration	first	last	Mid	list[mid]	Number of comparisons
1	0	11	5	39	2
2	6	11	8	66	2
3	9	11	10	89	1(found is true)

The item is found at location 10, and the total number of comparisons is 5.

Next, let us search the list for item 34. Table 7-2 shows the values of first, last, and mid each time through the loop. It also shows the number of times the item is compared with an element in the list each time through the loop.

TABLE 7-2: Values of first, last, and mid and the number of comparisons for search item 34

Iteration	first	last	mid	list[mid]	Number of comparisons
1	0	11	5	39	2
2	0	4	2	19	2
3	3	4	3	25	2
4	4	4	4	34	1 (found is true)

The item is found at location 4, and the total number of comparisons is 7.

Let us now search for item 22, as shown in Table 7-3.

TABLE 7-3: Values of first, last, and mid and the number of comparisons for search item 22

Iteration	first	last	mid	list[mid]	Number of comparisons
1	0	11	5	39	2
2	0	4	2	19	2
3	3	4	3	25	2
4	3	2	The loop stops (because first > last)		

This is an unsuccessful search. The total number of comparisons is 6.

7.1.1.2 Performance Of Binary Search

Suppose that *L* is a sorted list of size 1024 and we want to determine if an item *x* is in *L*. From the binary search algorithm, it follows that every iteration of the while loop cuts the size of the search list by half. (For example, see Figures 7-2 and 7-3.) Because $1024 = 2^{10}$, the while loop will have, at most, 11 iterations to determine whether *x* is in *L*. Because every iteration of the while loop makes two item (key) comparisons, that is, *x* is compared twice with the elements of *L*, the binary search will make, at most, 22 comparisons to determine whether *x* is in *L*. On the other hand, recall that a sequential search on average will make 512 comparisons to determine whether *x* is in *L*.

To better understand how fast binary search is compared with sequential search, suppose that *L* is of size

1048576. Because 1048576 = 2^{20}, it follows that the while loop in a binary search will have at most 21 iterations to determine whether an element is in *L*. Every iteration of the while loop makes two key (that is, item) comparisons. Therefore, to determine whether an element is in *L*, a binary search makes at most 42 item comparisons.

Note that $40 = 2 * 20 = 2 * \log_2 2^{20} = 2 * \log_2(1048576)$.

In general, suppose that *L* is a sorted list of size *n*. Moreover, suppose that *n* is a power of 2, that is, $n = 2^m$, for some nonnegative integer *m*. After each iteration of the for loop, about half the elements are left to search, that is, the search sublist for the next iteration is half the size of the current sublist. For example, after the first iteration, the search sublist is of size about $n/2 = 2^{m-1}$. It is easy to see that the maximum number of the iteration of the for loop is about $m + 1$. Also $m = \log_2 n$. Each iteration makes 2 key comparisons. Thus, the maximum number of comparisons to determine whether an element *x* is in *L* is $2(m + 1) = 2(\log_2 n + 1) = 2\log_2 n + 2$.

In the case of a successful search, it can be shown that for a list of length *n*, on average, a binary search makes $2\log_2 n - 3$ key comparisons. In the case of an unsuccessful search, it can be shown that for a list of length *n*, a binary search makes approximately $2\log_2(n+1)$ key comparisons.

Now that we know how to effectively search an ordered list stored in an array, let us discuss how to insert an item into an ordered list.

7.5 Insertion into an Ordered List

Suppose that you have an ordered list and want to insert an item in the list. After insertion, the resulting list must also be ordered. Chapter 3 described how to insert an item into an ordered linked list. This section describes how to insert an item into an ordered list stored in an array.

To store the item in the ordered list, first we must find the place in the list where the item is to be inserted. Then we slide the list elements one array position down to make room for the item to be inserted, and then we insert the item. Because the list is sorted and stored in an array, we can use an algorithm similar to the binary search algorithm to find the place in the list where the item is to be inserted. We can then use the function insertAt (of the class arrayListType) to insert the item. (Note that we cannot use the binary search algorithm as designed previously because it returns –1 if the item is not in the list. Of course, we can write another function using the binary search technique to find the position in the array where the item is to be inserted.) Therefore, the algorithm to insert the item is: (The special cases, such as inserting an item in an empty list or in a full list, are handled separately.)

1. Use an algorithm similar to the binary search algorithm to find the place where the item is to be

inserted.

2. if the item is already in this list

   output an appropriate message else use the function

   insertAt to insert the item in the list.

The following function, insertOrd, implements this algorithm.

```
template <class elemType>
void orderedArrayListType<elemType>::insertOrd(const elemType& item)
{ int first = 0; int last =
        length - 1; int mid;
        bool found = false;
        if (length == 0) //the list is empty
        {
                list[0] = item;
```

```
            length++;
        }
        else if (length == maxSize) cerr << "Cannot insert into
        a full list." << endl; else
        {
            while (first <= last && !found)
            { mid = (first + last) / 2; if
                (list[mid] == item)
                found = true; else if
                (list[mid] > item) last =
                mid - 1; else
                    first = mid + 1;
            }//end while if (found) cerr << "The insert item is
            already in the list. "
                    << "Duplicates are not allowed." << endl;
            else
            {
                if (list[mid] < item)
                mid++;
                insertAt(mid, item);
            }
        }

}//end insertOrd
```

Similarly, you can write a function to remove an element from an ordered list.

If we add the binary search algorithm and the insertOrd algorithm to the class orderedArrayListType, the definition of this class is as follows:

```
template <class elemType>
class orderedArrayListType: public arrayListType<elemType>
{ public:
```

```
        void insertOrd(const elemType&);
        int binarySearch(const elemType& item) const; orderedArrayListType(int
        size = 100);
};
```

Because the class orderedArrayListType is derived from the class arrayListType, and the list elements of an orderedArrayListType are ordered, we must override the functions insertAt and insertEnd of the class arrayListType in the class orderedArrayListType. We do this so that if these functions are used by an object of type orderedArrayListType, then after using these functions, the list elements of the object are still in order. We leave the details of these functions as an exercise for you. Furthermore, you can also override the function seqSearch so that while performing a sequential search on an ordered list, it takes into account that the elements are in order. We leave the details of this function also as an exercise.

Table 7-4 summarizes the algorithm analysis of the search algorithms discussed earlier.

TABLE 7-4 Number of comparisons for a list of length *n*

Algorithm	Successful search	Unsuccessful search
Sequential search	$(n + 1) / 2 = O(n)$	$n = O(n)$
Binary search	$2\log_2 n - 3 = O(\log_2 n)$	$2\log_2(n+1) = O(\log_2 n)$

7.2 Lower Bound on Comparison-Based Search Algorithms

Sequential and binary search algorithms search the list by comparing the target element with the list elements. For this reason, these algorithms are called comparison-based search algorithms. Earlier sections of this chapter showed that a sequential search is of the order n, and a binary search is of the order $\log_2 n$, where n is the size of the list. The obvious question is: Can we devise a search algorithm that has an order less than $\log_2 n$? Before we answer this question, first we obtain the lower bound on the number of comparisons for the comparison-based search algorithms.

Theorem: Let L be a list of size $n > 1$. Suppose that the elements of L are sorted. If SRH(n) denotes the minimum number of comparisons needed, in the worst case, by using a comparison-based algorithm to recognize whether an element x is in L, then SRH(n) _ log2(n + 1).

Corollary: The binary search algorithm is the optimal worst-case algorithm for solving search problems by the comparison method.

From these results, it follows that if we want to design a search algorithm that is of an order less than $\log_2$n, it cannot be comparison based.

7.6 Hashing

Previous sections of this chapter discussed two search algorithms: sequential and binary. In a binary search, the data must be sorted; in a sequential search, the data does not need to be in any particular order. We also analyzed both these algorithms and showed that a sequential search is of order n, and a binary search is of order $\log_2 n$, where n is the length of the list. The obvious question is: Can we construct a search algorithm that is of order less than $\log_2 n$? Recall that both search algorithms, sequential and binary, are comparisonbased algorithms. We obtained a lower bound on comparisonbased search

algorithms, which shows that comparison-based search algorithms are at least of order $\log_2 n$. Therefore, if we want to construct a search algorithm that is of order less than $\log_2 n$, it cannot be comparison based. This section describes an algorithm that, on average, is of order 1.

The previous section showed that for comparison-based algorithms, a binary search achieves the lower bound. However, a binary search requires the data to be specially organized, that is, the data must be sorted. The search algorithm that we now describe, called hashing, also requires the data to be specially organized.

In hashing, the data is organized with the help of a table, called the hash table, denoted by *HT*, and the hash table is stored in an array. To determine whether a particular item with a key, say *X*, is in the table, we apply a function *h*, called the hash function, to the key *X*; that is, we compute $h(X)$, read as *h* of *X*. The function *h* is typically an arithmetic function and $h(X)$ gives the address of the item in the hash table. Suppose that the size of the hash table, *HT*, is *m*. Then $0 \leq h(X) < m$. Thus, to determine whether the item

with key X is in the table, we look at the entry $HT[h(X)]$ in the hash table. Because the address of an item is computed with the help of a function, it follows that the items are stored in no particular order.

Before continuing with this discussion, let us consider the following questions:

- How do we choose a hash function?
- How do we organize the data with the help of the hash table?

First, we discuss how to organize the data in the hash table.

There are two ways that data is organized with the help of the hash table. In the first approach, the data is stored within the hash table, that is, in an array. In the second approach, the data is stored in linked lists and the hash table is an array of pointers to those linked lists. Each approach has its own advantages and disadvantages, and we discuss both approaches in detail. However, first we introduce some more terminology that is used in this section.

The hash table HT is, usually, divided into, say b buckets $HT[0], HT[1], \ldots, HT[b-1]$. Each bucket is capable of holding, say r items. Thus, it follows that $br = m$, where m is the size of HT. Generally, $r = 1$ and so each bucket can hold one item.

The hash function h maps the key X onto an integer t, that is, $h(X) = t$, such that $0 \le h(X) \le b - 1$.

EXAMPLE 7-2

Suppose there are six students a_1, a_2, a_3, a_4, a_5, a_6 in the Data Structures class and their IDs are a_1: 197354863; a_2: 933185952; a_3: 132489973; a_4: 134152056; a_5: 216500306; and a_6: 106500306.

Let $k_1 = 197354863$, $k_2 = 933185952$, $k_3 = 132489973$, $k_4 = 134152056$, $k_5 = 216500306$, and $k_6 = 106500306$.

Suppose that HT denotes the hash table and HT is of size 13 indexed 0, 1, 2, ..., 12.

Define the function h: $\{k_1, k_2, k_3, k_4, k_5, k_6\} \rightarrow \{0, 1, 2, \ldots, 12\}$ by $h(k_i) = k_i \% 13$. (Note that % denotes the mod operator.)

Now

$h(k_1) = h(197354863) = 197354863 \% 13 = 4$	$h(k_4) = h(134152056) = 134152056 \% 13 = 12$
$h(k_2) = h(933185952) = 933185952 \% 13 = 10$	$h(k_5) = h(216500306) = 216500306 \% 13 = 9$
$h(k_3) = h(132489973) = 132489973 \% 13 = 5$	$h(k_6) = h(106500306) = 106500306 \% 13 = 3$

Suppose $HT[b] \leftarrow a$ means "store the data of the student with ID a into $HT[b]$." Then

$HT[4] \leftarrow 197354863$	$HT[5] \leftarrow 132489973$	$HT[9] \leftarrow 216500306$
$HT[10] \leftarrow 933185952$	$HT[12] \leftarrow 134152056$	$HT[3] \leftarrow 106500306$

We consider now a slight variation of Example 7-2.

EXAMPLE 7-3

Suppose there are eight students in the class in a college and their IDs are 197354864, 933185952, 132489973, 134152056, 216500306, 106500306, 216510306, and 197354865. We want to store each student's data into HT in this order.

Let $k_1 = 197354864$, $k_2 = 933185952$, $k_3 = 132489973$, $k_4 = 134152056$, $k_5 = 216500306$, $k_6 = 106500306$, $k_7 = 216510306$, and $k_8 = 197354865$.

Suppose that HT denotes the hash table and HT is of size 13 indexed 0, 1, 2, . . ., 12.

Define the function h: $\{k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_8\} \rightarrow \{0, 1, 2, \ldots, 12\}$ by $h(k_i) = k_i \% 13$. Now

$h(k_1) = 197354864 \% 13 = 5$	$h(k_4) = 134152056 \% 13 = 12$	$h(k_7) = 216510306 \% 13 = 12$
$h(k_2) = 933185952 \% 13 = 10$	$h(k_5) = 216500306 \% 13 = 9$	$h(k_8) = 197354865 \% 13 = 6$
$h(k_3) = 132489973 \% 13 = 5$	$h(k_6) = 106500306 \% 13 = 3$	

As before, suppose $HT[b] \leftarrow a$ means "store the data of the student with ID a into $HT[b]$." Then

$HT[5] \leftarrow 197354864$	$HT[12] \leftarrow 134152056$	$HT[12] \leftarrow 216510306$
$HT[10] \leftarrow 933185952$	$HT[9] \leftarrow 216500306$	$HT[6] \leftarrow 197354865$
$HT[5] \leftarrow 132489973$	$HT[3] \leftarrow 106500306$	

It follows that the data of the student with ID 132489973 is to be stored in $HT[5]$. However, $HT[5]$ is already occupied by the data of the student with ID 197354864. In such a situation, we say that a *collision* has occurred. Later in this section, we discuss some ways to handle collisions.

Two keys, X_1 and X_2, such that $X_1 \neq X_2$, are called synonyms if $h(X_1)$ ¼ $h(X_2)$. Let X be a key and $h(X) = t$. If bucket t is full, we say that an overflow occurs. Let X_1 and X_2 be two nonidentical keys. If $h(X_1) = h(X_2)$, we say that a collision occurs. If $r = 1$, that is, the bucket size is 1, an overflow and a collision occur at the same time.

When choosing a hash function, the main objectives are to:

- Choose a hash function that is easy to compute. Minimize
- the number of collisions.

Next, we consider some examples of hash functions.

Suppose that *HTSize* denotes the size of the hash table, that is, the size of the array holding the hash table. We assume that the bucket size is 1. Thus, each bucket can hold one item and, therefore, overflow and collision occur simultaneously.

7.2.1 Hash Functions: Some Examples

Several hash functions are described in the literature. Here we describe some of the commonly used hash functions.

Mid-Square: In this method, the hash function, *h*, is computed by squaring the identifier, and then using the appropriate number of bits from the middle of the square to obtain the bucket address. Because the middle bits of a square usually depend on all the characters, it is expected that different keys will yield different hash addresses with high probability, even if some of the characters are the same.

Folding: In folding, the key *X* is partitioned into parts such that all the parts, except possibly the last parts, are of equal length. The parts are then added, in some convenient way, to obtain the hash address.

Division (Modular arithmetic): In this method, the key *X* is converted into an integer i_X. This integer is then divided by the size of the hash table to get the remainder, giving the address of *X* in *HT*. That is, (in C++) $h(X) = i_X$ % *HTSize*;

Suppose that each key is a string. The following C++ function uses the division method to compute the address of the key.

```
int hashFunction(char *insertKey, int keyLength)
{ int sum = 0;
        for (int j = 0; j < keyLength; j++) sum = sum +
                static_cast<int>(insertKey[j]);
        return (sum % HTSize);
}                                   // end hashFunction
```

7.2.2 Collision Resolution

As noted previously, the hash function that we choose not only should be easy to compute, but it is most desirable that the number of collisions is minimized. However, in reality, collisions are unavoidable because usually a hash function always maps a larger domain onto a smaller range. Thus, in hashing, we must include algorithms to handle collisions. Collision resolution techniques are classified into two categories: open addressing (also called closed hashing), and chaining (also called open hashing). In open addressing, the data is stored within the hash table. In chaining, the data is organized in linked lists and the hash table is an array of pointers to the linked lists. First we discuss collision resolution by open addressing.

7.2.3 Open Addressing

As described previously, in open addressing, the data is stored within the hash table. Therefore, for each key *X*, *h*(*X*) gives the index in the array where the item with key *X* is likely to be stored. Open addressing can be implemented in several ways. Next, we describe some of the common ways to implement it.

7.2.3.1 Linear Probing

Suppose that an item with key *X* is to be inserted in *HT*. We use the hash function to compute the index *h*(*X*) of this item in *HT*. Suppose that *h*(*X*) = *t*. Then 0 ≤ *h*(*X*) ≤ *HTSize* – 1. If *HT*[*t*] is empty, we store this item into this array slot. Suppose that *HT*[*t*] is already occupied by another item; we have a collision. In linear probing, starting at location *t*, we search the array sequentially to find the next available array slot. In linear probing, we assume that the array is circular so that if the lower portion of the array is full, we can continue the search in the top portion of the array. This can be easily accomplished by using the mod

operator. That is, starting at *t*, we check the array locations *t*, (*t* + 1) % *HTSize*, (*t* + 2) % *HTSize*, . . ., (*t* + *j*) % *HTSize*. This is called the probe sequence.

The next array slot is given by

$$(h(X) + j) \% HTSize$$

where *j* is the *j*th probe.

EXAMPLE 7-4

Consider the students' IDs and the hash function given in Example 7-3. Then we know that

$h(197354864) = 5 = h(132489973)$	$h(134152056) = 12 = h(216510306)$	$h(106500306) = 3$
$h(933185952) = 10$	$h(216500306) = 9$	$h(197354865) = 6$

Using the linear probing, the array position where each student's data is stored is:

ID	*h*(ID)	(*h*(ID) + 1) % 13	(*h*(ID) + 2) % 13
197354864	5		
933185952	10		
132489973	5	6	
134152056	12		
216500306	9		
106500306	3		
216510306	12	0	
197354865	6	7	

As before, suppose *HT*[*b*] ← *a* means ''store the data of the student with ID *a* into *HT*[*b*+.'' Then

HT[5] ← 197354864	*HT*[12] ← 134152056	*HT*[0] ← 216510306
HT[10] ← 933185952	*HT*[9] ← 216500306	*HT*[7] ← 197354865
HT[6] ← 132489973	*HT*[3] ← 106500306	

The following C++ code implements linear probing:

```
hIndex = hashFunction(insertKey);
found = false;
while (HT[hIndex] != emptyKey && !found) if
        (HT[hIndex].key == key)
                found = true;
        else hIndex = (hIndex + 1) % HTSize;
```

```
if (found) cerr << "Duplicate items are not allowed." <<
endl; else
        HT[hIndex] = newItem;
```

From the definition of linear probing, we see that linear probing is easy to implement. However, linear probing causes clustering; that is, more and more new keys would likely be hashed to the array slots that are already occupied. For example, consider the hash table of size 20, as shown in Figure 7-5.

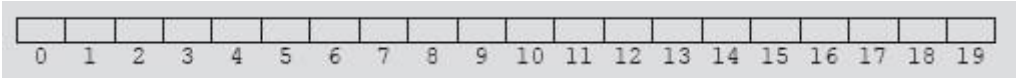


FIGURE 7-5 Hash table of size 20

Initially, all the array positions are available. Because all the array positions are available, the probability of any position being probed is (1/20). Suppose that after storing some of the items, the hash table is as shown in Figure 7-6.

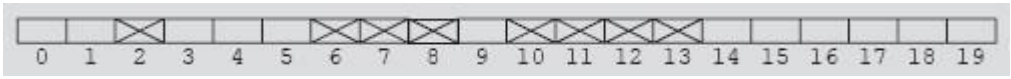


FIGURE 7-6 Hash table of size 20 with certain positions occupied

In Figure 7-6, a cross indicates that this array slot is occupied. Slot 9 will be occupied next if, for the next key, the hash address is 6, 7, 8, or 9. Thus, the probability that slot 9 will be occupied next is 4/20. Similarly, in this hash table, the probability that array position 14 will be occupied next is 5/20.

Now consider the hash table of Figure 7-7.

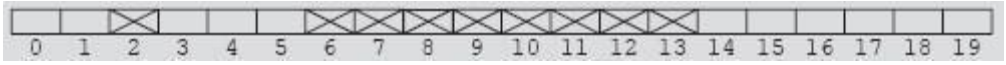


FIGURE 7-7 Hash table of size 20 with certain positions occupied

In this hash table, the probability that the array position 14 will be occupied next is 9/20, whereas the probability that the array positions 15, 16, or 17 will be occupied next is 1/20. We see that items tend to cluster, which would increase the search length. Linear probing, therefore, causes clustering. This clustering is called primary clustering.

One way to improve linear probing is to skip array positions by a fixed constant, say *c*, rather than 1. In this case, the hash address is as follows:

$$(h(X) + i * c) \% HTSize$$

If $c = 2$ and $h(X) = 2k$, that is, $h(X)$ is even, only the even-numbered array positions are visited. Similarly, if $c = 2$ and $h(X) = 2k + 1$, that is, $h(X)$ is odd, only the odd-numbered array positions are visited. To visit all the array positions, the constant c must be relatively prime to *HTSize*.

7.2.3.2 Random Probing

This method uses a random number generator to find the next available slot. The *i*th slot in the probe sequence is

$$(h(X) + r_i) \% HTSize$$

where r_i is the *i*th value in a random permutation of the numbers 1 to *HTSize* – 1. All insertions and searches use the same sequence of random numbers.

EXAMPLE 7-5

> Suppose that the size of the hash table is 101, and for the keys X_1 and X_2, $h(X_1) = 26$ and $h(X_2) = 35$. Also suppose that $r_1 = 2$, $r_2 = 5$, and $r_3 = 8$. Then the probe sequence of X_1 has the elements 26, 28, 31, and 34. Similarly, the probe sequence of X_2 has the elements 35, 37, 40, and 43.

7.7 Rehashing

In this method, if a collision occurs with the hash function *h*, we use a series of hash functions, *h*1, *h*2, . . ., h_s. That is, if the collision occurs at *h*(*X*), the array slots $h_i(X)$, $1 \le h_i(X) \le s$ are examined.

7.2.3.3 Quadratic Probing

Suppose that an item with key *X* is hashed at *t*, that is, *h*(*X*) = *t* and $0 \le t \le HTSize - 1$. Further suppose that position *t* is already occupied. In quadratic probing, starting at position *t,* we linearly search the array at locations (*t* + 1)% *HTSize*, $(t + 2^2) \% HTSize = (t + 4) \% HTSize$, $(t + 3^2) \% HTSize = (t + 9) \% HTSize$, . . ., $(t + i^2) \% HTSize$. That is, the probe sequence is: *t*, (*t* + 1) % *HTSize* $(t + 2^2) \% HTSize$, $(t + 3^2) \% HTSize$, . . ., $(t + i^2) \% HTSize$.

EXAMPLE 7-6

Suppose that the size of the hash table is 101 and for the keys X_1, X_2, and X_3, $h(X_1) = 25$, $h(X_2) = 96$, and $h(X_3) = 34$. Then the probe sequence for X_1 is 25, 26, 29, 34, 41, and so on. The probe sequence for X_2 is 96, 97, 100, 4, 11, and so on. (Notice that $(96 + 3^2)$ % $101 = 105$ % $101 = 4$.)

The probe sequence for X_3 is 34, 35, 38, 43, 50, 59, and so on. Even though element 34 of the probe sequence of X_3 is the same as the fourth element of the probe sequence of X_1, both probe sequences after 34 are different.

Although quadratic probing reduces primary clustering, we do not know if it probes all the positions in the table. In fact, it does not probe all the positions in the table. However, when *HTSize* is a prime, quadratic probing probes about half the table before repeating the probe sequence. Let us prove this observation.

Suppose that *HTSize* is a prime and for $0 \le i < j \le HTSize$,

$$(t + i^2)\%HTSize = (t + j^2)\%HTSize:$$

This implies that *HTSize* divides ($j^2 - i^2$), that is, *HTSize* divides ($j - i$) ($j + i$). Because *HTSize* is a prime, we get *HTSize* divides ($j - i$) or *HTSize* divides ($j + i$).
Now because $0 < j - i < HTSize$, it follows that *HTSize* does not divide ($j - i$). Hence, *HTSize* divides ($j + i$). This implies that $j + i \geq HTSize$, so $j \geq (HTSize / 2)$.

Hence, quadratic probing probes half the table before repeating the probe sequence. Thus, it follows that if the size of *HTSize* is a prime at least twice the number of items, we can resolve all the collisions.

Because probing half the table is already a considerable number of probes, after making these many probes we assume that the table is full and stop the insertion (and search). (This can occur when the table is actually half full; in practice, it seldom happens unless the table is nearly full.)

Next we describe how to generate the probe sequence.

Note that

$2^2 = 1 + (2 * 2 - 1)$

$3^2 = 1 + 3 + (2 * 3 - 1)$ 4^2

$= 1 + 3 + 5 + (2 * 4 - 1)$

.

.

. $i^2 = 1 + 3 + 5 + 7 + \ldots + (2 * i - 1)$; $i \geq$

1:

Thus, it follows that

$$(t + i^2)\%HTSize = (t + 1 + 3 + 5 + 7 + \ldots + (2 * i - 1))\%\text{HTSize}$$

Consider the probe sequence t, $t + 1$, $t + 2^2$, $t + 3^2$, . . . , $(t + i^2)$ % *HTSize*. The following C++ code computes the ith probe, that is, $(t + i^2)$ % *HTSize*:

```
int inc = 1;
int pCount = 0;
```

```
while (p < i)
{ t = (t + inc) % HTSize; inc =
        inc + 2; pCount++;
}
```

The following pseudocode implements quadratic probing (assume that *HTSize* is a prime):

```
int pCount; int
inc;
int hIndex;
hIndex = hashFunction(insertKey);
pCount = 0;
```

```
inc = 1;

while (HT[hIndex] is not empty && HT[hIndex] is not the same as the insert item
                && pCount < HTSize / 2)
{
        pCount++;
        hIndex = (hIndex + inc ) % HTSize;
        inc = inc + 2;
}

if (HT[hIndex] is empty)
        HT[hIndex] = newItem; else if (HT[hIndex] is the
same as the insert item) cerr << "Error: No duplicates are
allowed." << endl; else cerr << "Error: The table is full. "
                << "Unable to resolve the collisions." << endl;
```

Both randomand quadratic probings eliminate primary clustering. However, if two nonidentical keys, say X_1 and X_2, are hashed to the same home position, that is, $h(X_1) = h(X_2)$, then the same probe sequence is followed for both keys. The same probe sequence is used for both keys because random probing and quadratic probing are functions of the home positions, not the original key. It follows that if the hash function causes a cluster at a particular home position, the cluster remains under these probings. This is called secondary clustering.

One way to solve secondary clustering is to use linear probing, with the increment value a function of the key. This is called double hashing. In double hashing, if a collision occurs at $h(X)$, the probe sequence is generated by using the rule:

$$(h(X) + i * g(X)) \% HTSize$$

where g is the second hash function, and $i = 0, 1, 2, 3, \ldots$.

If the size of the hash table is a prime p, then we can define g as follows:

$g(k) = 1+(k \% (p - 2))$

EXAMPLE 7-7

Suppose that the size of the hash table is 101 and for the keys X_1 and X_2, $h(X_1) = 35$ and $h(X_2) = 83$. Also suppose that $g(X_1) = 3$ and $g(X_2) = 6$. Then the probe sequence for X_1 is 35, 38, 41, 44, 47, and so on. The probe sequence for X_2 is 83, 89, 95, 0, 6, and so on. (Notice that (83 + 3 * 6) % 101 = 101 % 101 = 0.)

EXAMPLE 7-8

Suppose there are six students in the Data Structures class and their IDs are 115, 153, 586, 206, 985, and 111, respectively. We want to store each student's data in this order. Suppose HT is of the size 19 indexed 0,1,2,3, . . ., 18. Consider the prime number $p = 19$. Then $p - 2 = 17$. For the ID k, we define the hashing functions:

$h(k) = k \% 19$ and $g(k) = 1 + (k \% (p - 2)) = 1 + (k \% 17)$

Let $k = 115$. Now $h(115) = 115 \% 19 = 1$. So the data of the student with ID 115 is stored in $HT[1]$.

Next consider $k = 153$. Now $h(153) = 153 \% 19 = 1$. However, $HT[1]$ is already occupied. So we first calculate $g(153)$, to find the probe sequence of 153. Now $g(153) = 1 + (153 \% 17) = 1 + 0 = 1$. Thus, $h(153) = 1$ and $g(153) = 1$. Therefore, probe sequence of 153 is given by $(h(153) + i \cdot g(153)) \% 19 = (1 + i \cdot 1) \% 19$, $i = 0, 1, 2, 3, \ldots$. Hence, the probe sequence of 153 is 1, 2, 3, ... Because $HT[2]$ is empty, the data of the student with ID 153 is stored in $HT[2]$.

Consider $k = 586$. Now $h(586) = 586 \% 19 = 16$. Because $HT[16]$ is empty, we store the data of the student with ID 586 in $HT[16]$.

Consider $k = 206$. Now $h(206) = 206 \% 19 = 16$. Because $HT[16]$ is already occupied, we compute $g(206)$. Now $g(206) = 1 + (206 \% 17) = 1 + 2 = 3$. So the probe sequence of 206 is, 16, 0, 3, 6, ... Note that $(16 + 3) \% 19 = 0$. Because $HT[0]$ is empty, the data of the student with ID 206 is stored in $HT[0]$.

We apply this process and find the array position to store the data of each student. If a collision occurs for an ID, then the following table shows the probe sequence of that ID.

ID	h(ID)	g(ID)	Probe sequence	
115	1			
153	1	1	1, 2, 3, 4, 5, . . .	
586	16			
206	16	3	16, 0, 3, 6, 9,	Note that (16 + 3) % 19 = 0
985	16	17	16, 14, 12, 10, . . .	Note that (16 + 17) % 19 = 14
111	16	10	16, 7, 17, 8, . . .	

As before, suppose $HT[b] \leftarrow a$ means "store the data of the student with ID a into $HT[b]$." Then

$HT[1] \leftarrow 115$	$HT[16] \leftarrow 586$	$HT[14] \leftarrow 985$
$HT[2] \leftarrow 153$	$HT[0] \leftarrow 206$	$HT[7] \leftarrow 111$

7.2.4 Deletion: Open Addressing

Suppose that an item, say *R*, is to be deleted from the hash table, *HT.* Clearly, we first must find the index

of *R* in *HT*. To find the index of *R*, we apply the same criteria we applied to *R* when *R* was inserted in *HT*. Let us further assume that after inserting *R*, another item, *R*', was inserted in *HT*, and the home position

of *R* and *R*' is the same. The probe sequence of *R* is contained in the probe sequence of *R*' because *R*' was inserted in the hash table after *R*. Suppose that we delete *R* simply by marking the array slot containing *R* as empty. If this array position stays empty, then while searching for *R*' and following its probe sequence, the search terminates at this empty array position. This gives the impression that *R*' is not in the table, which, of course, is incorrect. The item *R* cannot be deleted simply by marking its position as empty from the hash table.

One way to solve this problem is to create a special key to be stored in the key of the items to be deleted. The special key in any slot indicates that this array slot is available for a new item to be inserted. However, during the search, the search should not terminate at this location. This, unfortunately, makes the deletion algorithm slow and complicated.

Another solution is to use another array, say indexStatusList of int, of the same size as the hash table as follows: Initialize each position of indexStatusList to 0, indicating that the corresponding position in the hash table is empty. When an item is added to the hash table at position, say i, we set indexStatusList[i] to 1. When an item is deleted from the hash table at position, say k, we set indexStatusList[k] to -1. Therefore, each entry in the array indexStatusList is -1, 0, or 1.

For example, suppose that you have the hash table as shown in Figure 7-8.

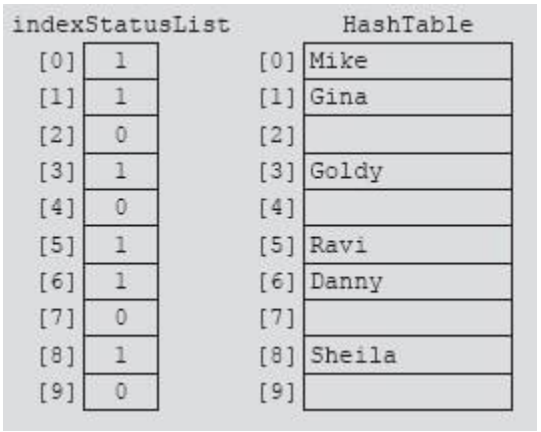
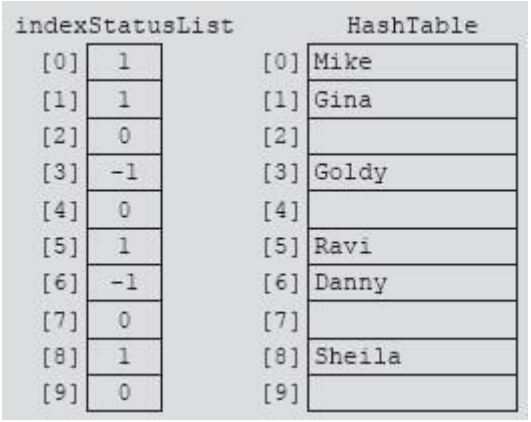


FIGURE 7-8 Hash table and indexStatusList

In Figure 7-8, the hash table positions 0, 1, 3, 5, 6, and 8 are occupied. Suppose that the entries at positions 3 and 6 are removed. To remove these entries from the hash table, we store -1 at positions 3 and 6 in the array indexStatusList (see Figure 7-9).



	indexStatusList		HashTable
[0]	1	[0]	Mike
[1]	1	[1]	Gina
[2]	0	[2]	
[3]	-1	[3]	Goldy
[4]	0	[4]	
[5]	1	[5]	Ravi
[6]	-1	[6]	Danny
[7]	0	[7]	
[8]	1	[8]	Sheila
[9]	0	[9]	

FIGURE 7-9 Hash table and indexStatusList after removing the entries at positions 3 and 6

7.2.5 Hashing: Implementation Using Quadratic Probing

Better behaviour is usually obtained with **quadratic probing**, where the secondary hash function depends on the re-hash index:

7.7.1.1 address = h(key) + c i^2

on the t^{th} re-hash. (A more complex function of ***i*** may also be used.) Since keys which are mapped to the same value by the primary hash function follow the same sequence of addresses, quadratic probing shows **secondary clustering**. However, secondary clustering is not nearly as severe as the clustering shown by linear probes.

Re-hashing schemes use the originally allocated table space and thus avoid linked list overhead, but require advance knowledge of the number of items to be stored.

However, the collision elements are stored in slots to which other key values map directly, thus the potential for multiple collisions increases as the table becomes full. This method uses the hash function of the form

$h(k, i) = (h`(k) + c_1 i + c_2 i^2) \bmod m \quad \text{for } i = 0, 1, 2, \ldots, m\text{-}1$

where h` is an auxiliary hash function. $c_1 + c_2 \neq 0$ are auxiliary constants.

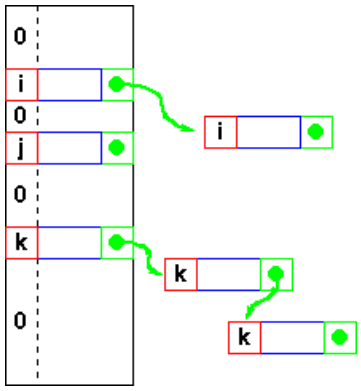
This method works much better than linear probing. Quadratic probing suffers a milder form of clustering, called secondary clustering.

7.8 Chaining

One simple scheme is to chain all collisions in lists attached to the appropriate slot. This allows an unlimited number of collisions to be handled and doesn't require *a priori* knowledge of how many elements are contained in the collection. The tradeoff is the same as with linked lists versus array implementations of collections: linked list overhead in space and, to a lesser extent, in time.

7.2.5.1 Collision Resolution by Chaining

When there is a collision (keys hash to the same slot), the incoming keys is stored in an overflow area and the corresponding record is appeared at the end of the linked list.



Each slot T[j] contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_n)$ and $h(k_5) = h(k_2) = h(k_7)$.

- The worst case running time for insertion is O(1).
- Deletion of an element x can be accomplished in O(1) time if the lists are doubly linked.
- In the worst case behavior of chain-hashing, all n keys hash to the same slot, creating a list of length n.

The worst-case time for search is thus θ(n) plus the time to compute the hash function.

keys: 5,28,19,15,20,33,12,17,10 slots: 9

hash function = h(k) = k mod 9

h(5) = 5 mod 9 = 4 h(28)

= 28 mod 9 = 1 h(19) =

19 mod 9 = 1 h(15) = 15

mod 9 = 6 h(20) = 20

mod 9 = 2 h(33) = 33

mod 9 = 6 h(12) = 12

mod 9 = 3 h(17) = 17

mod 9 = 8 h(10) = 10

mod 9 = 1

A good hash function satisfies the assumption of simple uniform hashing, each element is equally likely to hash into any of the m slots, independently of where any other element has hash to. But usually it is not possible to check this condition because one rarely knows the probability distribution according to which the keys are drawn.

In practice, we use heuristic techniques to create a hash function that perform well. One good approach is to derive the hash value in a way that is expected to be independent of any patterns that might exist in the data (division method).

Most hash function assume that the universe of keys is the set of natural numbers. Thus, its keys are not natural to interpret than as natural numbers.

7.2.5.2 Method for Creating Hash Function

1. The division method.
2. The multiplication method.
3. Universal hashing.

7.8.1.1 The Division Method

Map a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is h(k) = k mod m.

Example:

If table size m = 12

key k = 100

than

h(100) = 100 mod 12

= 4

Poor choices of *m* m should not be a power of 2, since if m = 2p, then h(k) is just the p lowest-order bits of k. So, 2p may be a poor choice, because permuting the characters of k does not change value.

Good m choice of *m*

A prime not too close to an exact of 2.

7.8.2 The Multiplication Method

Two step process

Step 1: Multiply the key *k* by a constant 0< A < 1 and extract the fraction part of *k*A. Step 2: Multiply kA by m and take the floor of the result.

The hash function using multiplication method is:

h(*k*) = ëm(*k*A mod 1)û

where "*k*A mod 1" means the fractional part of *k*A, that is, *k*A - ë*k*Aû.

Advantage of this method is that the value of m is not critical and can be implemented on most computers.

A reasonable value of constant A is

» (sqrt5 - 1) /2

7.8.3 Universal Hashing

Open Addressing

This is another way to deal with collisions.

In this technique all elements are stored in the hash table itself. That is, each table entry contains either an element or NIL. When searching for element (or empty slot), we systematically examine slots until we found an element (or empty slot). There are no lists and no elements stored outside the table. That implies that table can completely "fill up"; the load factor α can never exceed 1.Advantage of this technique is

that it avoids pointers (pointers need space too). Instead of chasing pointers, we compute the sequence of slots to be examined. To perform insertion, we successively examine or probe, the hash table until we find an empty slot. The sequence of slots probed "depends upon the key being inserted." To determine which slots to probe, the hash function includes the probe number as a second input.

Thus, the hash function becomes

h: $\mathcal{U}$ × {0, 1, . . . m -1 }--> {0, 1, . . . , m-1}

and the probe sequence

< h(k, 0), h(k, 1), . . . , h(k, m-1) >

in which every slot is eventually considered.

Exercise

1. Suppose there are eight students with IDs 197354883, 933185971, 132489992, 134152075, 216500325, 106500325, 216510325, 197354884. Suppose hash table, HT, is of the size 19, indexed 0, 1, 2, . . ., 18. Show how these students' IDs, in the order given, are inserted in HT using the hashing function h(k) = k % 19. Use linear probing to resolve collision.

2. Suppose there are six workers, in a workshop, with IDs 147, 169, 580, 216, 974, and 124. Suppose hash table, HT, is of the size 13, indexed 0, 1, 2, . . ., 12. Show how these workers' IDs, in the order given, are inserted in HT using the hashing function h(k) = k % 13. Use linear probing to resolve collision.

3. Suppose there are five workers, in a shop, with IDs 909, 185, 657, 116, and 150. Suppose hash table, HT, is of the size 7, indexed 0, 1, 2, . . ., 6. Show how these workers' IDs, in the order given, are inserted in HT using the hashing function h(k) = k % 7. Use linear probing to resolve collision.

4. Suppose there are seven students with IDs 5701, 9302, 4210, 9015, 1553, 9902, and 2104. Suppose hash table, HT, is of the size 19, indexed 0,1,2, . . ., 18. Show how these students' IDs, in the order given, are inserted in HT using the hashing function h(k) = k % 19. Use double hashing to resolve collision, where the second hash function is given by g(k) = (k+1) % 17.

5. Suppose that an item is to be removed from a hash table that was implemented using linear or quadratic probing. Why wouldn't you mark the position of the item to be deleted as empty?

6. What are the advantages of open hashing?

7. Give a numerical example to show that collision resolution by quadratic probing is better than chaining.

8. Give a numerical example to show that collision resolution by chaining is better than quadratic probing.

9. Suppose that the size of the hash table is 1001 and the table has 850 items. What is the load factor?

10. Suppose that the size of the hash table is 1001 and the table has 750 items. On average, how many comparisons are made to determine whether an item is in the list if: a. Linear probing is used.

b. Quadratic probing is used.

c. Chaining is used.

11. Suppose that 550 items are to be stored in a hash table. If, on average, three key comparisons are needed to determine whether an item is in the table, what should be the size of the hash table if: a. Linear probing is used.

b. Quadratic probing is used.

c. Chaining is used.

8 TOPIC 8: SORTING ALGORITHMS



LEARNING OUTCOMES:

After studying this topic you should be able to:

- Learn the various sorting algorithms
- Explore how to implement selection sort, insertion sort, Shellsort, quicksort, mergesort, and heapsort
- Discover how the sorting algorithms discussed in this chapter perform
- Learn how priority queues are implemented

Chapter 7 discussed the search algorithms on lists. A sequential search does not assume that the data is in any particular order; however, as noted, this search does not work efficiently for large lists. By contrast, a binary search is very fast for array-based lists, but it requires the data to be in order. Because a binary search requires the data to be in order and its performance is good for array-based lists, this chapter focuses on sorting algorithms.

8.1 Sorting Algorithms

There are several sorting algorithms in the literature. In this chapter, we discuss some of the most commonly used sorting algorithms. To compare the performance of these algorithms, we also provide the analysis of these algorithms. These sorting algorithms can be applied to either array-based lists or linked lists.We will specify whether the algorithm being developed is for array-based lists or linked lists.

The functions implementing these sorting algorithms are included as public members of the related class. (For example, for an array-based list, these are the members of the class arrayListType.) By doing so, the algorithms have direct access to the list elements.

Suppose that the sorting algorithm selection sort (described in the next section) is to be applied to array-based lists. The following statements show how to include selection sort as a member of the class arrayListType:

```
template <class elemType> class
```

```
arrayListType
{ public:
        void selectionSort();
    ...
};
```

8.1 Selection Sort: Array-Based Lists

In selection sort, a list is sorted by selecting elements in the list, one at a time, and moving them to their proper positions. This algorithm finds the location of the smallest element in the unsorted portion of the list and moves it to the top of the unsorted portion (that is, the whole list) of the list. The first time we locate the smallest item in the entire list, the second time we locate the smallest item in the list starting from the second element in the list, and so on. Selection sort described here is designed for array-based lists.

Suppose you have the list shown in Figure 8-1.

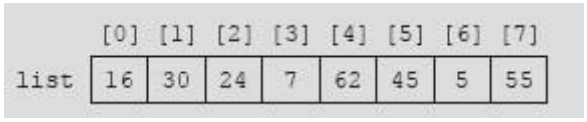


FIGURE 8-1 List of 8 elements

Figure 8-2 shows the elements of list in the first iteration.

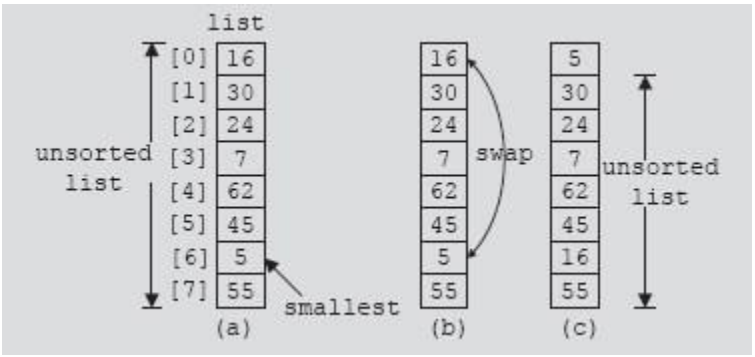


FIGURE 8-2 Elements of list during the first iteration

Initially, the entire list is unsorted. So we find the smallest item in the list. The smallest item is at position 6, as shown in Figure 8-2(a). Because this is the smallest item, it must be moved to position 0. So we swap 16 (that is, list[0]) with 5 (that is, list[6]), as shown in Figure 8-2(b). After swapping these elements, the resulting list is as shown in Figure 8-2(c).

Figure 8-3 shows the elements of list in the second iteration.

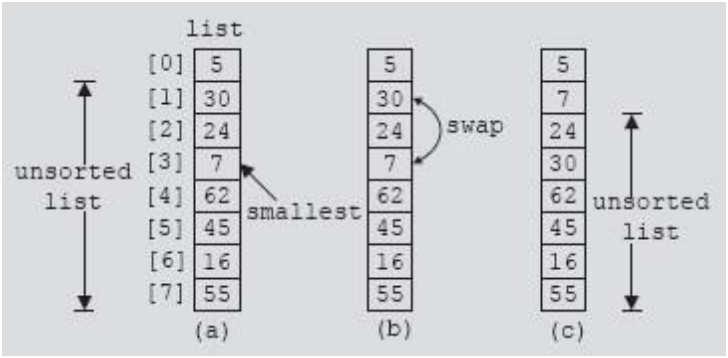


FIGURE 8-3 Elements of list during the second iteration

Now the unsorted list is list[1]...list[7]. So we find the smallest element in the unsorted list. The smallest element is at position 3, as shown in Figure 8-3(a). Because the smallest element in the unsorted list is at position 3, it must be moved to position 1. So we swap 7 (that is, list[3]) with 30 (that is, list[1]), as shown

in Figure 8-3(b). After swapping list[1] with list[3], the resulting list is as shown in Figure 8-3(c).

Now the unsorted list is list[2]...list[7]. So we repeat the preceding process of finding the (position of the) smallest element in the unsorted portion of the list and moving it to the beginning of the unsorted portion of the list. Selection sort, thus, involves the following steps.

In the unsorted portion of the list:

1. Find the location of the smallest element.
2. Move the smallest element to the beginning of the unsorted list.

Initially, the entire list, list[0]...list[length-1], is the unsorted list. After executing Steps 1 and 2 once, the unsorted list is list[1]...list[length-1]. After executing Steps 1 and 2 a second time, the unsorted list is list[2]...list[length-1], and so on. We can keep track of the unsorted portion of the list and repeat Steps a and b with the help of a for loop as follows:

```
for (index = 0; index < length - 1; index++)
{
        1. Find the location, smallestIndex, of the smallest element in list[index]...list[length - 1].
        2. Swap the smallest element with list[index]. That is, swap list[smallestIndex] with
           list[index].
}
```

The first time through the loop, we locate the smallest element in list[0]...list[length-1] and swap this smallest element with list[0]. The second time through the loop, we locate the smallest element in list[1]...list[length-1] and swap this smallest element with list[1], and so on. This process continues until the length of the unsorted list is 1. (Note that a list of length 1 is sorted.) It, therefore, follows that to implement selection sort, we need to implement Steps 1 and 2.

Given the starting index, first, and the ending index, last, of the list, the following C++ function returns the index of the smallest element in list[first]...list[last]:

```
template <class elemType>
int arrayListType<elemType>::minLocation(int first, int last)
{
        int minIndex;

        minIndex = first;
        for (int loc = first + 1; loc <= last; loc++) if(
                list[loc] < list[minIndex])
                minIndex = loc;
```

```
        return minIndex;
} //end minLocation
```

Given the locations in the list of the elements to be swapped, the following C++ function, swap, swaps those elements:

```
template <class elemType>
void arrayListType<elemType>::swap(int first, int second)
{
        elemType temp;

        temp = list[first]; list[first]
        = list[second];
        list[second] = temp;
}//end swap
```

We can now complete the definition of the function selectionSort:

```
template <class elemType>
void arrayListType<elemType>::selectionSort()
{
        int minIndex;
        for (int loc = 0; loc < length - 1; loc++)
        {
                minIndex = minLocation(loc, length - 1); swap(loc,
                minIndex);
        }
}
```

You can add the functions to implement selection sort in the definition of the class arrayListType as follows:

```
template<class elemType> class
arrayListType
{ public:
                //Place the definitions of the function given earlier here. void
                selectionSort();
                ... private:
                //Place the definitions of the members given earlier here.
                void swap(int first, int second); int minLocation(int first,
                int last);
};
```

EXAMPLE 8-1

The following program tests selection sort:

```
//*************************************************************
// This program illustrates how to use selection sort in a //
program.
//*************************************************************
#include <iostream>                                                    //Line 1
#include "arrayListType.h"                                             //Line 2
using namespace std;                                                   //Line 3
int main()                                                             //Line 4
{                                                                      //Line 5
    arrayListType<int> list;                                           //Line 6
    int num;                                                           //Line 7

    cout << "Line 8: Enter numbers ending with -999" << endl;          //Line 8

    cin >> num;                                                        //Line 9

    while (num != -999)                                                //Line 10
```

```
        {          //Line 11 list.insert(num);        //Line 12 cin >> num;        //Line 13
        }                                                                                    //Line 14

        cout << "Line 15: The list before sorting:" << endl;      //Line 15 list.print();          //Line 16
        cout << endl;          //Line 17

        list.selectionSort();                                                                //Line 18

        cout << "Line 19: The list after sorting:" << endl;     //Line 19 list.print();          //Line 20 cout
        << endl;          //Line 21

        return 0;                                                                            //Line 22
}                                                                                            //Line 23
```

Sample Run: In this sample run, the user input is shaded.

```
Line 8: Enter numbers ending with -999
34 67 23 12 78 56 36 79 5 32 66 -999
Line 15: The list before sorting: 34 67
23 12 78 56 36 79 5 32 66 Line 19: The
list after sorting:
5 12 23 32 34 36 56 66 67 78 79
```

For the most part, the preceding output is self-explanatory. Notice that the statement in Line 12 calls the function insert of the class arrayListType. Similarly, the statements in Lines 16 and 20 call the function print of the class arrayListType. The statement in Line 18 calls the function selectionSort to sort the list.

8.1.1 Analysis: Selection Sort

In the case of search algorithms (Chapter 7), our only concern was with the number of key (item) comparisons. A sorting algorithm makes key comparisons and also moves the data. Therefore, in analyzing the sorting algorithm, we look at the number of key comparisons as well as the number of data movements. Let us look at the performance of selection sort.

Suppose that the length of the list is n. The function swap does three item assignments and is executed n - 1 times. Hence, the number of item assignments is 3(n - 1).

The key comparisons are made by the function minLocation. For a list of length k, the function minLocation makes k - 1 key comparisons. Also, the function minLocation is executed n - 1 times (by the function selectionSort). The first time, the function minLocation finds the index of the smallest key item in the entire list and so makes n - 1 comparisons. The second time, the function minLocation finds the index of the smallest element in the sublist of length n - 1 and so makes n - 2 comparisons, and so on. Hence the number of key comparisons is as follows:

$$(n-1)+(n-2)+\cdots+2+1=(1/2)n(n-1)=(1/2)n^2-(1/2)n=O(n^2)$$

Thus, it follows that if n = 1000, the number of key comparisons the selection sort makes is $1/2(1000^2) - 1/2(1000) = 499500 \approx 500000$.

8.2 Insertion Sort: Array-Based Lists

The previous section described and analyzed the selection sort algorithm. It was shown that if n = 1000, the number of key comparisons is approximately 500,000, which is quite high. This section describes the sorting algorithm called insertion sort, which tries to improve—that is, reduce—the number of key comparisons.

Insertion sort sorts the list by moving each element to its proper place. Consider the list given in Figure 8-4.

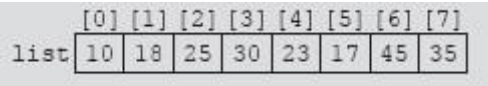


FIGURE 8-4 list

The length of the list is 8. In this list, the elements list[0], list[1], list[2], and list[3] are in order. That is, list[0]...list[3] is sorted, as shown in Figure 8-5(a).

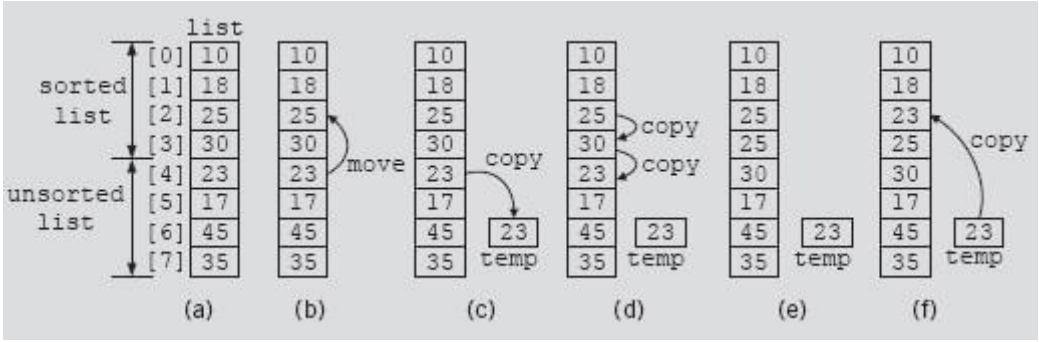


FIGURE 8-5 list elements while moving list[4] to its proper place

Next, we consider the element list[4], the first element of the unsorted list. Because list[4] < list[3], we need to move the element list[4] to its proper location. It follows that element list[4] should be moved to list[2], as shown in Figure 8-5(b). To move list[4] into list[2], first we copy list[4] into temp, a temporary memory space—see Figure 8-5(c).

Next, we copy list[3] into list[4], and then list[2] into list[3], as shown in Figure 8-5(d). After copying list[3] into list[4] and list[2] into list[3], the list is as shown in Figure 8-5(e). Next we copy temp into list[2]. Figure 8-5(f) shows the resulting list.

Now list[0]...list[4] is sorted and list[5]...list[7] is unsorted. We repeat this process on the resulting list by

moving the first element of the unsorted list into the sorted list in the proper place.

From this discussion, we see that during the sorting phase the array containing the list is divided into two sublists, upper and lower. Elements in the upper sublist are sorted; elements in the lower sublist are to be moved to the upper sublist in their proper places one at a time. We use an index—say, firstOutOfOrder—to point to the first element in the lower sublist; that is, firstOutOfOrder gives the index of the first element in the unsorted portion of the array. Initially, firstOutOfOrder is initialized to 1.

This discussion translates into the following pseudoalgorithm:

```
for (firstOutOfOrder = 1; firstOutOfOrder < length; firstOutOfOrder++) if
        (list[firstOutOfOrder] is less than list[firstOutOfOrder - 1])
        {
                copy list[firstOutOfOrder] into temp

                initialize location to firstOutOfOrder do
                {
                        a. move list[location - 1] one array slot down
                        b. decrement location by 1 to consider the next
                           element sorted of the portion of the array
                }
                while (location > 0 && the element in the upper list at location
                - 1 is greater than temp)
        }
copy temp into list[location]
```

The length of this list is 8; that is, length = 8. We initialize firstOutOfOrder to 1 (see Figure 8-6).

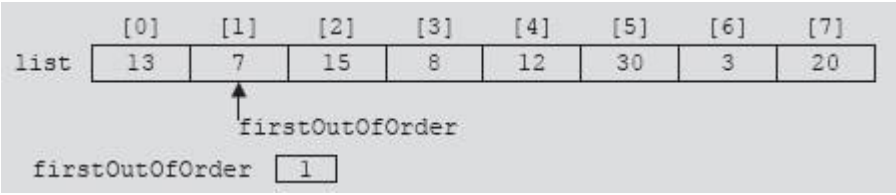


FIGURE 8-6 firstOutOfOrder = 1

Now list[firstOutOfOrder] = 7, list[firstOutOfOrder - 1] = 13 and 7 < 13, and the expression in the if statement evaluates to true, so we execute the body of the if statement.

```
        temp = list[firstOutOfOrder] = 7 location
        = firstOutOfOrder = 1
```

Next, we execute the do...while loop.

```
        list[1] = list[0] = 13          (copy list[0] into list[1])
        location = 0                    (decrement location)
```

The do...while loop terminates because location = 0. We copy temp into list[location]—that is, into list[0].

Figure 8-7 shows the resulting list.

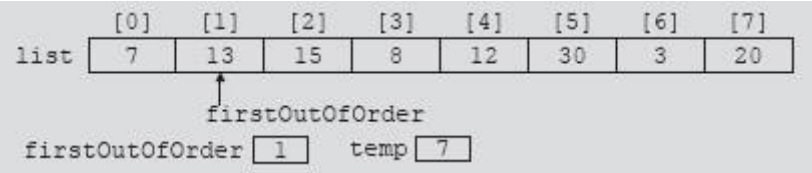


FIGURE 8-7 list after the first iteration of insertion sort

Now suppose that we have the list given in Figure 8-8(a).

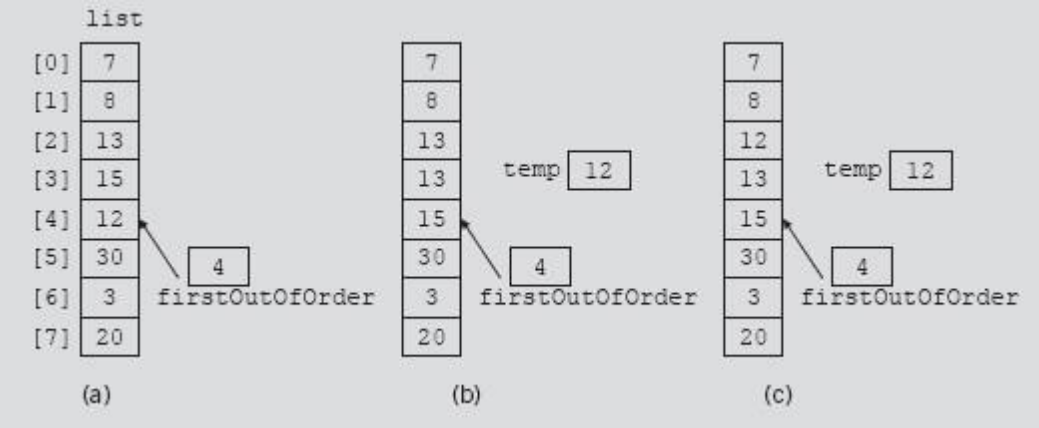


FIGURE 8-8 list elements while moving list[4] to its proper place

Here list[0]...list[3], or the elements list[0], list[1], list[2], and list[3], are in order. Now firstOutOfOrder = 4. Because list[4] < list[3], the element list[4], which is 12, needs to be moved to its proper location. As before:

temp = list[firstOutOfOrder] = 12
location = firstOutOfOrder = 4

First, we copy list[3] into list[4] and decrement location by 1. Then we copy list[2] into list[3] and again decrement location by 1. Now the value of location is 2. At this point, the list is as shown in Figure 88(b).

Next, because list[1] < temp, the do...while loop terminates. At this point, location is 2, so we copy temp into list[2]. That is, list[2] = temp = 12. Figure 8-8(c) shows the resulting list.

Suppose that we have the list given in Figure 8-9.

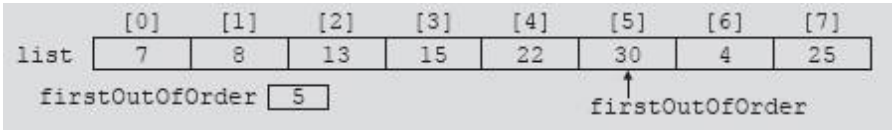


FIGURE 8-9 First out-of-order element is at position 5

Here list[0]...list[4], or the elements list[0], list[1], list[2], list[3], and list[4], are in order. Now firstOutOfOrder = 5. Because list[5] > list[4], the if statement evaluates to false. So the body of the if statement does not execute and the next iteration of the for loop, if any, takes place. Note that this is the case when the firstOutOfOrder element is already at the proper place. So we simply need to advance firstOutOfOrder to the next array element, if any.

We can repeat this process for the remaining elements of list to sort list.

The following C++ function implements the previous algorithm:

```
template <class elemType>
void arrayListType<elemType>::insertionSort()
{
        int firstOutOfOrder, location; elemType
        temp;
```

```
    for (firstOutOfOrder = 1; firstOutOfOrder < length; firstOutOfOrder++) if
            (list[firstOutOfOrder] < list[firstOutOfOrder - 1])
            {
                    temp = list[firstOutOfOrder];
                    location = firstOutOfOrder;

                    do
                    { list[location] = list[location - 1];
                            location--;
                    }
                    while (location > 0 && list[location - 1] > temp);

                    list[location] = temp;
            }
} //end insertionSort
```

8.2 Insertion Sort: Linked List-Based Lists

Insertion sort can also be applied to linked lists. Therefore, this section describes insertion sort for linked lists. Consider the linked list shown in Figure 8-10.



FIGURE 8-10 Linked list

In Figure 8-10, first is a pointer to the first node of the linked list.

If the list is stored in an array, we can traverse the list in either direction using an index variable. However, if the list is stored in a linked list, we can traverse the list in only one direction starting at the first node because the links are only in one direction, as shown in Figure 8-10. Therefore, in the case of a linked list, to find the location of the node to be inserted, we do the following. Suppose that firstOutOfOrder is a pointer to the node that is to be moved to its proper location, and lastInOrder is a pointer to the last node of the sorted portion of the list. For example, see the linked list in Figure 8-11.

(We assume that the nodes are in the usual info-link form as described in Chapter 3.)

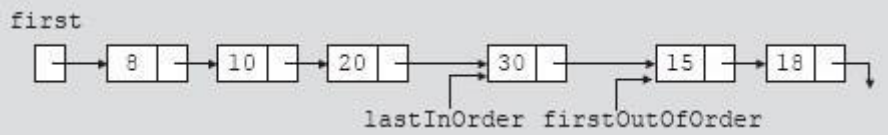


FIGURE 8-11 Linked list and pointers lastInOrder and firstOutOfOrder

First, we compare the info of firstOutOfOrder with the info of the first node. If the info of firstOutOfOrder is smaller than the info of first, then the node firstOutOfOrder is to be moved before the first node of the list; otherwise, we search the list starting at the second node to find the location where to move firstOutOfOrder. As usual, we search the list using two pointers, for example current and trailCurrent. The pointer trailCurrent points to the node just before current. In this case, the node firstOutOfOrder is to be

moved between trailCurrent and current. Of course, we also handle any special cases such as an empty list, a list with only one node, or a list in which the node firstOutOfOrder is already in the proper place. This discussion translates into the following algorithm:

```
if (firstOutOfOrder->info is less than first->info)
move firstOutOfOrder before first else
{ set trailCurrent to first
        set current to the second node in the list first->link;

        //search the list
        while (current->info is less than firstOutOfOrder->info)
        {
                advance trailCurrent;
                advance current;
        }

        if (current is not equal to firstOutOfOrder)
        { //insert firstOutOfOrder between current and trailCurrent
                lastInOrder->link = firstOutOfOrder->link; firstOutOfOrder->link
                = current;
                trailCurrent->link = firstOutOfOrder;
        }
        else //firstOutOfOrder is already at the first place lastInOrder
                = lastInOrder->link;
}
```

Let us illustrate this algorithm on the list shown in Figure 8-12. We consider several cases.

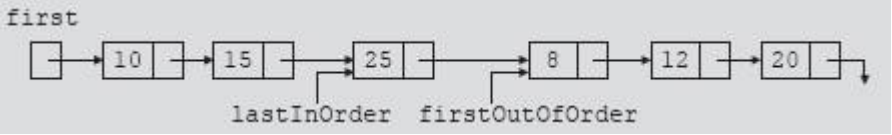


FIGURE 8-12 Linked list and pointers lastInOrder and firstOutOfOrder

Case 1: Because firstOutOfOrder->info is less than first->info, the node firstOutOfOrder is to be moved before first. So we adjust the necessary links, and Figure 8-13 shows the resulting list.

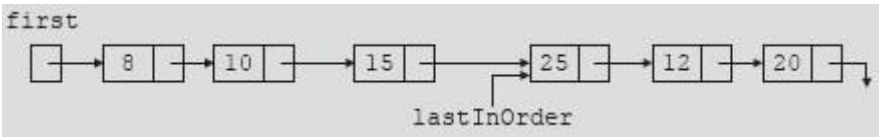


FIGURE 8-13 Linked list after moving the node with info 8 to the beginning

Case 2: Consider the list shown in Figure 8-14.

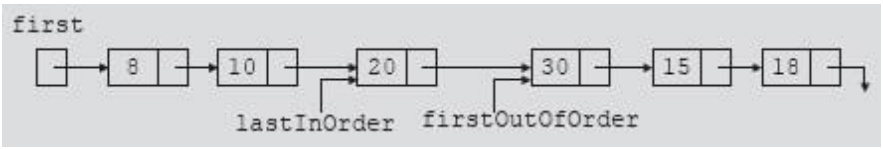


FIGURE 8-14 Linked list and pointers lastInOrder and firstOutOfOrder

Because firstOutOfOrder->info is greater than first->info, we search the list to find the place where firstOutOfOrder is to be moved. As explained previously, we use the pointers trailCurrent and current to traverse the list. For this list, these pointers end up at the nodes as shown in Figure 8-15.

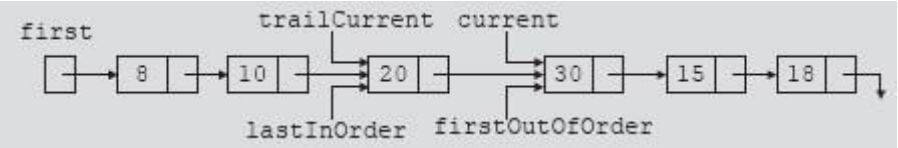


FIGURE 8-15 Linked list and pointers trailCurrent and current

Because current is the same as firstOutOfOrder, the node firstOutOfOrder is in the right place. So no adjustment of the links is necessary.

Case 3: Consider the list in Figure 8-16.

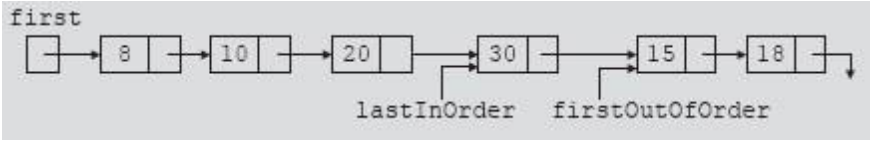


FIGURE 8-16 Linked list and pointers lastInOrder and firstOutOfOrder

Because firstOutOfOrder->info is greater than first->info, we search the list to find the place where firstOutOfOrder is to be moved. As in Case 2, we use the pointers trailCurrent and current to traverse the list. For this list, these pointers end up at the nodes as shown in Figure 8-17.

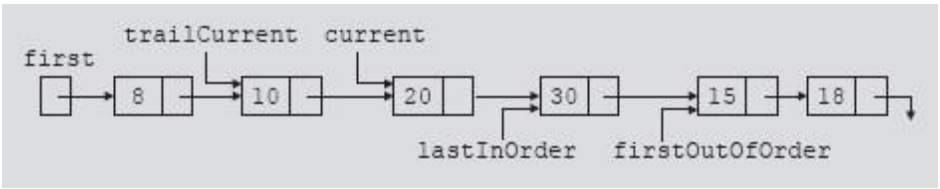


FIGURE 8-17 Linked list and pointers trailCurrent and current

Now, firstOutOfOrder is to be moved between trailCurrent and current. So we adjust the necessary links and obtain the list as shown in Figure 8-18.

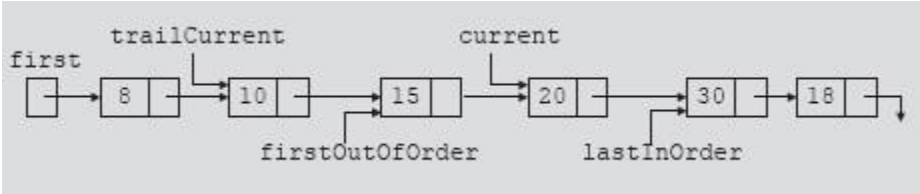


FIGURE 8-18 Linked list after moving firstOutOfOrder between trailCurrent and current

We now write the C++ function, linkedInsertionSort, to implement the previous algorithm:

```
template <class elemType>
void unorderedLinkedList<elemType>::linkedInsertionSort()
{
```

```
nodeType<elemType> *lastInOrder; nodeType<elemType>
*firstOutOfOrder; nodeType<elemType> *current;
nodeType<elemType> *trailCurrent; lastInOrder = first; if
(first == NULL) cerr << "Cannot sort an empty list." <<
endl; else if (first->link == NULL) cout << "The list is of
length 1. "<< "It is already in order." << endl; else
        while (lastInOrder->link != NULL)
        {
                firstOutOfOrder = lastInOrder->link;
                if (firstOutOfOrder->info < first->info)
                {
                        lastInOrder->link = firstOutOfOrder->link;
                        firstOutOfOrder->link = first; first =
                        firstOutOfOrder;
                }
                else

                { trailCurrent = first;
                        current = first->link;

                        while (current->info < firstOutOfOrder->info)
                        { trailCurrent = current;
                                current = current->link;
                        }

                        if (current != firstOutOfOrder)
                        {
                                lastInOrder->link = firstOutOfOrder->link; firstOutOfOrder->link
                                = current;
                                trailCurrent->link = firstOutOfOrder;
                        }
                        else lastInOrder = lastInOrder->link;
                }
```

```
            } //end while
        } //end linkedInsertionSort
```

We leave it as exercise for you to write a program to test insertion sort. See Programming Exercises 2 and 3 at the end of this chapter.

8.2.1 Analysis: Insertion Sort

Suppose that the list is of length n. If the list is sorted, the number of comparisons is (n – 1) and the number of item assignments is 0. This is the best case. (See Exercise 15 at the end of this chapter.) Now suppose that the list is sorted, but in the reverse order. In this case, it can be checked that the number of comparisons is $(1/2)(n^2 – n)$ and the number of item assignments is $(1/2)(n^2 +3n) – 2$. This is the worst case. (See Exercise 14 at the end of this chapter.)

Table 8-1 summarizes the average-case behavior of selection and insertion sort.

TABLE 8-1: Average-case behavior of the selection sort and insertion sort for a list of length n

Algorithm	Number of comparisons	Number of swaps/item assignments
Selection sort	$(1/2)n(n-1) = O(n^2)$	$3(n-1) = O(n)$
Insertion sort	$(1/4)n^2 + O(n) = O(n^2)$	$(1/4)n^2 + O(n) = O(n^2)$

8.3 Shellsort

In the previous sections, we described selection sort and insertion sort. We noticed that selection sort makes more comparisons and less item movements than insertion sort. Selection sort makes more comparisons because it makes many redundant comparisons. The number of item movements in selection sort is less because each item is moved at most one time. In fact, the number of item movements in insertion sort is considerably more than selection sort because it moves items one position at a time, so to move an item to its final position, it might require many movements.

We can reduce the number of item movements in insertion sort by modifying it. The modified insertion sort that we present next was introduced in 1959 by D. E. Shell and is known as the Shellsort algorithm. This sort is also known as diminishing-increment sort.

In Shellsort, the elements of the list are viewed as sublists at a particular distance. Each sublist is sorted, so that elements that are far apart move closer to their final position. For example, suppose that you have a list of 15 elements, as shown in Figure 8-19. First we view the list as 7 sublists, that is, we sort the elements at a distance of 7. Note that several elements have moved closer to their final position. For example, 2, 19, and 60 are closer to their final position. In the next iteration, we sort the elements at a distance of 4, as shown in Figure 8-19(b). Finally, we sort the elements at a distance of 1, that is, the entire list is sorted. Figure 8-19(c) shows the elements before and after the final sorting phase.

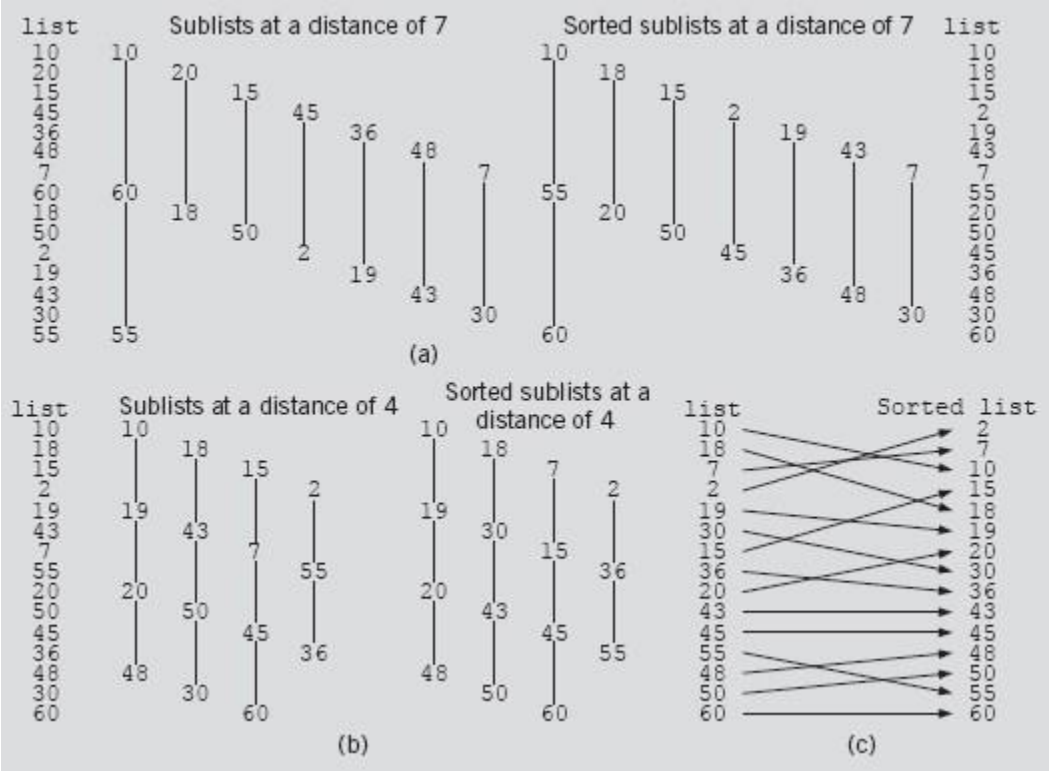

list
Sublists at a distance of 7
Sorted sublists at a distance of 7
list
(a)
list
Sublists at a distance of 4
Sorted sublists at a distance of 4
list
Sorted list
(b)
(c)

FIGURE 8-19 Lists during Shellsort

In Figure 8-19, we sorted the elements at a distance of 7, 4, and then 1. The sequence 1, 4, 7 is called the increment sequence. How do we choose an increment sequence? In general, this question cannot be answered satisfactorily. The literature provides a discussion of various increment sequences, and some have been found to be useful. Typically, the increment sequences are chosen to decrease roughly geometrically so that the number of increments is logarithmic in the size of the list. For example, if the number of increments is about one-half of the previous increment, then we need at most 20 increments to sort a list of 1 million elements. However, using as few increments as possible is desirable.

D. E. Knuth recommended the increment sequence 1, 4, 13, 40, 121, 364, 1093, 3280, ...The ratio between successive increments is about one-third. In fact, the ith increment = 3 • (i – 1)th increment + 1. There are many other increment sequences that could lead to more efficient sorts. However, for large lists, it is difficult to get a better performance by more than 20% than the increment sequence recommended by Knuth.

There are certain increment sequences that must be avoided. For example, the increment sequence, 1, 2, 4, 8, 16, 32, 64, 128, 256, . . . , is likely to lead to a bad performance because elements at odd positions are not compared with the elements at even positions until the final pass. We will use the increment sequence suggested by Knuth in the Shellsort algorithm we implement.

This following function implements the Shellsort algorithm:

```
template <class elemType>
void arrayListType<elemType>::shellSort()
{
        int inc;
        for (inc = 1; inc < (length - 1) / 9; inc = 3 * inc + 1);
        do
        { for (int begin = 0; begin < inc; begin++)
                intervalInsertionSort(begin, inc);
                inc = inc / 3;
        }
        while (inc > 0);
} //end shellSort
```

In the function shellSort, we use the function intervalInsertionSort, which is a modified version of insertion sort for array-based lists, discussed earlier in this chapter. In the intervalInsertionSort, the sublist starts at the variable begin, and the increment between successive elements is given by the variable inc instead of 1. We leave the details of the function intervalInsertionSort as an exercise for you.

The analysis of the Shellsort is difficult to obtain. In fact, to date, good estimates of the number of comparisons and item movements have been obtained only under special conditions depending on the increment sequence. Empirical studies suggest that for large lists of size n, the number of moves is in the range of $n^{1.25}$ to $1.6n^{1.25}$, which is a considerable improvement over insertion sort.

8.3 Lower Bound on Comparison-Based Sort Algorithms

The previous sections discussed selection sort and insertion sort, and noted that the average case behavior of these algorithms is $O(n^2)$. Both of these algorithms are comparison-based algorithms; that is, the lists are sorted by comparing their respective keys. Before discussing any additional sorting algorithms, let us discuss the best-case scenario for the comparisonbased sorting algorithms.

We can trace the execution of a comparison-based algorithm using a graph called a comparison tree. Let L be a list of n distinct elements, where n> 0. For any j and k, where 1 ≤ j, k ≤ n, either L* j+ < L*k+ or L* j+ > L[k]. Because each comparison of the keys has two outcomes, the comparison tree is a binary tree.While drawing this figure, we draw each comparison as a circle, called a node. The node is labeled as j:k, representing the comparison of L[j] with L[k]. If L[j] < L[k], follow the left branch; otherwise, follow the right branch. Figure 8-20 shows the comparison tree for a list of length 3. (In Figure 8-20, the rectangle, called leaf, represents the final ordering of the nodes.)

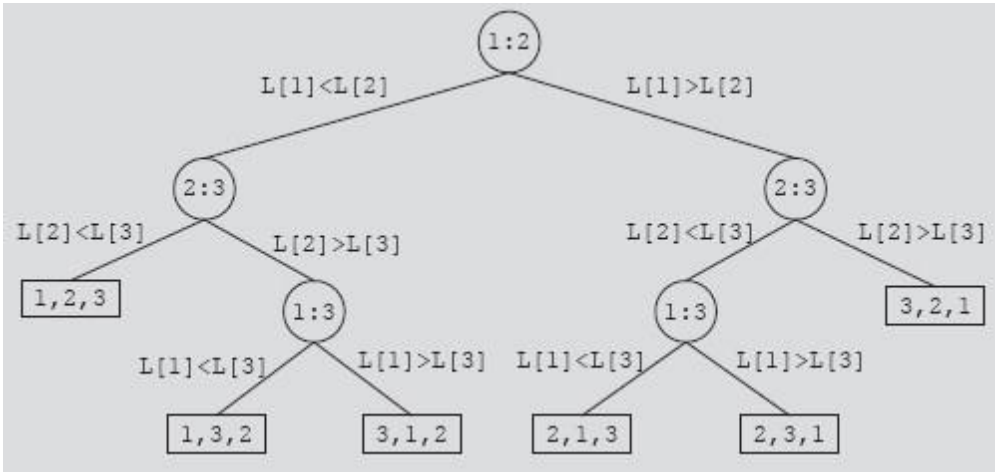


FIGURE 8-20 Comparison tree for sorting three items

We call the top node in the figure the root node. The straight line that connects the two nodes is called a branch. A sequence of branches from a node, x, to another node, y, is called a path from x to y.

Associated with each path from the root to a leaf is a unique permutation of the elements of L. This uniqueness follows because the sort algorithm only moves the data and makes comparisons. Furthermore, the data movement on any path from the root to a leaf is the same regardless of the initial inputs. For a list of n elements, n> 0, there are n! different permutations. Any one of these n! permutations might be the correct ordering of L. Thus, the comparison tree must have at least n! leaves.

Let us consider the worst case for all comparison-based sorting algorithms. We state the following result without proof.

Theorem: Let L be a list of n distinct elements. Any sorting algorithm that sorts L by comparison of the keys only, in its worst case, makes at least $O(n\log_2 n)$ key comparisons.

As analyzed in the previous sections, both selection sort and insertion sort are of the order $O(n^2)$. The remainder of this chapter discusses sorting algorithms that, on average, are of the order $O(n\log_2 n)$.

8.4 Quicksort: Array-Based Lists

In the previous section, we noted that the lower bound on comparison-based algorithms is $O(n\log_2 n)$. The sorting algorithms selection sort and insertion sort, discussed earlier in this chapter, are $O(n^2)$. In this and the next two sections, we discuss sorting algorithms that are usually of the order $O(n\log_2 n)$. The first algorithm is quicksort.

Quicksort uses the divide-and-conquer technique to sort a list. The list is partitioned into two sublists, and the two sublists are then sorted and combined into one list in such a way so that the combined list is sorted. Thus, the general algorithm is as follows:

```
if (the list size is greater than 1)
{
    a. Partition the list into two sublists, say lowerSublist and upperSublist.
    b. Quicksort lowerSublist.
    c. Quicksort upperSublist.
    d. Combine the sorted lowerSublist and sorted upperSublist.
}
```

After partitioning the list into two sublists—lowerSublist and upperSublist—these two sublists are sorted using quicksort. In other words, we use recursion to implement quicksort.

Quicksort described here is for array-based lists. The algorithm for linked lists can be developed in a similar manner and is left as an exercise for you.

In quicksort, the list is partitioned in such a way that combining the sorted lowerSublist and upperSublist is trivial. Therefore, in quicksort, all the sorting work is done in partitioning the list. Because all the sorting work occurs during the partition, we first describe the partition procedure in detail.

To partition the list into two sublists, first we choose an element of the list called the pivot. The pivot is used to divide the list into two sublists: the lowerSublist and the upperSublist. The elements in the lowerSublist are smaller than the pivot, and the elements in the upperSublist are greater than the pivot. For example, consider the list in Figure 8-21.

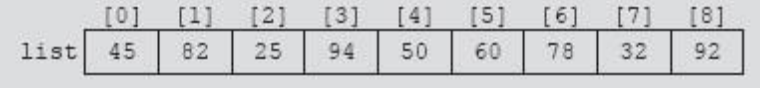


FIGURE 8-21 list before the partition

There are several ways to determine the pivot. However, the pivot is chosen so that, it is hoped, the lowerSublist and upperSublist are of nearly equal size. For illustration purposes, let us choose the middle element of the list as the pivot. The partition procedure that we describe partitions this list using the pivot as the middle element, in our case 50, as shown in Figure 8-22.

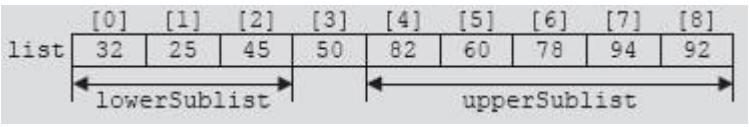


FIGURE 8-22 list after the partition

From Figure 8-22, it follows that after partitioning list into lowerSublist and upperSublist, the pivot is in the right place. Thus, after sorting lowerSublist and upperSublist, combining the two sorted sublists is trivial.

The partition algorithm is as follows: (We assume that pivot is chosen as the middle element of the list.)

1. Determine the pivot, and swap the pivot with the first element of the list.

   Suppose that the index smallIndex points to the last element smaller than the pivot. The index smallIndex is initialized to the first element of the list.

2. For the remaining elements in the list (starting at the second element) If the current element is smaller than the pivot

   a. Increment smallIndex.
   b. Swap the current element with the array element pointed to by smallIndex.

3. Swap the first element, that is, the pivot, with the array element pointed to by smallIndex.

Step 2 can be implemented using a for loop, with the loop starting at the second element of the list. Step 1 determines the pivot and moves the pivot in the first array position. During the execution of Step 2, the elements of the list get arranged as shown in Figure 8-23. (Suppose the name of the array containing the list elements is list.)

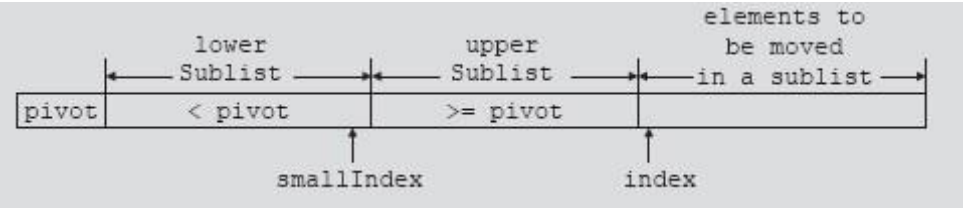


FIGURE 8-23 List during the execution of Step 2

As shown in Figure 8-23, the pivot is in the first array position, elements in lowerSublist are less than the pivot, and elements in the upperSublist are greater than or equal to the pivot. The variable smallIndex contains the index of the last element of lowerSublist and the variable index contains the index of the next element that needs to be moved either in lowerSublist or in upperSublist. As explained in Step 2, if the next element of the list (that is, list[index]) is less than the pivot, we advance smallIndex to the next array

position and swap list[index] with list[smallIndex]. Next we illustrate Step 2.

Suppose that list is as given in Figure 8-24.

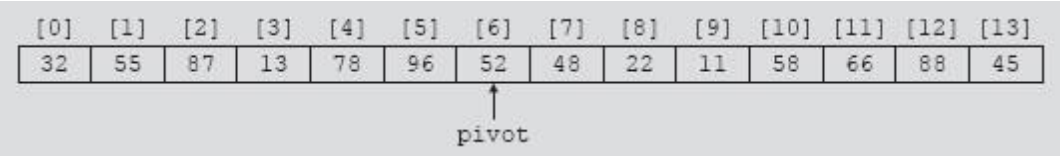


FIGURE 8-24 list before sorting

For the list in Figure 8-24, the pivot is at position 6. After moving the pivot at the first array position, the list is shown in Figure 8-25. (Notice that in Figure 8-25, 52 is swapped with 32.)

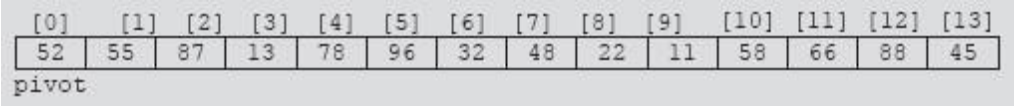


FIGURE 8-25 List after moving pivot at the first array position

Suppose that after executing Step 2 a few times, the list is as shown in Figure 8-26.

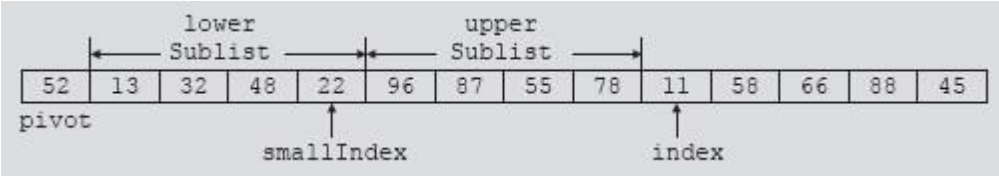


FIGURE 8-26 List after a few iterations of Step 2

As shown in Figure 8-26, the next element of the list that needs to be moved in a sublist is indicated by index. Because list[index] < pivot, we need to move the element list[index] in lowerSublist. To do so, we first advance smallIndex to the next array position and then swap list[smallIndex] with list[index]. The resulting list is shown in Figure 8-27. (Notice that 11 is swapped with 96.)

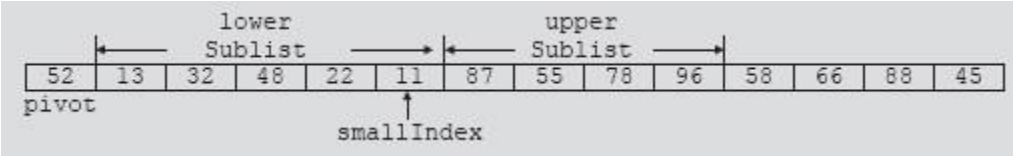


FIGURE 8-27 List after moving 11 into lowerSublist

Now consider the list in Figure 8-28.

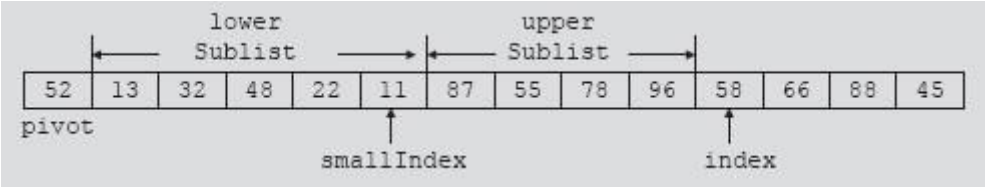


FIGURE 8-28 List before moving 58 into a sublist

For the list in Figure 8-28, list[index] is 58, which is greater than the pivot. Therefore, list[index] is to be moved in upperSublist. This is accomplished by leaving 58 at its position and increasing the size of upperSublist, by one, to the next array position. After moving 58 into upperSublist, the list is as shown in Figure 8-29.

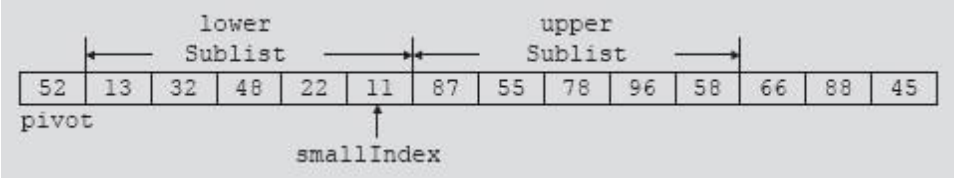


FIGURE 8-29 List after moving 58 into upperSublist

After moving the elements that are less than the pivot into lowerSublist and elements that are greater than the pivot into upperSublist (that is, after completely executing Step 2), the resulting list is as shown in Figure 8-30.



FIGURE 8-30 List elements after arranging into lowerSublist and upperSublist

Next, we execute Step 3 and move 52, the pivot, to the proper position in the list. This is accomplished by swapping 52 with 45. The resulting list is shown in Figure 8-31.

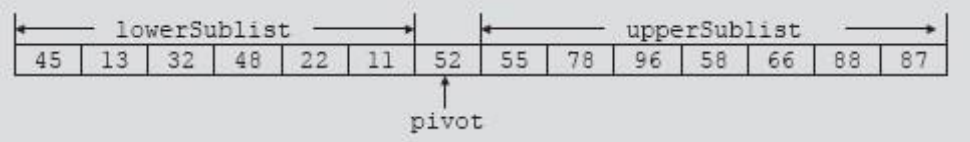


FIGURE 8-31 List after swapping 52 with 45

As shown in Figure 8-31, the preceding algorithm, Steps 1, 2, and 3, partitions the list into two sublists such that the elements less than the pivot are in lowerSublist and elements greater than or equal to the pivot are in upperSublist.

To partition the list into the lower and upper sublists, we only need to keep track of the last element of the lowerSublist and the next element of the list that needs to be moved either in lowerSublist or in upperSublist. In fact, upperSublist is between the two indices smallIndex and index.

We now write the function, partition, to implement the preceding partition algorithm. After rearranging the elements of the list, the function returns the location of the pivot so that we can determine the starting and ending locations of the sublists. Also, because the function partition will be a member of the class, it has direct access to the array containing the list. Thus, to partition a list, we need to pass only the starting and ending indices of the list.

```
template <class elemType>
int arrayListType<elemType>::partition(int first, int last)
{
        elemType pivot;
```

```
int index, smallIndex;
swap(first, (first + last) / 2);
pivot = list[first];
smallIndex = first;

for (index = first + 1; index <= last; index++)
        if (list[index] < pivot)
        {
```

```
                smallIndex++;
                swap(smallIndex, index);
            }

        swap(first, smallIndex);
        return smallIndex;
}
```

As you can see from the definition of the function partition, certain elements of the list need to be swapped. The following function, swap, accomplishes this task:

```
template <class elemType>
void arrayListType<elemType>::swap(int first, int second)
{
        elemType temp;

        temp = list[first]; list[first]
        = list[second];
        list[second] = temp;
}
```

Once the list is partitioned into lowerSublist and upperSublist, we again apply the quicksort method to sort the two sublists. Because both sublists are sorted using the same quicksort algorithm, the easiest way to implement this algorithm is to use recursion. Therefore, this section gives the recursive version of quicksort. As explained previously, after rearranging the elements of the list, the function partition returns the index of the pivot so that the starting and ending indices of the sublists can be determined.

Given the starting and ending indices of a list, the following function, recQuickSort, implements the recursive version of quicksort:

```
template <class elemType>
void arrayListType<elemType>::recQuickSort(int first, int last)
{
        int pivotLocation;
```

```
        if (first < last)
        { pivotLocation = partition(first, last);
                recQuickSort(first, pivotLocation - 1);
                recQuickSort(pivotLocation + 1,
                last);
        }
}
```

Finally, we write the quicksort function, quickSort, that calls the function recQuickSort of the original list:

```
template <class elemType>
void arrayListType<elemType>::quickSort()
{ recQuickSort(0, length -1);
}
```

We leave it as an exercise for you to write a program to test quicksort.

8.4.1 Analysis: Quicksort

Table 8-2 summarizes the behavior of quicksort for a list of length n. (The proofs of these results are given in Appendix F.)

TABLE 8-2: Analysis of quicksort for a list of length n

	Number of comparisons	Number of swaps
Average case	$1.39n\log_2 n + O(n) = O(n\log_2 n)$	$0.69n\log_2 n + O(n) = O(n\log_2 n)$
Worst case	$(1/2)(n^2 - n) = O(n^2)$	$(1/2)n^2 + (3/2)n - 2 = O(n^2)$

8.4 Mergesort: Linked List-Based Lists

In the previous section, we described quicksort and stated that the average-case behaviour of quicksort is $O(n\log_2 n)$. However, the worst-case behavior of quicksort is $O(n^2)$. This section describes a sorting algorithm whose behavior is always $O(n\log_2 n)$.

Like quicksort, mergesort uses the divide-and-conquer technique to sort a list. Mergesort also partitions the list into two sublists, sorts the sublists, and then combines the sorted sublists into one sorted list. This section describes mergesort for linked list-based lists.

We leave it for you to develop mergesort for array-based lists, which can be done by using the techniques described for linked lists.

Mergesort and quicksort differ in how they partition the list. As discussed earlier, quicksort first selects an element in the list, called pivot, and then partitions the list so that the elements in one sublist are less than pivot and the elements in the other sublist are greater than or equal to pivot. By contrast, mergesort divides the list into two sublists of nearly equal size. For example, consider the list whose elements are as follows:

list: 35 28 18 45 62 48 30 38

Mergesort partitions this list into two sublists as follows:

first sublist: 35 28 18 45

second sublist: 62 48 30 38

The two sublists are sorted using the same algorithm (that is, mergesort) used on the original list. Suppose that we have sorted the two sublists. That is, suppose that the lists are now as follows:

first sublist: 18 28 35 45 second

sublist: 30 38 48 62

Next, mergesort combines, that is, merges, the two sorted sublists into one sorted list.

Figure 8-32 further illustrates the mergesort process.

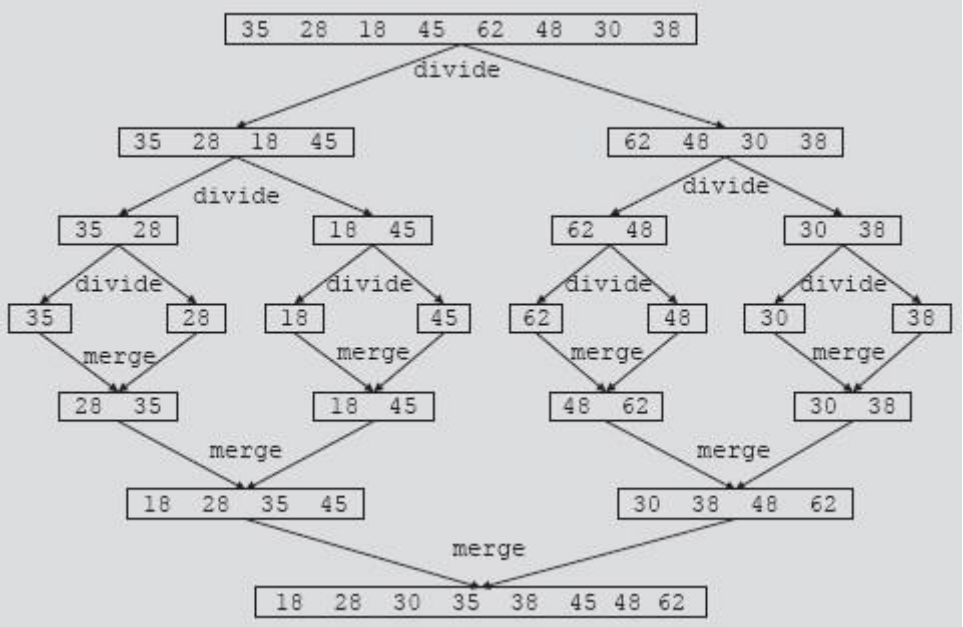


FIGURE 8-32 Mergesort algorithm

From Figure 8-32, it is clear that in mergesort, most of the sorting work is done in merging the sorted sublists.

The general algorithm for mergesort is as follows:

if the list is of a size greater than 1

{

1. Divide the list into two sublists.
2. Mergesort the first sublist.
3. Mergesort the second sublist.
4. Merge the first sublist and the second sublist.

}

As remarked previously, after dividing the list into two sublists—the first sublist and the second sublist—these two sublists are sorted using mergesort. In other words, we use recursion to implement mergesort.

We next describe the necessary algorithm to:

- Divide the list into two sublists of nearly equal size.
- Mergesort both sublists.

- Merge the sorted sublists.

8.4.2 Divide

Because data is stored in a linked list, we do not know the length of the list. Furthermore, a linked list is not a random access data structure. Therefore, to divide the list into two sublists, we need to find the middle node of the list.

Consider the list in Figure 8-33.

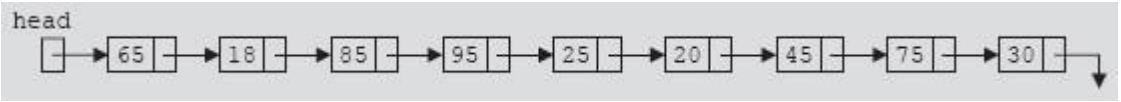


FIGURE 8-33 Unsorted linked list

To find the middle of the list, we traverse the list with two pointers—say, middle and current. The pointer middle is initialized to the first node of the list. Because this list has more than two nodes, we initialize current to the third node. (Recall that we sort the list only if it has more than one element because a list of size 1 is already sorted. Also, if the list has only two nodes, we set current to NULL.) Consider the list shown in Figure 8-34.

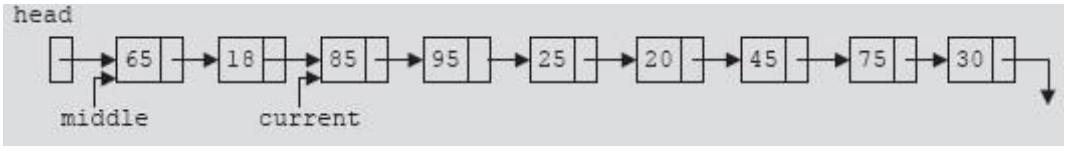


FIGURE 8-34 middle and current before traversing the list

Every time we advance middle by one node, we advance current by one node. After advancing current by one node, if current is not NULL, we again advance current by one node. That is, for the most part, every time middle advances by one node, current advances by two nodes. Eventually, current becomes NULL and middle points to the last node of the first sublist. For example, for the list in Figure 8-34, when current becomes NULL, middle points to the node with info 25 (see Figure 8-35).

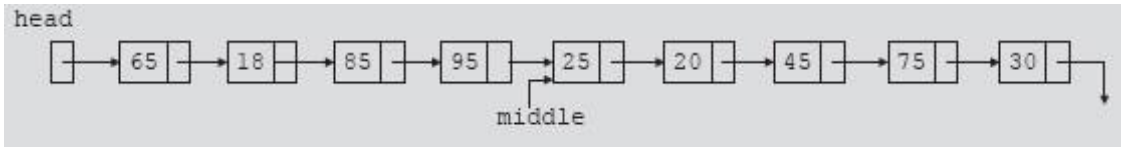


FIGURE 8-35 middle after traversing the list

It is now easy to divide the list into two sublists. First, using the link of middle, we assign a pointer to the node following middle. Then we set the link of middle to NULL. Figure 8-36 shows the resulting sublists.

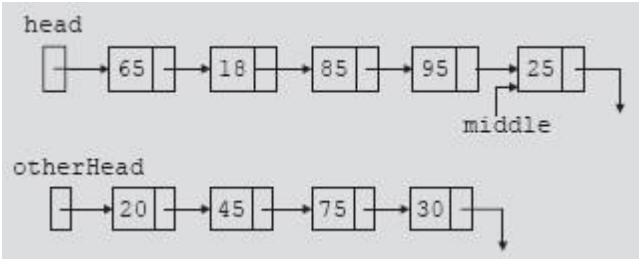


FIGURE 8-36 List after dividing it into two lists

This discussion translates into the following C++ function, divideList:

```
template <class Type>
```

```
void unorderedLinkedList<Type>::divideList(nodeType<Type>* first1,nodeType<Type>* &first2)
{
        nodeType<Type>* middle;
        nodeType<Type>* current;
        if (first1 == NULL)                first2          //list is empty
                = NULL;
```

```
        else if (first1->link == NULL)   first2          //list has only one node
                = NULL;
        else
        { middle = first1; current =
                first1->link;
                if (current != NULL)                     //list has more than two nodes
                        current = current->link; while
                (current != NULL)
                {
                        middle = middle->link; current
                        = current->link; if (current !=
                        NULL) current = current->link;
                }                                        //end while
                first2 = middle->link;                   //first2 points to the first
                                                         //node of the second sublist
                middle->link = NULL;                     //set the link of the last node
                                                         //of the first sublist to NULL
        }                                                //end else
}                                                        //end divideList
```

Now that we know how to divide a list into two sublists of nearly equal size, next we focus on merging the sorted sublists. Recall that, in mergesort, most of the sorting work is done in merging the sorted sublists.

8.4.3 Merge

Once the sublists are sorted, the next step in mergesort is to merge the sorted sublists. Sorted sublists are merged into a sorted list by comparing the elements of the sublists, and then adjusting the references of the nodes with the smaller info. Let us illustrate this procedure on the sublists shown in Figure 8-37. Suppose that first1 points to the first node of the first sublist, and first2 points to the first node of the second sublist.

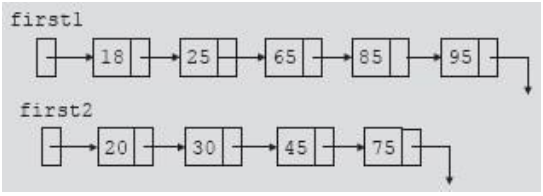


FIGURE 8-37 Sublists before merging

We first compare the info of the first node of each of the two sublists to determine the first node of the merged list. We set newHead to point to the first node of the merged list. We also use the pointer lastMerged to keep track of the last node of the merged list. The pointer of the first node of the sublist with the smaller node then advances to the next node of that sublist. Figure 8-38 shows the sublist of Figure 8-37 after setting newHead and lastMerged and advancing first1.

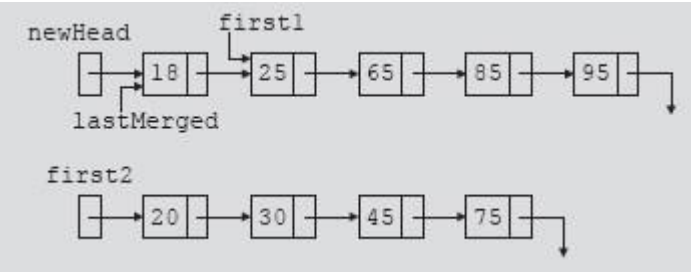


FIGURE 8-38 Sublists after setting newHead and lastMerged and advancing first1

In Figure 8-38, first1 points to the first node of the first sublist that is yet to be merged with the second sublist. So we again compare the nodes pointed to by first1 and first2, and adjust the link of the smaller node and the last node of the merged list so as to move the smaller node to the end of the merged list. For the sublists shown in Figure 8-38, after adjusting the necessary links, we have Figure 8-39.

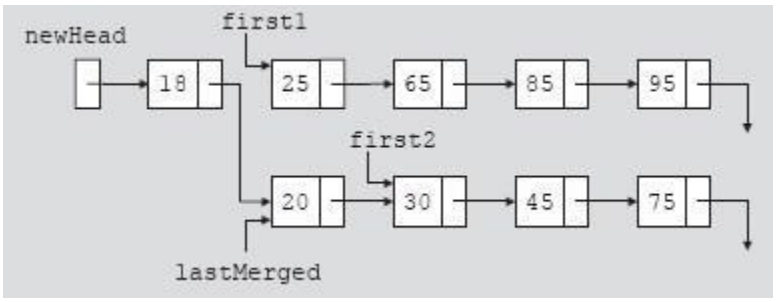


FIGURE 8-39 Merged list after putting the node with info 20 at the end of the merged list

We continue this process for the remaining elements of both sublists. Every time we move a node to the merged list, we advance either first1 or first2 to the next node. Eventually, either first1 or first2 becomes NULL. If first1 becomes NULL, the first sublist is exhausted first, so we attach the remaining nodes of the second sublist at the end of the partially merged list. If first2 becomes NULL, the second sublist is exhausted first, so we attach the remaining nodes of the first sublist at the end of the partially merged list.

Following this discussion, we can now write the C++ function, mergeList, to merge the two sorted sublists. The references (that is, addresses) of the first nodes of the sublists are passed as parameters to the function mergeList.

```
template <class Type>
nodeType<Type>* unorderedLinkedList<Type>::mergeList
        (nodeType<Type>* first1,nodeType<Type>* first2)
{
```

```
nodeType<Type> *lastSmall;          //pointer to the last node of the merged list
nodeType<Type> *newHead;            //pointer to the merged list
if (first1 == NULL)                 //the first sublist is empty
        return first2;
else if (first2 == NULL)            //the second sublist is empty
        return first1;
else
{
```

```
    if (first1->info < first2->info)          //compare the first nodes
    {
        newHead = first1; first1
        = first1->link;
        lastSmall = newHead;
    }
    else
    {
        newHead = first2; first2
        = first2->link; lastSmall
        = newHead;
    }

    while (first1 != NULL && first2 != NULL)
    {
        if (first1->info < first2->info)
        { lastSmall->link = first1; lastSmall
            = lastSmall->link; first1 =
            first1->link;
        }
        else
        { lastSmall->link = first2; lastSmall
            = lastSmall->link; first2 =
            first2->link;
        }
    }                                         //end while

    if (first1 == NULL)                       //first sublist is exhausted first lastSmall->link
        = first2;
    else                                      //second sublist is exhausted first
        lastSmall->link = first1;
    return newHead;
}
```

```
}                                                    //end mergeList
```

Finally, we write the recursive mergesort function, recMergeSort, which uses the divideList and mergeList functions to sort a list. The reference of the first node of the list to be sorted is passed as a parameter to the function recMergeSort.

```
template <class Type>
void  unorderedLinkedList<Type>::recMergeSort(nodeType<Type>*  &head)
{
        nodeType<Type> *otherHead;

        if (head != NULL)                                    //if the list is not empty if (head-
                >link != NULL)                      //if the list has more than one node
                {
```

```
                divideList(head, otherHead);
                recMergeSort(head);
                recMergeSort(otherHead);
                head = mergeList(head, otherHead);
        }
}                                                   //end recMergeSort
```

We can now give the definition of the function mergeSort, which should be included as a public member of the class unorderedLinkedList. (Note that the functions divideList, merge, and recMergeSort can be included as private members of the class unorderedLinkedList because these functions are used only to implement the function mergeSort.) The function mergeSort calls the function recMergeSort and passes first to this function. It also sets last to point to the last node of the list. The definition of the function mergeSort is as follows:

```
template<class Type>
void unorderedLinkedList<Type>::mergeSort()
{
        recMergeSort(first);

        if (first == NULL)
        last = NULL; else
        {
                last = first;
                while (last->link != NULL) last
                        = last->link;
        }
}                                                   //end mergeSort
```

We leave it as an exercise for you to write a program to test mergesort.

8.4.4 Analysis: Mergesort

Suppose that L is a list of n elements, where n> 0. Suppose that n is a power of 2, that is, $n = 2^m$ for some

nonnegative integer m, so that we can divide the list into two sublists, each of size n / 2 = 2^m / 2 = 2^{m-1}. Moreover, each sublist can also be divided into two sublists of the same size. Each call to the function recMergeSort makes two recursive calls to the function recMergeSort and each call divides the sublist into two sublists of the same size. Suppose that m = 3, that is, n = 2^3 = 8. So the length of the original list is 8. The first call to the function recMergeSort divides the original list into two sublists, each of size 4. The first call then makes two recursive calls to the function recMergeSort. Each of these recursive calls divides each sublist, of size 4, into two sublists, each of size 2. We now have 4 sublists, each of size 2. The next set of recursive calls divides each sublist, of size 2, into sublists of size 1. So we now have 8 sublists, each of size 1. It follows that the exponent 3 in 2^3 indicates the level of the recursion, as shown in Figure 8-40.

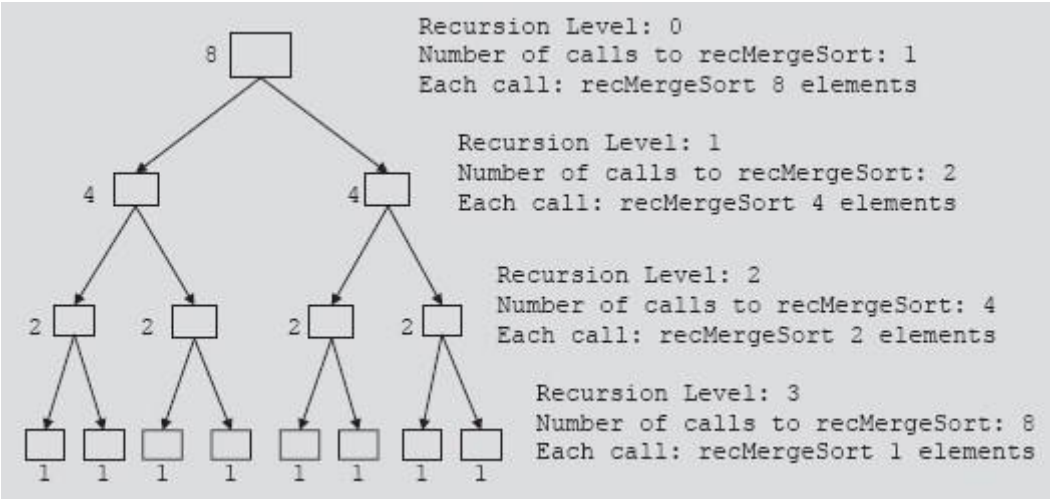


FIGURE 8-40 Levels of recursion levels to recMergeSort for a list of length 8

Let us consider the general case when $n = 2^m$. Note that the number of recursion levels is m. Also, note that to merge a sorted list of size s with a sorted list of size t, the maximum number of comparisons is s + t - 1.

Consider the function mergeList, which merges two sorted lists into a sorted list. Note that this is where the actual work, comparisons and assignments, is done. The initial call to the function recMergeSort, at level 0, produces two sublists, each of size n / 2. To merge these two lists, after they are sorted, the maximum number of comparisons is n / 2 + n / 2 – 1 = n – 1 = O(n). At level 1, we merge two sets of sorted lists, where each sublist is of size n / 4. To merge two sorted sublists, each of size n / 4, we need at most n / 4 + n / 4 – 1 = n / 2 – 1 comparisons. Thus, at level 1 of the recursion, the number of comparisons is 2(n / 2 – 1) = n – 2 = O(n). In general, at level k of the recursion, there are a total of 2^k calls to the function mergeList. Each of these calls merge two sublists, each of size $n / 2^{k+1}$, which requires a maximum of $n / 2^k - 1$ comparisons. Thus, at level k of the recursion, the maximum number of comparisons is $2^k (n / 2^k - 1) = n – 2^k = O(n)$. It now follows that the maximum number of comparisons at each level of the recursion is O(n). Because the number of levels of the recursion is m, the maximum number of comparisons made by mergesort is O(nm). Now $n = 2^m$ implies that $m = \log_2 n$. Hence, the maximum number of comparisons made by mergesort is $O(n \log_2 n)$.

If W(n) denotes the number of key comparisons in the worst case to sort L, then $W(n) = O(n \log_2 n)$.

Let A(n) denote the number of key comparisons in the average case. In the average case, during the merge process one of the sublists will exhaust before the other list. From this, it follows that on average merging of two sorted sublists of combined size n, the number of comparisons will be less than n - 1. On average,

it can be shown that the number of comparisons for mergesort is given by the following equation: If n is a power of 2, $A(n) = n \log_2 n - 1.25n = O(n \log_2 n)$. This is also a good approximation when n is not a power of 2.

8.5 Heapsort: Array-Based Lists

In an earlier section, we described the quicksort algorithm for contiguous lists, that is, array-based lists. We remarked that, on average, quicksort is of the order $O(n\log_2 n)$. However, in the worst case, quicksort is of the order $O(n^2)$. This section describes another algorithm, the heapsort, for array-based lists. This algorithm is of order $O(n \log_2 n)$ even in the worst case, therefore overcoming the worst case of the quicksort.

Definition: A heap is a list in which each element contains a key, such that the key in the element at position k in the list is at least as large as the key in the element at position 2k + 1 (if it exists) and 2k + 2 (if it exists).

Recall that, in C++ the array index starts at 0. Therefore, the element at position k is in fact the k + 1th element of the list.

Consider the list in Figure 8-41.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
85	70	80	50	40	75	30	20	10	35	15	62	58

FIGURE 8-41 A heap

It can be checked that the list in Figure 8-41 is in a heap. For example, consider list[3], which is 50. The elements at position list[7] and list[8] are 20 and 10, respectively. Clearly, list[3] is larger than list[7] and list[8].

In heapsort, elements at position k, 2k + 1, and 2k + 2, if they exist, are accessed frequently. Therefore, to facilitate the discussion of heapsort, we typically view data in the form of a complete binary tree as described next. For example, the data given in Figure 8-41 can be viewed in a complete binary tree, as shown in Figure 8-42.

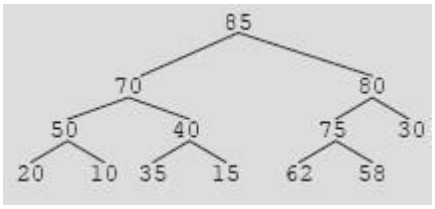


FIGURE 8-42 Complete binary tree corresponding to the list in Figure 8-41

In Figure 8-42, the first element of the list, which is 85, is the root node of the tree. The second element of the list, which is 70, is the left child of the root node; the third element of the list, which is 80, is the right child of the root node. Thus, in general, for the node k, which is the k - 1th element of the list, its left child is the 2kth (if it exists) element of the list, which is at position 2k - 1 in the list, and the right child is the 2k + 1st (if it exists) element of the list, which is at position 2k in the list. Note that Figure 842 clearly shows that the list in Figure 8-41 is in a heap. Also note that in Figure 8-42, the elements 20, 10, 35, 15, 62, 58, and 30 are called leaves as they have no children.

As remarked earlier, to demonstrate the heapsort algorithm, we will draw the complete binary tree corresponding to a list. Note that even though we will draw a complete binary tree to illustrate heapsort, the data gets manipulated in an array. We now describe heapsort.

The first step in heapsort is to convert the list into a heap, called buildHeap. After we convert the array into a heap, the sorting phase begins.

8.4.5 Build Heap

This section describes the build heap algorithm.

The general algorithm is as follows: Suppose length denotes the length of the list. Let index = length / 2 – 1. Then list[index] is the last element in the list which is not a leaf; that is, this element has at least one child. Thus, elements list[index + 1]...list[length – 1] are leaves.

First, we convert the subtree with the root node list[index] into a heap. Note that this subtree has at most three nodes. We then convert the subtree with the root node list[index - 1] into a heap, and so on.

To convert a subtree into a heap, we perform the following steps: Suppose that list[a] is the root node of the subtree, list[b] is the left child, and list[c], if it exists, is the right child of list[a].

Compare list[b] with list[c] to determine the larger child. If list[c] does not exist, then list[b] is the larger child. Suppose that largerIndex indicates the larger child. (Notice that, largerIndex is either b or c.)

Compare list[a] with list[largerIndex]. If list[a] < list[largerIndex], then swap list[a] with list[largerIndex]; otherwise, the subtree with root node list[a] is already in a heap.

Suppose that list[a] < list[largerIndex] and we swap the elements list[a] with list[largerIndex]. After making this swap, the subtree with root node list[largerIndex] might not be in a heap. If this is the case, then we repeat Steps 1 and 2 at the subtree with root node list[largerIndex] and continue this process until either the heap in the subtrees is restored or we arrive at an empty subtree. This step is implemented using a loop, which will be described when we write the algorithm.

Consider the list in Figure 8-43. Let us call this list.

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
list	15	60	72	70	56	32	62	92	45	30	65

FIGURE 8-43 Array list

Figure 8-44 shows the complete binary tree corresponding to the list in Figure 8-43.

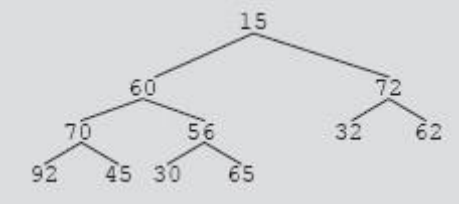


FIGURE 8-44 Complete binary tree corresponding to the list in Figure 8-43

To facilitate this discussion, when we say node 56, we mean the node with info 56.

This list has 11 elements, so the length of the list is 11. To convert the array into a heap, we start at the list element n/2 - 1 = 11/2 – 1 = 5 – 1 = 4, which is the fifth element of the list.

Now list[4] = 56. The children of list[4] are list[4 * 2 + 1] and list[4 * 2 + 2], that is, list[9] and list[10]. In the previous list, both list[9] and list[10] exist.

To convert the tree with root node list[4], we perform the previous hree steps:

1. Find the larger of list[9] and list[10], that is, the largest child of list[4]. In this case, list[10] is larger than list[9].
2. Compare the larger child with the parent node. If the larger child is larger than the parent, swap the larger child with the parent. Because list[4] < list[10], we swap list[4] with list[10].
3. Because list[10] does not have a subtree, Step 3 does not execute.

Figure 8-45(a) shows the resulting binary tree.

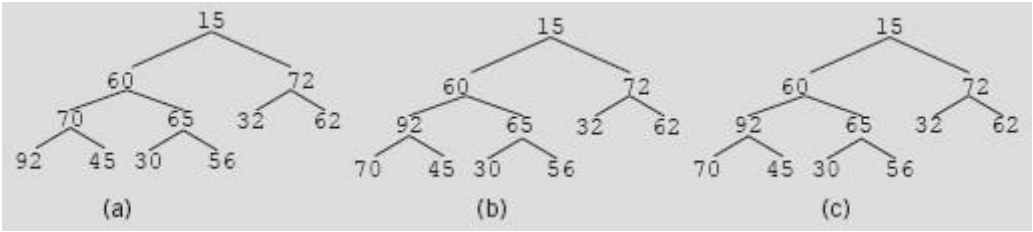


FIGURE 8-45 Binary tree while building heaps at list[4], list[3], and list[2]

Next, we consider the subtree with root node list[3], that is, 70 and repeat the three steps given earlier, to obtain the complete binary tree as given in Figure 8-45(b). (Notice that Step 3 does not execute here either.)

Now we consider the subtree with the root node list[2], that is, 72, and apply the three steps given earlier. Figure 8-45(c) shows the resulting binary tree. (Note that in this case, because the parent is larger than both children, this subtree is already in a heap.)

Next, we consider the subtree with the root node list[1], that is, 60, see 8-45(c). First, we apply Steps 1 and 2. Because list[1] = 60 < list[3] = 92 (the larger child), we swap list[1] with list[3], to obtain the tree as given in Figure 8-46(a).

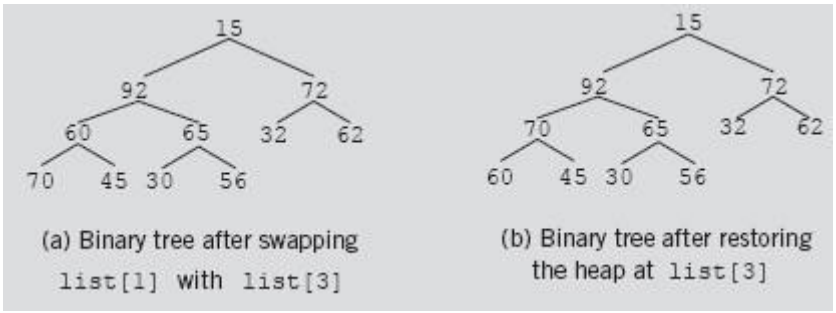


FIGURE 8-46 Binary tree while building heap at list[1]

However, after swapping list[1] with list[3], the subtree with the root node list[3], that is, 60, is no longer a heap. Thus, we must restore the heap in this subtree. To do this, we apply Step 3 and find the larger child of 60 and swap it with 60. We then obtain the binary tree as given in Figure 8-46(b).

Once again, the subtree with the root node list[1], that is, 92, is in a heap (see Figure 8-46(b)).

Finally, we consider the tree with the root node list[0], that is, 15. We repeat the previous three steps to obtain the binary tree as given in Figure 8-47(a).

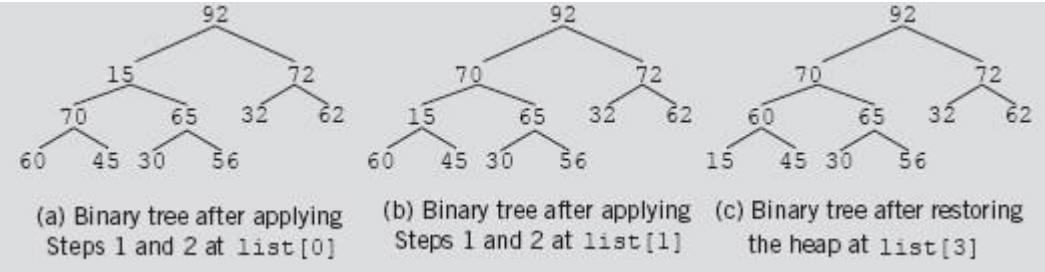


FIGURE 8-47 Binary tree while building heap at list[0]

We see that the subtree with the root node list[1], that is, 15, is no longer in a heap. So we must apply Step 3 to restore the heap in this subtree. (This requires us to repeat Steps 1 and 2 at the subtree with root node list[1].) We swap list[1] with the larger child, which is list[3] (that is, 70). We then get the binary tree of Figure 8-47(b).

The subtree with the root node list[3] = 15 is not in a heap, and so we must restore the heap in this subtree. To do so, we apply Steps 1 and 2 at the subtree with root node list[3]. We swap list[3] with the larger child, which is list[7] (that is, 60). Figure 8-47(c) shows the resulting binary tree.

The resulting binary tree in Figure 8-47(c) is in a heap, and so the list corresponding to this complete binary tree is in a heap.

Thus, in general, starting at the lowest level from right to left, we look at a subtree and convert the subtree into a heap as follows: If the root node of the subtree is smaller than the larger child, we swap the root node with the larger child. After swapping the root node with the larger child, we must restore the heap in the subtree whose root node was swapped.

Suppose low contains the index of the root node of the subtree and high contains the index of the last item in the list. The heap is to be restored in the subtree rooted at list[low]. The preceding discussion translates into the following C++ algorithm:

```
int largeIndex = 2 * low + 1;                    //index of the left child

while (largeIndex <= high)
{
        if ( largeIndex < high) if (list[largeIndex] <
```

```
            list[largeIndex + 1])
                largeIndex = largeIndex + 1;          //index of the larger child
        if (list[low] > list[largeIndex])          break; //the subtree is already in a heap
        else
        {
            swap(list[low], list[largeIndex]);      //Line swap**
            low = largeIndex;                       //go to the subtree to further
                                                    //restore the heap
            largeIndex = 2 * low + 1;
        }                                           //end else
}                                                   //end while
```

The swap statement at the line marked Line swap** swaps the parent with the larger child. Because a swap statement makes three item assignments to swap the contents of two variables, each time through the loop three item assignments are made. The while loop moves the parent node to a place in the tree so that the resulting subtree with the root node list[low] is in a heap. We can easily reduce the number of assignments each time through the loop from three to one by first storing the root node in a temporary location, say temp. Then each time through the loop, the larger child is compared with temp. If the larger child is larger than temp, we move the larger child to the root node of the subtree under consideration.

Next, we describe the function heapify, which restores the heap in a subtree by making one item assignment each time through the loop. The index of the root node of the list and the index of the last element of the list are passed as parameters to this function.

```
template<class elemType>
void arrayListType<elemType>::heapify(int low, int high)
{
        int largeIndex;

        elemType temp = list[low];                      //copy the root node of the subtree
        largeIndex = 2 * low + 1;                       //index of the left child
        while (largeIndex <= high)
        { if (largeIndex < high) if (list[largeIndex] < list[largeIndex + 1]) largeIndex =
                largeIndex + 1;  //index of the largest child

                if (temp > list[largeIndex])    break;      //subtree is already in a heap
                else
                {
                        list[low] = list[largeIndex];       //move the larger child to the root
                        low = largeIndex;                   //go to the subtree to restore the heap
                        largeIndex = 2 * low + 1;
                }
        }                                               //end while
```

```
        list[low] = temp;                                   //insert temp into the tree, that is, list
}                                                           //end heapify
```

Next, we use the function heapify to implement the buildHeap function to convert the list into a heap.

```
template <class elemType>
void arrayListType<elemType>::buildHeap()
{ for (int index = length / 2 - 1; index >= 0; index--)
        heapify(index, length - 1);
}
```

We now describe heapsort.

Suppose the list is in a heap. Consider the complete binary tree representing the list as given in Figure 8-48(a).

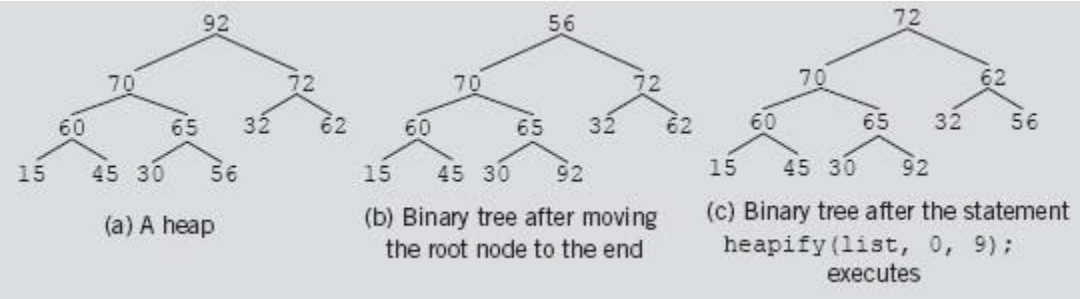


FIGURE 8-48 Heapsort

Because this is a heap, the root node is the largest element of the tree, that is, the largest element of the list. So it must be moved to the end of the list. We swap the root node of the tree, that is, the first element of the list, with the last node in the tree (which is the last element of the list). We then obtain the binary tree as shown in Figure 8-48(b).

Because the largest element is now in its proper place, we consider the remaining elements of the list, that is, elements list[0]...list[9]. The complete binary tree representing this list is no longer a heap, and so wemust restore the heap in this portion of the complete binary tree. We use the function heapify to restore the heap. A call to this function is as follows:

heapify(list, 0, 9);

We thus obtain the binary tree as shown in Figure 8-48(c).

We repeat this process for the complete binary tree corresponding to the list elements list[0]...list[9]. We swap list[0] with list[9] and then restore the heap in the complete binary tree corresponding to the list elements list[0]...list[8]. We continue this process.

The following C++ function describes this algorithm:

```
template <class elemType>
void arrayListType<elemType>::heapSort()
{
        elemType temp; buildHeap();
        for (int lastOutOfOrder = length - 1; lastOutOfOrder >= 0; lastOutOfOrder--)
        {
                temp = list[lastOutOfOrder];
```

```
                list[lastOutOfOrder] = list[0]; list[0]
                = temp;
                heapify(0, lastOutOfOrder - 1);
        }//end for
}//end heapSort
```

We leave as an exercise for you to write a program to test heapsort;

8.4.6 Analysis: Heapsort

Suppose that L is a list of n elements, where n> 0. In the worst case, the number of key comparisons in heapsort to sort L (the number of comparisons in heapSort and the number of comparisons in buildHeap) is $2n\log_2 n + O(n)$. Also, in the worst case, the number of item assignments in heapsort to sort L is $n\log_2 n + O(n)$. On average, the number of comparisons made by heapsort to sort L is of $O(n\log_2 n)$.

In the average case of quicksort, the number of key comparisons is $1.39n\log_2 n + O(n)$ and the number of swaps is $0.69n\log_2 n + O(n)$. Because each swap is three assignments, the number of item assignments in the average case of quicksort is at least $1.39n\log_2 n + O(n)$. It now follows that for the key comparisons, the average case of quicksort is somewhat better than the worst case of heapsort. On the other hand, for the item assignments, the average case of quicksort is somewhat poorer than the worst case of heapsort. However, the worst case of quicksort is of $O(n^2)$. Empirical studies have shown that heapsort usually takes twice as long as quicksort, but avoids the slight possibility of poor performance.

8.6 Priority Queues (Revisited)

Chapter 6 introduced priority queues. Recall that in a priority queue, customers or jobs with higher priorities are pushed to the front of the queue. Chapter 6 stated that we would discuss the implementation of priority queues after describing heapsort. For simplicity, we assume that the priority of the queue elements is assigned using the relational operators.

In a heap, the largest element of the list is always the first element of the list. After removing the largest element of the list, the function heapify restores the heap in the list. To ensure that the largest element of the priority queue is always the first element of the queue, we can implement priority queues as heaps. We can write algorithms similar to the ones used in the function heapify to insert an element in the priority queue (addQueue operation), and remove an element from the queue (deleteQueue operation). The next two sections describe these algorithms.

8.4.7 Insert An Element In The Priority Queue

Assuming the priority queue is implemented as a heap, we perform the following steps:

1. Insert the new element in the first available position in the list. (This ensures that the array holding the list is a complete binary tree.)

2. After inserting the new element in the heap, the list might no longer be a heap. So to restore the heap:

   while (the parent of the new entry is smaller than the new entry) swap
   the parent with the new entry.

Notice that restoring the heap might result in moving the new entry to the root node.

8.4.8 Remove An Element From The Priority Queue

Assuming the priority queue is implemented as a heap, to remove the first element of the priority queue, we perform the following steps:

1. Copy the last element of the list into the first array position.

2. Reduce the length of the list by 1.
3. Restore the heap in the list.

The other operations for priority queues can be implemented in the same way as implemented for queues. We leave the implementation of the priority queues as an exercise for you.

Key Terms

1. Selection sort sorts a list by finding the smallest (or equivalently, the largest) element in the list, and moving it to the beginning (or the end) of the list.
2. For a list of length n, where n> 0, selection sort makes (1/2)n(n – 1) key comparisons and 3(n – 1) item assignments.
3. For a list of length n, where n> 0, on average, insertion sort makes $(1/4)n^2 + O(n) = O(n^2)$ key comparisons and $(1/4)n^2 + O(n) = O(n^2)$ item assignments.
4. Empirical studies suggest that for large lists of size n, the number of moves in Shellsort is in the range of $n_{1.25}$ to $1.6n_{1.25}$.
5. Let L be a list of n distinct elements. Any sorting algorithm that sorts L by comparison of the keys only, in its worst case, makes at least $O(n\log_2 n)$ key comparisons. 6. Both quicksort and mergesort sort a list by partitioning the list.
7. To partition a list, quicksort first selects an item from the list, called the pivot. The algorithm then rearranges the elements so that the elements in one of the sublists are less than the pivot, and the elements in the second sublist are greater than or equal to the pivot.
8. In a quicksort, the sorting work is done in partitioning the list.
9. On average, the number of key comparisons in quicksort is $O(n\log_2 n)$. In the worst case, the number of key comparisons in quicksort is $O(n^2)$.
10. Mergesort partitions the list by dividing it in the middle.
11. In mergesort, the sorting work is done in merging the list.
12. The number of key comparisons in mergesort is $O(n\log_2 n)$.
13. A heap is a list in which each element contains a key, such that the key in the element at position k in the list is at least as large as the key in the element at position 2k + 1 (if it exists) and 2k + 2 (if it exists).
14. The first step in the heapsort algorithm is to convert the list into a heap, called buildHeap. After we convert the array into a heap, the sorting phase begins.
15. Suppose that L is a list of n elements, where n> 0. In the worst case, the number of key comparisons

in heapsort to sort L is $2n\log_2 n + O(n)$. Also, in the worst case, the number of item assignments in heapsort to sort L is $n\log_2 n + O(n)$.

Exercises

1. Sort the following list using selection sort as discussed in this chapter. Show the list after each iteration of the outer for loop.

   26, 45, 17, 65, 33, 55, 12, 18

2. Sort the following list using selection sort as discussed in this chapter. Show the list after each iteration of the outer for loop.

   36, 55, 17, 35, 63, 85, 12, 48, 3, 66

3. Assume the following list of keys: 5, 18, 21, 10, 55, 20

The first three keys are in order. To move 10 to its proper position using insertion sort as described in this chapter, exactly how many key comparisons are executed?

4. Assume the following list of keys: 7, 28, 31, 40, 5, 20

The first four keys are in order. To move 5 to its proper position using insertion sort as described in this chapter, exactly how many key comparisons are executed?

5. Assume the following list of keys: 28, 18, 21, 10, 25, 30, 12, 71, 32, 58, 15

This list is to be sorted using insertion sort as described in this chapter for array-based lists. Show the resulting list after six passes of the sorting phase—that is, after six iterations of the for loop.

6. Recall insertion sort for array-based lists as discussed in this chapter. Assume the following list of keys: 18, 8, 11, 9, 15, 20, 32, 61, 22, 48, 75, 83, 35, 3

Exactly how many key comparisons are executed to sort this list using insertion sort?

7. Explain why the number of item movements in Shellsort is less than the number of item movements in insertion sort.

8. Consider the following list of keys: 80, 57, 65, 30, 45, 77, 27, 4, 90, 54, 45, 2, 63, 38, 81, 28, 62.

Suppose that this list is to be sorted using Shellsort.

Show the list during each increment, as in this chapter. a.
Use the increment sequence 1, 3, 5
b. Use the increment sequence 1, 4, 7.

9. Both mergesort and quicksort sort a list by partitioning the list. Explain how mergesort differs from quicksort in partitioning the list.

10. Assume the following list of keys: 16, 38, 54, 80, 22, 65, 55, 48, 64, 95, 5, 100, 58, 25, 36

This list is to be sorted using quicksort as discussed in this chapter. Use pivot as the middle element of the list.

a. Give the resulting list after one call to the partition procedure.
b. Give the resulting list after two calls to the partition procedure.

11. Assume the following list of keys: 18, 40, 16, 82, 64, 67, 57, 50, 37, 47, 72, 14, 17, 27, 35

This list is to be sorted using quicksort as discussed in this chapter. Use pivot as the median of the first, last, and middle elements of the list.

a. What is the pivot?

b. Give the resulting list after one call to the partition procedure.

12. Using the function buildHeap as given in this chapter, convert the following array into a heap. Show the final form of the array.

47, 78, 81, 52, 50, 82, 58, 42, 65, 80, 92, 53, 63, 87, 95, 59, 34, 37, 7, 20

13. Suppose that the following list was created by the function buildHeap during the heap creation phase of heapsort.

100, 85, 94, 47, 72, 82, 76, 30, 20, 60, 65, 50, 45, 17, 35, 14, 28, 5

Show the resulting array after two passes of heapsort. (Use the heapify procedure as given in this chapter.) Exactly how many key comparisons are executed during the first pass?

14. Suppose that L is a list is of length n and it is sorted using insertion sort. If L is already sorted in the reverse order, show that the number of comparisons is $(1/2)(n^2 – n)$ and the number of item assignments is $(1/2)(n^2 +3n) – 2$.

15. Suppose that L is a list is of length n and it is sorted using insertion sort. If L is already sorted, show that the number of comparisons is $(n – 1)$ and the number of item assignments is 0.

16. Write the definition of the class arrayListType that implements the sorting algorithms for arraybased lists as discussed in this chapter.

17. Write the definition of the class unorderedLinkedList that implements the searching (described in Chapter 3) and sorting algorithms for linked lists as discussed in this chapter.

9 TOPIC 9: BINARY TREES AND B-TREES



LEARNING OUTCOMES:

After studying this topic you should be able to:

- Understand the importance of programming language paradigms
- Become aware of structured-design and object-oriented design programming methodologies
- Learn what an algorithm is and explore problem-solving techniques
- Discover what a compiler is and what it does
- Examine a C++ program
- Explore how a C++ program is processed

When data is being organized, a programmer's highest priority is to organize it in such a way that item insertion, deletion, and lookups (searches) are fast. You have already seen how to store and process data in an array. Because an array is a random access data structure, if the data is properly organized (for example, sorted), we can use a search algorithm, such as a binary search, to effectively find and retrieve an item from the list. However, we know that storing data in an array has its limitations. For example, iteminsertion (especially if the array is sorted) and item deletion can be very time consuming, especially if the list size is very large, because each of these operations requires data movement. To speed up item insertion and deletion, we can use linked lists. Item insertion and deletion in a linked list do not require any data movement; we simply adjust some of the pointers in the list. However, one of the drawbacks of linked lists is that theymust be processed sequentially. That is, to insert or delete an item, or simply search the list for a particular item, we must begin our search at the first node in the list. As you know, a sequential search is good only for very small lists because the average search length of a sequential search is half the size of the list.

9.1 Binary Trees

This chapter discusses how to dynamically organize data so that item insertion, deletion, and lookups are more efficient.

We first introduce some definitions to facilitate our discussion.

Definition: A binary tree, T, is either empty or such that

i. T has a special node called the root node.
ii. T has two sets of nodes, L_T and R_T, called the left subtree and right subtree of T, respectively.
iii. L_T and R_T are binary trees.

A binary tree can be shown pictorially. Suppose that T is a binary tree with a root node A. Let L_A denote the left subtree of A and R_A denote the right subtree of A. Now L_A and R_A are binary trees. Suppose that B is the root node of L_A and C is the root node of R_A. B is called the left child of A; C is called the right child of A. Furthermore, A is called the parent of B and C.

In the diagram of a binary tree, each node of the binary tree is represented as a circle and the circle is labeled by the node. The root node of the binary tree is drawn at the top. The left child of the root node (if any) is drawn below and to the left of the root node. Similarly, the right child of the root node (if any)

is drawn below and to the right of the root node. Children are connected to the parent by an arrow from the parent to the child. An arrow is usually called a directed edge or a directed branch (or simply a branch). Because the root node, B, of L_A is already drawn, we apply the same procedure to draw the remaining parts of L_A. RA is drawn similarly. If a node has no left child, for example, when we draw an arrow from the node to the left child, we end the arrow with three lines. That is, three lines at the end of an arrow indicate that the subtree is empty.

The diagram in Figure 9-1 is an example of a binary tree. The root node of this binary tree is A. The left subtree of the root node, which we denoted by L_A, is the set L_A = {B, D, E, G} and the right subtree of the root node, which we denote by R_A, is the set R_A = {C, F, H}. The root node of the left subtree of A—that is, the root node of L_A—is node B. The root node of R_A is C, and so on. Clearly, L_A and R_A are binary trees. Because three lines at the end of an arrow mean that the subtree is empty, it follows that the left subtree of D is empty.

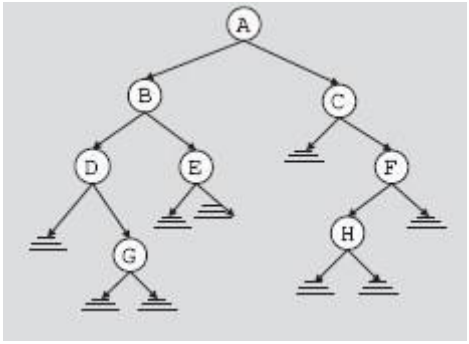


FIGURE 9-1 Binary tree

In Figure 9-1, the left child of A is B and the right child of A is C. Similarly, for node F, the left child is H and node F has no right child.

Example 9-1 shows nonempty binary trees.

EXAMPLE 9-1

Figure 9-2 shows binary trees with one, two, or three nodes.

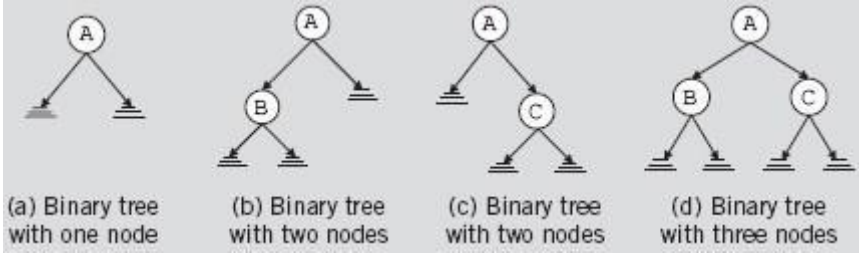


FIGURE 9-2 Binary tree with one, two, or three nodes

In the binary tree of Figure 9-2(a), the root node is A, L_A = empty, and R_A = empty.

In the binary tree of Figure 9-2(b), the root node is A, L_A = {B}, and R_A = empty. The root node of L_A = B, L_B = empty, and R_B = empty.

In the binary tree of Figure 9-2(c), the root node is A, L_A = empty, R_A = {C}. The root node of R_A = C, L_C = empty, and R_C = empty.

In the binary tree of Figure 9-2(d), the root node is A, L_A = {B}, R_A = {C}. The root node of L_A = B, L_B = empty, R_B = empty. The root node of R_A = C, L_C = empty, R_C = empty.

EXAMPLE 9-2

This example shows other cases of nonempty binary trees with three nodes. See Figure 9-3.

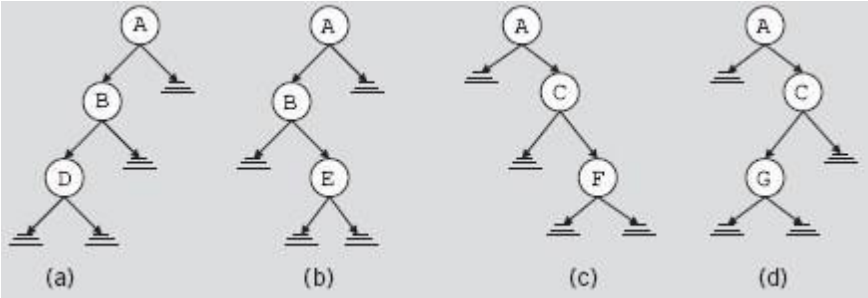


FIGURE 9-3 Various binary trees with three nodes

As you can see from the preceding examples, every node in a binary tree has at most two children. Thus, every node, other than storing its own information, must keep track of its left subtree and right subtree. This implies that every node has two pointers, llink and rlink. The pointer llink points to the root node of the left subtree; the pointer rlink points to the root node of the right subtree.

The following struct defines the node of a binary tree:

```
template <class elemType>
struct binaryTreeNode
{
        elemType info; binaryTreeNode<elemType>
        *llink;
        binaryTreeNode<elemType> *rlink;
};
```

From the definition of the node, it is clear that for each

- node, The data is stored in info.
- A pointer to the left child is stored in llink.
- A pointer to the right child is stored in rlink.

Furthermore, a pointer to the root node of the binary tree is stored outside the binary tree in a pointer

variable, usually called the root, of type binaryTreeNode. Thus, in general, a binary tree looks like the diagram in Figure 9-4.

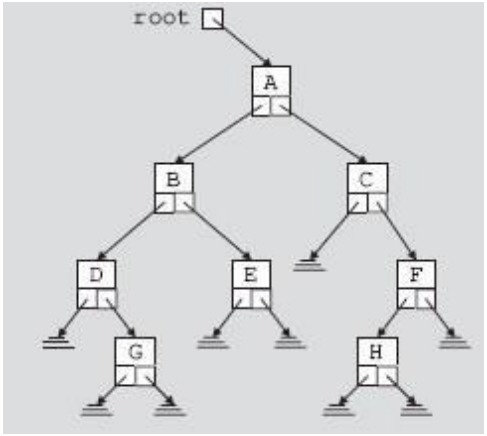


FIGURE 9-4 Binary tree

For simplicity, we will continue to draw binary trees as before. That is, we use circles to represent nodes, and left and right arrows to represent links. As before, three lines at the end of an arrow mean that the subtree is empty.

Before we leave this section, let us define a few more terms.

A node in the binary tree is called a leaf if it has no left and right children. Let U and V be two nodes in the binary tree T. U is called a parent of V if there is a branch from U to V. A path from a node X to a node Y in the binary tree is a sequence of nodes $X_0, X_1, \ldots, X_n$ such that

i. $X = X_0, X_n = Y$
ii. X_{i-1} is the parent of X_i for all $i = 1, 2, \ldots, n$. That is, there is a branch from X_0 to X_1, X_1 to $X_2, \ldots, X_{i-1}$ to $X_i, \ldots, X_{n-1}$ to X_n.

Because the branches go only from a parent to its children, from the previous discussion it is clear that in a binary tree, there is a unique path from the root to every node in the binary tree.

Definition: The level of a node in a binary tree is the number of branches on the path from the root to the node.

Clearly, the level of the root node of a binary tree is 0, and the level of the children of the root node is 1.
Definition: The height of a binary tree is the number of nodes on the longest path from the root to a leaf.
Suppose that a pointer p to the root node of a binary tree is given. We next describe the C++ function height to find the height of the binary tree. The pointer to the root node is passed as a parameter to the function height.

If the binary tree is empty, the height is 0. Suppose that the binary tree is nonempty. To find the height of the binary tree, we first find the height of the left subtree and the height of the right subtree. We then take the maximum of these two heights and add 1 to find the height of the binary tree. To find the height of the left (right) subtree, we apply the same procedure because the left (right) subtree is a binary tree. Therefore, the general algorithm to find the height of a binary tree is as follows. Suppose height(p) denotes the height of the binary tree with root p.

if (p is NULL) height(p) = 0 else height(p) = 1 + max(height(p->llink), height(p->rlink))

Clearly, this is a recursive algorithm. The following function implements this algorithm:

```
template <class elemType>
int height(binaryTreeNode<elemType> *p) const
{ if (p == NULL) return 0; else return 1 + max(height(p->llink),
        height(p->rlink));
}
```

The definition of the function height uses the function max to determine the larger of two integers. The function max can be easily implemented.

Similarly, we can implement algorithms to find the number of nodes and number of leaves in a binary tree.

9.1.1 Copy Tree

One useful operation on binary trees is to make an identical copy of a binary tree. A binary tree is a dynamic data structure; that is, memory for its nodes is allocated and deallocated during program execution. Therefore, if we use just the value of the pointer of the root node to make a copy of a binary tree, we get a shallow copy of the data. To make an identical copy of a binary tree, we need to create as many nodes as there are in the binary tree to be copied. Moreover, in the copied tree, these nodes must appear in the same order as they are in the original binary tree.

Given a pointer to the root node of a binary tree, we next describe the function copyTree, which makes a copy of a given binary tree. This function is also useful in implementing the copy constructor and overloading the assignment operator, as described later in this chapter (see the section, ‘‘Implementing Binary Trees’’).

```
template <class elemType>
void copyTree(binaryTreeNode<elemType>* &copiedTreeRoot, binaryTreeNode<elemType>*
                    otherTreeRoot)
{
        if (otherTreeRoot == NULL)
        copiedTreeRoot = NULL; else
        {
                copiedTreeRoot = new binaryTreeNode<elemType>; copiedTreeRoot->info
                = otherTreeRoot->info; copyTree(copiedTreeRoot->llink, otherTreeRoot-
                >llink); copyTree(copiedTreeRoot->rlink, otherTreeRoot->rlink);
        }
}//end copyTree
```

9.2 Binary Tree Traversal

The item insertion, deletion, and lookup operations require that the binary tree be traversed. Thus, the most common operation performed on a binary tree is to traverse the binary tree, or visit each node of the binary tree. As you can see from the diagram of a binary tree, the traversal must start at the root node because there is a pointer to the root node. For each node, we have two choices:

- Visit the node first.
- Visit the subtrees first.

These choices lead to three different traversals of a binary tree—Inorder, preorder, and postorder.

9.1.2 Inorder Traversal

In an inorder traversal, the binary tree is traversed as follows:

1. Traverse the left subtree.

2. Visit the node.
3. Traverse the right subtree.

9.1.3 Preorder Traversal

In a preorder traversal, the binary tree is traversed as follows:

1. Visit the node.
2. Traverse the left subtree.
3. Traverse the right subtree.

9.1.4 Postorder Traversal

In a postorder traversal, the binary tree is traversed as follows:

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the node.

Clearly, each of these traversal algorithms is recursive.

The listing of the nodes produced by the inorder traversal of a binary tree is called the inorder sequence. The listing of the nodes produced by the preorder traversal of a binary tree is called the preorder sequence. The listing of the nodes produced by the postorder traversal of a binary tree is called the postorder sequence.

Before giving the C++ code for each of these traversals, let us illustrate the inorder traversal of the binary tree in Figure 9-5. For simplicity, we assume that visiting a node means to output the data stored in the node. The section, ‘‘Binary Tree Traversal and Functions as Parameters,’’ located later in this chapter, explains how to modify the binary tree traversal algorithms so that by using a function, the user can specify the action to be performed on a node when the node is visited.

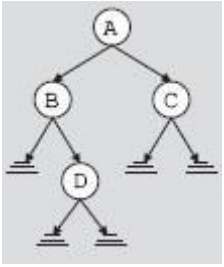


FIGURE 9-5 Binary tree for an inorder traversal

A pointer to the binary tree in Figure 9-5 is stored in the pointer variable root (which points to the node with info A). Therefore, we start the traversal at A.

1. Traverse the left subtree of A; that is, traverse L_A = {B, D}.
2. Visit A.
3. Traverse the right subtree of A; that is, traverse R_A = {C}.

Now, we cannot do Step 2 until we have finished Step 1.

1. Traverse the left subtree of A; that is, traverse L_A = {B, D}. Now L_A is a binary tree with the root node B. Because L_A is a binary tree, we apply the inorder traversal criteria to L_A.

    1.1. Traverse the left subtree of B; that is, traverse L_B = empty.

    1.2. Visit B.

    1.3. Traverse the right subtree of B; that is, traverse R_B = {D}.

As before, first we complete Step 1.1 before going to Step 1.2.

    1.1. Because the left subtree of B is empty, there is nothing to traverse.
    Step 1.1 is completed, so we proceed to Step 1.2.

    1.2. Visit B. That is, output B on an output device. Clearly, the first node printed is B. This completes Step 1.2, so we proceed to Step 1.3.

    1.3. Traverse the right subtree of B; that is, traverse R_B = {D}. Now RB is a binary tree with the root node D. Because R_B is a binary tree, we apply the inorder traversal criteria to R_B.

        1.3.1 Traverse the left subtree of D; that is, traverse L_D = empty.

        1.3.2 Visit D.

        1.3.3 Traverse the right subtree of D; that is, traverse R_D = empty.

        1.3.1. Because the left subtree of D is empty, there is nothing to traverse. Step 1.3.1 is completed, so we proceed to Step 1.3.2.

        1.3.2. Visit D. That is, output D on an output device. This completes Step 1.3.2, so we proceed to Step 1.3.3.

        1.3.3. Because the right subtree of D is empty, there is nothing to traverse. Step 1.3.3 is completed.

This completes Step 1.3. Because Steps 1.1, 1.2, and 1.3 are completed, Step 1 is completed, and so we go to Step 2.

2. Visit A. That is, output A on an output device. This completes Step 2, so we proceed to Step 3.
3. Traverse the right subtree of A; that is, traverse R_A = {C}. Now RA is a binary tree with the root node C. Because R_A is a binary tree, we apply the inorder traversal criteria to R_A.

3.1. Traverse the left subtree of C; that is, traverse L_C = empty.

3.2. Visit C.

3.3. Traverse the right subtree of C; that is, traverse R_C = empty.

3.1 Because the left subtree of C is empty, there is nothing to traverse. Step 3.1 is completed.

3.2. Visit C. That is, output C on an output device. This completes Step 3.2, so we proceed to Step 3.3.

3.3. Because the right subtree of C is empty, there is nothing to traverse. Step 3.3 is completed.

This completes Step 3, which in turn completes the traversal of the binary tree.

Clearly, the inorder traversal of the previous binary tree outputs the nodes in the following order:

Inorder sequence: B D A C

Similarly, the preorder and postorder traversals output the nodes in the following order:

Preorder sequence: A B D C

Postorder sequence: D B C A

As you can see from the walk-through of the inorder traversal, after visiting the left subtree of a node we must come back to the node itself. The links are only in one direction; that is, the parent node points to the left and right children, but there is no pointer from each child to the parent. Therefore, before going to a child, we must somehow save a pointer to the parent node. A convenient way to do this is to write a recursive inorder function because in a recursive call after completing a particular call, the control goes back to the caller. (Later we discuss how to write nonrecursive traversal functions.) The recursive definition of the function to implement the inorder traversal algorithms is as follows:

```
template <class elemType>
void inorder(binaryTreeNode<elemType> *p) const
{
        if (p != NULL)
        { inorder(p->llink); cout << p-
                >info << " ";
                inorder(p->rlink);
        }
}
```

To do the inorder traversal of a binary tree, the root node of the binary tree is passed as a parameter to the function inorder. For example, if the root points to the root node of the binary tree, a call to the function inorder is as follows:

```
inorder(root);
```

Similarly, we can write the functions to implement the preorder and postorder traversals.
The definitions of these functions are given next.

```
template <class elemType>
void preorder(binaryTreeNode<elemType> *p) const
{
```

```
        if (p != NULL)
        { cout << p->info << " ";
                preorder(p->llink);
                preorder(p->rlink);
        }
}

template <class elemType>
void postorder(binaryTreeNode<elemType> *p) const
{
        if (p != NULL)
```

```
        { postorder(p->llink);
                postorder(p-
                >rlink);
                cout << p->info << " ";
        }
}
```

9.1.5 Implementing Binary Trees

The previous sections described various operations that can be performed on a binary tree, as well as the functions to implement these operations. This section describes binary trees as an abstract data type (ADT). Before designing the class to implement a binary tree as an ADT, let us list various operations that are typically performed on a binary tree:

- Determine whether the binary tree is empty.
- Search the binary tree for a particular item.
- Insert an item in the binary tree.
- Delete an item from the binary tree.
- Find the height of the binary tree.
- Find the number of nodes in the binary tree.
- Find the number of leaves in the binary tree.
- Traverse the binary tree.
- Copy the binary tree.

The item search, insertion, and deletion operations all require the binary tree to be traversed. However, because the nodes of a binary tree are in no particular order, these algorithms are not very efficient on arbitrary binary trees. That is, no criteria exist to guide the search on these binary trees, as we will see in the next section. Therefore, we will discuss these algorithms when we discuss special binary trees.

Other than for the search, insertion, and deletion operations, the following class defines binary trees as an ADT. The definition of the node is the same as before. However, for the sake of completeness and easy reference, we give the definition of the node followed by the definition of the class.

```
//***********************************************************
```

```
// class binaryTreeType
// This class specifies the basic operations to implement a binary tree.
//**********************************************************
//Definition of the node template
<class elemType>
struct binaryTreeNode
{
        elemType info; binaryTreeNode<elemType>
        *llink;
        binaryTreeNode<elemType> *rlink;
};

//Definition of the class template
<class elemType>
class binaryTreeType
```

```
{
    public:

        const binaryTreeType<elemType>& operator=(const binaryTreeType<elemType>&);
        //Overload the assignment operator.

        bool isEmpty() const;
        //Returns true if the binary tree is empty; //otherwise,
        returns false.

        void inorderTraversal() const;
        //Function to do an inorder traversal of the binary tree.

        void preorderTraversal() const;
        //Function to do a preorder traversal of the binary tree.

        void postorderTraversal() const;
        //Function to do a postorder traversal of the binary tree.

        int treeHeight() const;
        //Returns the height of the binary tree.

        int treeNodeCount() const;
        //Returns the number of nodes in the binary tree.

        int treeLeavesCount() const;
        //Returns the number of leaves in the binary tree.

        void destroyTree();
        //Deallocates the memory space occupied by the binary tree.
        //Postcondition: root = NULL;
```

```
        binaryTreeType(const binaryTreeType<elemType>& otherTree);
        //copy constructor

        binaryTreeType(); //default
        constructor

        ~binaryTreeType();
        //destructor
protected:
        binaryTreeNode<elemType> *root;
         private:
```

```
void copyTree(binaryTreeNode<elemType>* &copiedTreeRoot,
binaryTreeNode<elemType>* otherTreeRoot);
//Makes a copy of the binary tree to which
//otherTreeRoot points. The pointer copiedTreeRoot
//points to the root of the copied binary tree.

void destroy(binaryTreeNode<elemType>* &p);
//Function to destroy the binary tree to which p points.
//Postcondition: p = NULL

void inorder(binaryTreeNode<elemType> *p) const;
//Function to do an inorder traversal of the binary
//tree to which p points.

void preorder(binaryTreeNode<elemType> *p) const;
//Function to do a preorder traversal of the binary
//tree to which p points.

void postorder(binaryTreeNode<elemType> *p)
const; //Function to do a postorder traversal of the
binary //tree to which p points.
int height(binaryTreeNode<elemType> *p) const;
//Function to return the height of the binary tree //to
which p points.

int max(int x, int y) const; //Returns
the larger of x and y.

int nodeCount(binaryTreeNode<elemType> *p) const; //Function
to return the number of nodes in the binary
//tree to which p points

int leavesCount(binaryTreeNode<elemType> *p) const;
```

```
        //Function to return the number of leaves in the binary //tree
        to which p points
};
```

Notice that the definition of the class binaryTreeType contains the statement to overload the assignment operator, copy constructor, and destructor. This is because the class binaryTreeType contains pointer data members. Recall that for classes with pointer data members, the three things that we must do are explicitly overload the assignment operator, include the copy constructor, and include the destructor.

The definition of the class binaryTreeType contains several member functions that are private members of the class. These functions are used to implement the public member functions of the class and the user need not know their existence. For example, to do an inorder traversal, the function inorderTraversal calls the function inorder and passes the pointer root as a parameter to this function. Suppose that you have the following statement:

```
binaryTreeType<int> myTree;
```

The following statement does an inorder traversal of myTree:

```
myTree.inorder();
```

Also, note that in the definition of the class binaryTreeType, the pointer root is declared as a protected member so that we can later derive special binary trees.

Next, we give the definitions of the member functions of the class binaryTreeType.

The binary tree is empty if root is NULL. So the definition of the function isEmpty is as follows:

```
template <class elemType>
bool binaryTreeType<elemType>::isEmpty() const
{ return (root == NULL);
}
```

The default constructor initializes the binary tree to an empty state; that is, it sets the pointer root to NULL. Therefore, the definition of the default constructor is as follows:

```
template <class elemType>
binaryTreeType<elemType>::binaryTreeType()
{
        root = NULL;
}
```

The definitions of the other functions are as follows:

```
template <class elemType>
void binaryTreeType<elemType>::inorderTraversal() const
{
        inorder(root);
}

template <class elemType>
```

```
void binaryTreeType<elemType>::preorderTraversal() const
{
        preorder(root);
}

template <class elemType>
void binaryTreeType<elemType>::postorderTraversal() const
{
        postorder(root);
}

template <class elemType>
```

```
int binaryTreeType<elemType>::treeHeight() const
{
        return height(root);
}

template <class elemType>
int  binaryTreeType<elemType>::treeNodeCount()  const
{
        return nodeCount(root);
}

template <class elemType>
int  binaryTreeType<elemType>::treeLeavesCount()  const
{
        return leavesCount(root);
}

template <class elemType>
void binaryTreeType<elemType>::inorder(binaryTreeNode<elemType> *p) const
{
        if (p != NULL)
        {
                inorder(p->llink); cout
                << p->info << " ";
                inorder(p->rlink);
        }
}

template <class elemType>
void binaryTreeType<elemType>::preorder(binaryTreeNode<elemType> *p) const
{
        if (p != NULL)
        { cout << p->info << " ";
```

```
                preorder(p->llink);
                preorder(p->rlink);
        }
}

template <class elemType>
void binaryTreeType<elemType>::postorder(binaryTreeNode<elemType> *p) const
{
        if (p != NULL)
        { postorder(p->llink);
                postorder(p-
                >rlink);
                cout << p->info << " ";
        }
}
```

```
template <class elemType>
int binaryTreeType<elemType>::height(binaryTreeNode<elemType> *p) const
{ if (p == NULL) return 0; else return 1 + max(height(p->llink),
        height(p->rlink));
}

template <class elemType>
int binaryTreeType<elemType>::max(int x, int y) const
{
        if (x >= y) return
        x; else
                return y;
}
```

The definitions of the functions nodeCount and leavesCount are left as exercises for you.

Next, we give the definitions of the functions copyTree, destroy, and destroyTree, as well as the definitions of the copy constructor and the destructor, and overload the assignment operator.

The definition of the function copyTree is the same as before; here this function is a member of the class binaryTreeType:

```
template <class elemType>
void binaryTreeType<elemType>::copyTree
(binaryTreeNode<elemType>* &copiedTreeRoot, binaryTreeNode<elemType>* otherTreeRoot)
{
        if (otherTreeRoot == NULL)
        copiedTreeRoot = NULL; else
        {
                copiedTreeRoot = new binaryTreeNode<elemType>; copiedTreeRoot->info
                = otherTreeRoot->info; copyTree(copiedTreeRoot->llink, otherTreeRoot-
                >llink); copyTree(copiedTreeRoot->rlink, otherTreeRoot->rlink);
```

```
        }
} //end copyTree
```

To destroy a binary tree, for each node, first we destroy its left subtree, then its right subtree, and then the node itself. We must use the operator delete to deallocate the memory occupied by each node. The definition of the function destroy is as follows:

```
template <class elemType>
void  binaryTreeType<elemType>::destroy(binaryTreeNode<elemType>*  &p)
{
        if (p != NULL)
        { destroy(p->llink);
                destroy(p-
                >rlink); delete p;
                p = NULL;
```

```
        }
}
```

To implement the function destroyTree, we use the function destroy and pass the root node of the binary tree to the function destroy. The definition of the function destroyTree is as follows:

```
template <class elemType>
void binaryTreeType<elemType>::destroyTree()
{ destroy(root);
}
```

Recall that when a class object is passed by value, the copy constructor copies the value of the actual parameters into the formal parameters. Because the class binaryTreeType has pointer data members, which creates dynamic memory, we must provide the definition of the copy constructor to avoid the shallow copying of data. The definition of the copy constructor, given next, uses the function copyTree to make an identical copy of the binary tree that is passed as a parameter.

```
//copy constructor template <class elemType>
binaryTreeType<elemType>::binaryTreeType
(const binaryTreeType<elemType>&
otherTree)
{
        if (otherTree.root == NULL) //otherTree is empty root
                = NULL;
        else copyTree(root, otherTree.root);
}
```

The definition of the destructor is quite straightforward. When an object of type binaryTreeType goes out of scope, the destructor deallocates the memory occupied by the nodes of the binary tree. The definition of the destructor uses the function destroy to accomplish this task.

```
//destructor
template <class elemType>
```

```
binaryTreeType<elemType>::~binaryTreeType()
{ destroy(root);
}
```

Next, we discuss the function to overload the assignment operator. To assign the value of one binary tree to another binary tree, we make an identical copy of the binary tree to be assigned by using the function copyTree. The definition of the function to overload the assignment operator is as follows:

```
//overloading the assignment operator template
<class elemType>
const binaryTreeType<elemType>& binaryTreeType<elemType>::operator= (const
        binaryTreeType<elemType>& otherTree)
{ if (this != &otherTree) //avoid self-copy
        {
```

```
        if (root != NULL)        //if the binary tree is not empty,
                                 //destroy the binary tree
            destroy(root);
        if (otherTree.root == NULL) //otherTree is empty root
            = NULL;
    else
            copyTree(root, otherTree.root);
    }//end if
    return *this;
}
```

9.3 Binary Search Trees

Now that you know the basic operations on a binary tree, this section discusses a special type of binary tree, called a binary search tree.

Consider the binary tree in Figure 9-6.

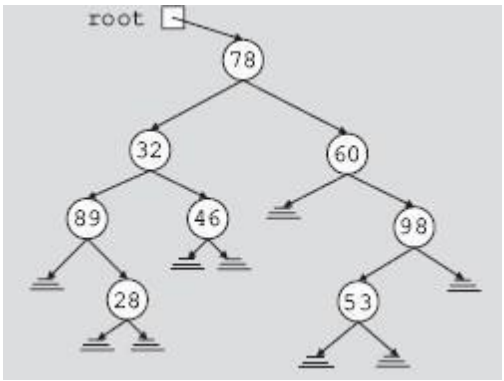


FIGURE 9-6 Arbitrary binary tree

Suppose that we want to determine whether 50 is in the binary tree. To do so, we can use any of the previous traversal algorithms to visit each node and compare the search item with the data stored in the node. However, this could require us to traverse a large part of the binary tree, so the search would be slow. We need to visit each node in the binary tree until either the item is found or we have traversed the entire binary tree because no criteria exist to guide our search. This case is like an arbitrary linked list where we must start our search at the first node, and continue looking at each node until either the item is found or the entire list is searched.

On the other hand, consider the binary tree in Figure 9-7.

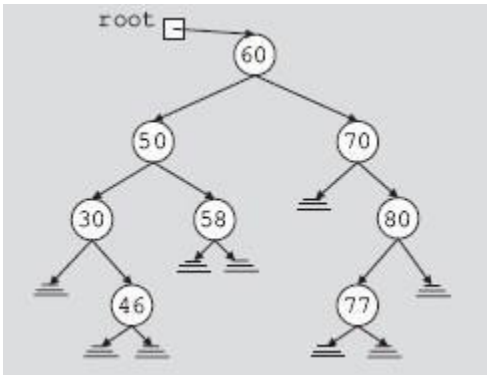


FIGURE 9-7 Binary search tree

In the binary tree in Figure 9-7, the data in each node is

- Larger than the data in its left subtree
- Smaller than the data in its right subtree

The binary tree in Figure 9-7 has some structure. Suppose that we want to determine whether 58 is in this binary tree. As before, we must start our search at the root node. We compare 58 with the data in the root node; that is, we compare 58 with 60. Because 58 ≠ 60 and 58 < 60, it is guaranteed that 58 will not be in the right subtree of the root node. Therefore, if 58 is in the binary tree, it must be in the left subtree of the root node. We follow the left pointer of the root node and go to the node with info 50.We now apply the same criteria at this node. Because 58 > 50, wemust follow the right pointer of this node and go to the node with info 58. At this node we find item 58.

This example shows that every time we move down to a child, we eliminate one of the subtrees of the node from our search. If the binary tree is nicely constructed, the search is very similar to the binary search on arrays.

The binary tree given in Figure 9-7 is a special type of binary tree, called a binary search tree. (In the following definition, by the term key of the node we mean the key of the data item that uniquely identifies the item.)

Definition: A binary search tree, T, is either empty or the following is true:

i. T has a special node called the root node.

ii. T has two sets of nodes, L_T and R_T, called the left subtree and right subtree of T, respectively.

iii. The key in the root node is larger than every key in the left subtree and smaller than every

key in the right subtree.

iv. L_T and R_T are binary search trees.

The following operations are typically performed on a binary search tree:

- Search the binary search tree for a particular item.
- Insert an item in the binary search tree.
- Delete an item from the binary search tree.
- Find the height of the binary search tree.
- Find the number of nodes in the binary search tree.
- Find the number of leaves in the binary search tree.
- Traverse the binary search tree. Copy
- the binary search tree.

Clearly, every binary search tree is a binary tree. The height of a binary search tree is determined the same way as the height of a binary tree. Similarly, the operations to find the number of nodes, to find the number of leaves, and to do inorder, preorder, and postorder traversals of a binary search tree are the same as those for a binary tree. Therefore, we can inherit all of these operations from the binary tree. That is, we can extend the definition of the binary tree by using the principle of inheritance and, hence, define the binary search tree.

The following class defines a binary search tree as an ADT by extending the definition of the binary tree:

```
//*************************************************************
// This class specifies the basic operations to implement a binary search tree.
//*************************************************************
template <class elemType>
class bSearchTreeType: public binaryTreeType<elemType>
{
        public:

                bool search(const elemType& searchItem) const;
                //Function to determine if searchItem is in the binary
                //search tree.
                //Postcondition: Returns true if searchItem is found in the //
                binary search tree; otherwise, returns false.

                void insert(const elemType& insertItem);
                //Function to insert insertItem in the binary search tree.
                //Postcondition: If no node in the binary search tree has the
```

```
        // same info as insertItem, a node with the info insertItem
        // is created and inserted in the binary search tree.

        void deleteNode(const elemType& deleteItem);
        //Function to delete deleteItem from the binary search tree.
        //Postcondition: If a node with the same info as deleteItem //
        is found, it is deleted from the binary search tree.

private:

        void deleteFromTree(binaryTreeNode<elemType>* &p);
        //Function to delete the node to which p points is deleted  //from
        the binary search tree.
        //Postcondition: The node to which p points is deleted from //
        the binary search tree.
```

};

Next, we describe each of these operations.

9.1.6 Search

The function search searches the binary search tree for a given item. If the item is found in the binary search tree, it returns true; otherwise, it returns false. Because the pointer root points to the root node of the binary search tree, we must begin our search at the root node. Furthermore, because root must always point to the root node, we need a pointer, say current, to traverse the binary search tree. The pointer current is initialized to root.

If the binary search tree is nonempty, we first compare the search item with the info in the root node. If they are the same, we stop the search and return true. Otherwise, if the search item is smaller than the info in the node, we follow llink to go to the left subtree; otherwise, we follow rlink to go to the right subtree. We repeat this process for the next node. If the search item is in the binary search tree, our search ends at the node containing the search item; otherwise, the search ends at an empty subtree. Thus, the general algorithm is as follows:

```
if root is NULL
        Cannot search an empty tree, returns false. else
{ current = root;
        while (current is not NULL and not found) if (current->info
                is the same as the search item) set found to true;
                else if(current->info is greater than the search
                item) follow the llink of current else follow the rlink
                of current
}
```

This pseudocode algorithm translates into the following C++ function:

```
template <class elemType> bool
bSearchTreeType<elemType>::
search(const elemType& searchItem) const
{
```

```
binaryTreeNode<elemType> *current;
bool found = false;

if (root == NULL) cerr << "Cannot search the empty
tree." << endl; else
{
        current = root;

        while (current != NULL && !found)
        {
                if (current->info == searchItem)
                found = true; else if (current->info
                > searchItem) current = current-
```

```
                >llink; else current = current-
                >rlink;
            }//end while
        }//end else

        return found;
}//end search
```

9.1.7 Insert

After inserting an item in a binary search tree, the resulting binary tree must also be a binary search tree. To insert a new item, first we search the binary search tree and find the place where the new item is to be inserted. The search algorithm is similar to the search algorithm of the function search. Here we traverse the binary search tree with two pointers—a pointer, say current, to check the current node and a pointer, say trailCurrent, pointing to the parent of current. Because duplicate items are not allowed, our search must end at an empty subtree. We can then use the pointer trailCurrent to insert the new item at the proper place. The item to be inserted, insertItem, is passed as a parameter to the function insert. The general algorithm is as follows:

a. Create a new node and copy insertItem into the new node. Also set llink and rlink of the new node to NULL.
b. if the root is NULL, the tree is empty so make root point to the new node.

```
    else
    { current = root;
        while (current is not NULL) //search the binary tree
        { trailCurrent = current;
            if (current->info is the same as the insertItem)
                Error: Cannot insert duplicate exit
            else if(current->info > insertItem)
                Follow llink of current else
                Follow rlink of current
        }
```

```
            //insert the new node in the binary tree

            if (trailCurrent->info > insertItem) make the new node
                the left child of trailCurrent
            else make the new node the right child of trailCurrent
        }
```

This pseudocode algorithm translates into the following C++ function:

```
template <class elemType>
void bSearchTreeType<elemType>::insert(const elemType& insertItem)
{
    binaryTreeNode<elemType> *current;          //pointer to traverse the tree
    binaryTreeNode<elemType> *trailCurrent;     //pointer behind current
```

```
        binaryTreeNode<elemType> *newNode;  newNode //pointer to create the node
        = new binaryTreeNode<elemType>;
        assert(newNode != NULL); newNode-
        >info = insertItem; newNode->llink =
        NULL; newNode->rlink = NULL;


        if (root == NULL) root =
                newNode;
        else
        {
                Current = root;

                while (current != NULL)
                {
                        trailCurrent = current;

                        if (current->info == insertItem)
                        {
                                cerr << "The insert item is already in the list-"; cerr <<
                                "duplicates are not allowed."<< insertItem << endl;
                                return;
                        }
                        else if (current->info > insertItem) current
                                = current->llink;
                        else current = current->rlink;
                }//end while

                if (trailCurrent->info > insertItem)
                trailCurrent->llink = newNode; else
                trailCurrent->rlink = newNode;
        }
}//end insert
```

9.3.3 Delete

As before, first we search the binary search tree to find the node to be deleted. To help you better understand the delete operation, before describing the function to delete an item from the binary search tree, let us consider the binary search tree given in Figure 9-8.

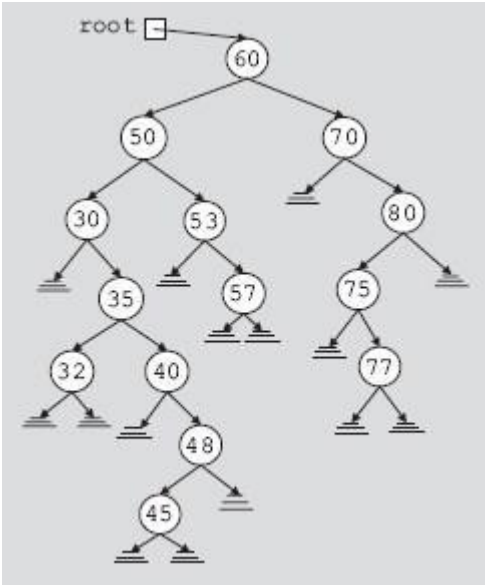


FIGURE 9-8 Binary search tree before deleting a node

After deleting the desired item (if it exists in the binary search tree), the resulting tree must be a binary search tree. The delete operation has four cases, as follows:

Case 1: The node to be deleted has no left and right subtrees; that is, the node to be deleted is a leaf. For example, the node with info 45 is a leaf.

Case 2: The node to be deleted has no left subtree; that is, the left subtree is empty, but it has a nonempty right subtree. For example, the left subtree of node with info 40 is empty and its right subtree is nonempty.

Case 3: The node to be deleted has no right subtree; that is, the right subtree is empty, but it has a nonempty left subtree. For example, the left subtree of node with info 80 is empty and its right subtree is nonempty.

Case 4: The node to be deleted has nonempty left and right subtrees. For example, the left and the right subtrees of node with info 50 are nonempty.

Figure 9-9 illustrates these four cases.

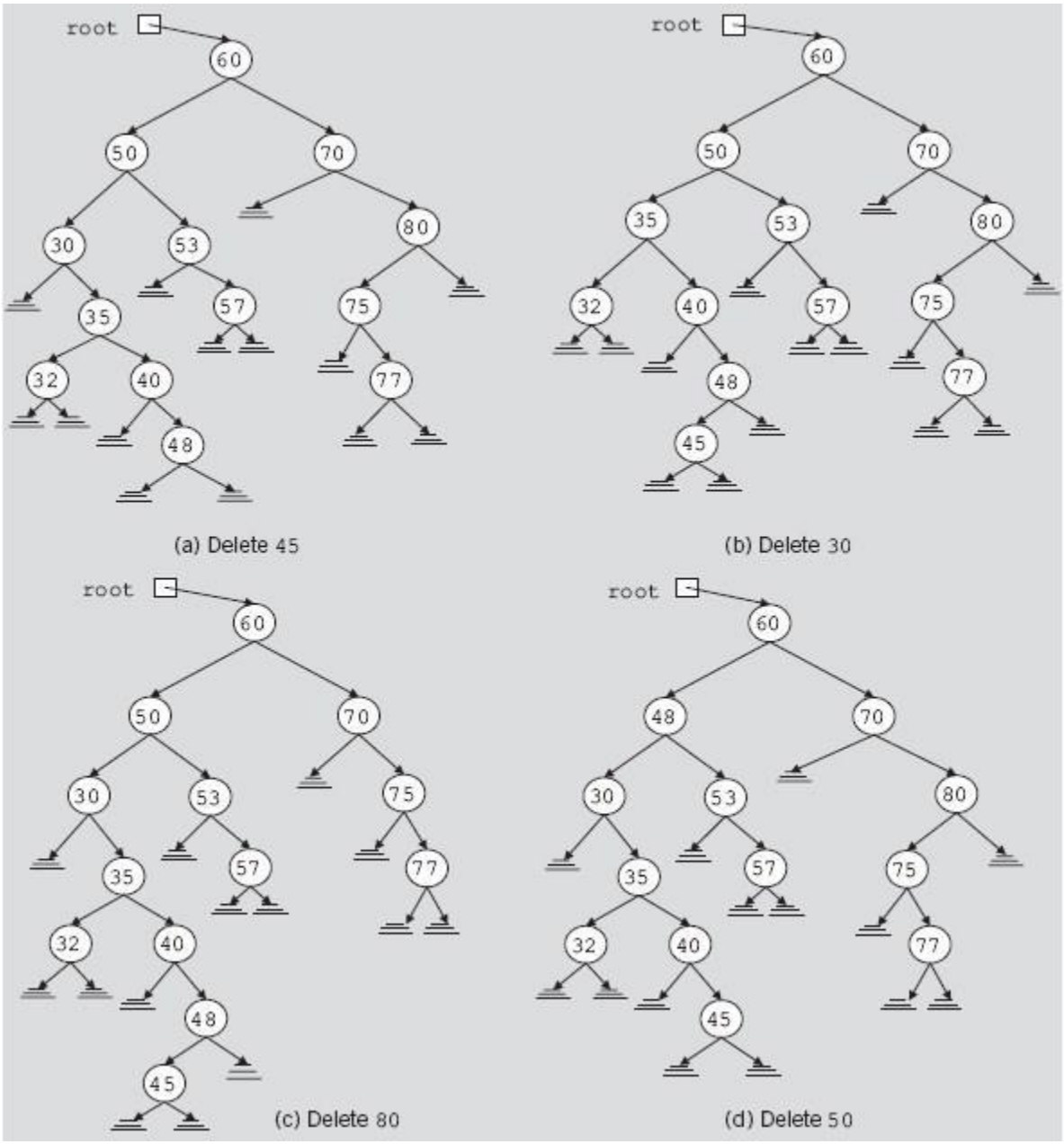


FIGURE 9-9 The binary tree of Figure 9-8 after deleting various items

Case 1: Suppose that we want to delete 45 from the binary search tree in Figure 9-8. We search the binary tree and arrive at the node containing 45. Because this node is a leaf and is the left child of its parent, we can simply set the llink of the parent node to NULL and deallocate the memory occupied by this node. After deleting this node, Figure 9-9(a) shows the resulting binary search tree.

Case 2: Suppose that we want to delete 30 from the binary search tree in Figure 9-8. In this case, the node to be deleted has no left subtree. Because 30 is the left child of its parent node, we make the llink of the parent node point to the right child of 30 and then deallocate the memory occupied by 30. Figure 9-9(b) shows the resulting binary tree.

Case 3: Suppose that we want to delete 80 from the binary search tree of Figure 9-8. The node containing 80 has no right child and is the right child of its parent. Thus, we make the rlink of the parent of 80—that is, 70—point to the left child of 80. Figure 9-9(c) shows the resulting binary tree.

Case 4: Suppose that we want to delete 50 from the binary search tree in Figure 9-8. The node with info 50 has a nonempty left subtree and a nonempty right subtree. Here, we first reduce this case to either Case 2 or Case 3 as follows. To be specific, suppose that we reduce it to Case 3—that is, the node to be deleted has no right subtree. For this case, we find the immediate predecessor of 50 in this binary tree, which is 48. This is done by first going to the left child of 50 and then locating the rightmost node of the left subtree of 50. To do so, we follow the rlink of the nodes. Because the binary search tree is finite, we eventually arrive at a node that has no right subtree. Next, we swap the info in the node to be deleted with the info of its immediate predecessor. In this case, we swap 48 with 50. This reduces to the case wherein the node to be deleted has no right subtree. We now apply Case 3 to delete the node. (Note that because we will delete the immediate predecessor from the binary tree, we, in fact, copy only the info of the immediate predecessor into the node to be deleted.) After deleting 50 from the binary search tree in Figure 9-8, the resulting binary tree is as shown in Figure 9-9(d).

In each case, we see that the resulting binary tree is again a binary search tree.

From this discussion, it follows that to delete an item from a binary search tree, we must do the following:

1. Find the node containing the item (if any) to be deleted.
2. Delete the node.

We accomplish the second step by a separate function, which we will call deleteFromTree. Given a pointer to the node to be deleted, this function deletes the node by taking into account the previous four cases.

The preceding examples show that whenever we delete a node from a binary tree, we adjust one of the pointers of the parent node. Because the adjustment has to be made in the parent node, we must call the function deleteFromTree by using an appropriate pointer of the parent node. For example, suppose that the node to be deleted is 35, which is the right child of its parent node. Further suppose that trailCurrent points to the node containing 30, the parent node of 35. A call to the function deleteFromTree is as follows:

```
deleteFromTree(trailCurrent->rlink);
```

Of course, if the node to be deleted is the root node, then the call to the function deleteFromTree is as follows:

```
deleteFromTree(root);
```

We now define the C++ function deleteFromTree:

```
template <class elemType>
void  bSearchTreeType<elemType>::deleteFromTree(binaryTreeNode<elemType>*  &p)
{
        binaryTreeNode<elemType> *current;                  //pointer to traverse the tree
        binaryTreeNode<elemType> *trailCurrent;  //pointer behind current
        binaryTreeNode<elemType> *temp;  //pointer to delete the node

        if (p == NULL) cerr << "Error: The node to be deleted is NULL."
        << endl; else if(p->llink == NULL && p->rlink == NULL)
        {
```

```
        temp = p; p
        = NULL;
        delete temp;
}
else if(p->llink == NULL)
{
        temp  =  p;  p  =
        temp->rlink;
        delete temp;
}
else if(p->rlink == NULL)
{
        temp = p; p =
        temp->llink;
        delete temp;
}
else
{ current = p->llink;
        trailCurrent = NULL;

        while (current->rlink != NULL)
        { trailCurrent = current; current
                = current->rlink;
        }//end while

        p->info = current->info;

        if (trailCurrent == NULL)                    //current did not move;
                                                     //current == p->llink; adjust p
                p->llink = current->llink; else
        trailCurrent->rlink = current->llink; delete
        current;
}//end else
```

}//end deleteFromTree

Next, we describe the function deleteNode. The function deleteNode first searches the binary search tree to find the node containing the item to be deleted. The item to be deleted, deleteItem, is passed as a parameter to the function. If the node containing deleteItem is found in the binary search tree, the function deleteNode calls the function deleteFromTree to delete the node. The definition of the function deleteNode is given next.

```
template <class elemType>
void bSearchTreeType<elemType>::deleteNode(const elemType& deleteItem)
{
        binaryTreeNode<elemType> *current; //pointer to traverse the tree
        binaryTreeNode<elemType> *trailCurrent; //pointer behind current
        bool found = false;
```

```
        if (root == NULL) cout << "Cannot delete from the empty
        tree." << endl; else
        { current = root;
                trailCurrent = root;

                while (current != NULL && !found)
                {
                        if (current->info == deleteItem)
                        found = true; else
                        {
                                trailCurrent = current;

                                if (current->info > deleteItem)
                                current = current->llink; else
                                current = current->rlink;
                        }
                }//end while

                if (current == NULL) cout << "The delete item is not in the
                tree." << endl; else if (found)
                {
                        if (current == root) deleteFromTree(root);
                else if (trailCurrent->info > deleteItem)
                deleteFromTree(trailCurrent->llink); else
                deleteFromTree(trailCurrent->rlink); }//end if
        }
 }//end deleteNode
```

9.4 Binary Search Tree: Analysis

This section provides an analysis of the performance of binary search trees. Let *T* be a binary search tree with *n* nodes, where *n*> 0. Suppose that we want to determine whether an item, *x*, is in *T*. The performance of the search algorithm depends on the shape of *T*. Let us first consider the worst case. In the worst case, *T* is linear. That is, *T* is one of the forms shown in Figure 9-10.

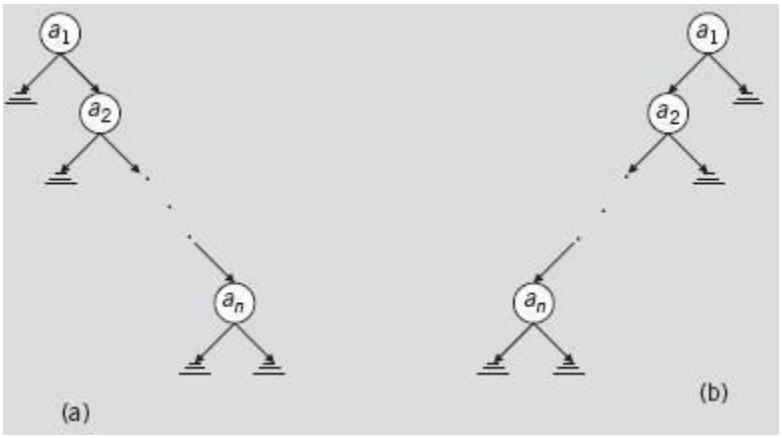


FIGURE 9-10 Linear binary trees

Because *T* is linear, the performance of the search algorithm on *T* is the same as its performance on a linear list. Therefore, in the successful case, on average, the search algorithm makes (*n* + 1) / 2 key comparisons. In the unsuccessful case, it makes *n* comparisons.

Let us now consider the average-case behavior. In the successful case, the search would end at a node. Because there are *n* items, there are *n*! possible orderings of the keys. We assume that all *n*! orderings of the keys are possible. Let *S*(*n*) denote the number of comparisons in the average successful case, and *U*(*n*) denote the number of comparisons in the average unsuccessful case.

The number of comparisons required to determine whether *x* is in *T* is one more than the number of comparisons required to insert *x* in *T*. Furthermore, the number of comparisons required to insert *x* in *T* is the same as the number of comparisons made in the unsuccessful search, reflecting that *x* is not in *T*. From tl

$$S(n) = 1 + \frac{U(0) + U(1) + \ldots + U(n-1)}{n}$$

(Equation 9-1)

It is also known that

$$S(n) = \left(1 + \frac{1}{n}\right) U(n) - 3$$

(Equation 9-2)

Solving Equations (9-1) and (9-2), it can be shown that $U(n) \approx 2.77\log_2 n$ and $S(n) \approx 1.39\log_2 n$.

We can now formulate the following result.

Theorem: Let *T* be a binary search tree with *n* nodes, where *n*> 0. The average number of nodes visited in a search of *T* is approximately $1.39\log_2 n = O(\log_2 n)$ and the number of key comparisons is approximately

$2.77 \log_2 n = O(\log_2 n)$.

9.5 Nonrecursive Binary Tree Traversal Algorithms

The previous sections described how to do the following:

- Traverse a binary tree using the inorder, preorder, and postorder methods.
- Construct a binary tree.
- Insert an item in a binary tree.
- Delete an item from a binary tree.

The traversal algorithms—inorder, preorder, and postorder—discussed earlier are recursive. Because traversing a binary tree is a fundamental operation and recursive functions are somewhat less efficient then their iterative versions, this section discusses the nonrecursive inorder, preorder, and postorder traversal algorithms.

9.1.8 Nonrecursive Inorder Traversal

In the inorder traversal of a binary tree, for each node, the left subtree is visited first, then the node, and then the right subtree. It follows that in an inorder traversal, the first node visited is the leftmost node of the binary tree. For example, in the binary tree in Figure 9-11, the leftmost node is the node with info 28.

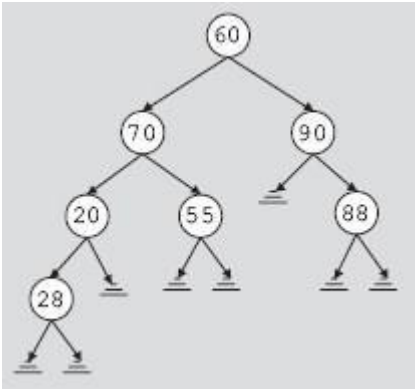


FIGURE 9-11 Binary tree; leftmost node is 28

To get to the leftmost node of the binary tree, we start by traversing the binary tree at the root node and then follow the left link of each node until the left link of a node becomes null. We then back up to the parent node, visit the node, and then move to the right node. Because links go in only one direction, to get back to a node, we must save a pointer to the node before moving to the child node. Moreover, the nodes must be backtracked in the order they were traversed. It follows that while backtracking, the nodes must be visited in a last-in, first-out manner. This can be done by using a stack. We, therefore, save a pointer to a node in a stack. The general algorithm is as follows:

1. current = root; //start

traversing the binary tree at the root node

2. while (current is not NULL or stack is nonempty) if

   (current is not NULL)

   {

   push current into the stack;

   current = current->llink;

   }

   else

   {

   pop stack into current;

   visit current; //visit the node

```
            current = current->rlink;        //move to the right child
        }
```

The following function implements the nonrecursive inorder traversal of a binary tree:

```
template <class elemType>
void  binaryTreeType<elemType>::nonRecursiveInTraversal()  const
{
        stackType<binaryTreeNode<elemType>* > stack; binaryTreeNode<elemType>
        *current;
        current = root;

        while ((current != NULL) || (!stack.isEmptyStack()))
                if (current != NULL)
                {
                        stack.push(current);
                        current = current->llink;
                }
                else
                { current = stack.top(); stack.pop();
                        cout << current->info << " ";
                        current = current->rlink;
                } cout
        << endl;
}
```

9.1.9 Nonrecursive Preorder Traversal

In a preorder traversal of a binary tree, for each node, first the node is visited, then the left subtree is visited, and then the right subtree is visited. As in the case of an inorder traversal, after visiting a node and before moving to the left subtree, we must save a pointer to the node so that after visiting the left subtree, we can visit the right subtree. The general algorithm is as follows:

1. current = root;
   //start the traversal at the root node

2. while (current is not NULL or stack is nonempty) if
   (current is not NULL)
   ```
   { visit current node; push
       current into stack;
       current = current->llink;
   }
   else
   {
       pop stack into current;
       current = current->rlink;          //prepare to visit the right subtree
   }
   ```

The following function implements the nonrecursive preorder traversal algorithm:

```
template <class elemType>
void  binaryTreeType<elemType>::nonRecursivePreTraversal()  const
{
        stackType<binaryTreeNode<elemType>*> stack; binaryTreeNode<elemType>
        *current;
        current = root;
        while ((current != NULL) || (!stack.isEmptyStack()))
                if (current != NULL)
                { cout << current->info << " ";
                        stack.push(current);
                        current = current->llink;
                }
                else
                { current = stack.top();
                        stack.pop();
                        current = current->rlink;
                } cout
        << endl;
}
```

9.1.10 Nonrecursive Postorder Traversal

In a postorder traversal of a binary tree, for each node, first the left subtree is visited, then the right subtree is visited, and then the node is visited. As in the case of an inorder traversal, in a postorder traversal, the first node visited is the leftmost node of the binary tree. Because—for each node—the left and right subtrees are visited before visiting the node, we must indicate to the node whether the left and right subtrees have been visited. After visiting the left subtree of a node and before visiting the node, wemust visit its right subtree. Therefore, after returning from a left subtree, we must tell the node that the right subtree needs to be visited, and after visiting the right subtree we must tell the node that it can now be visited. To do this, other than saving a pointer to the node (to get back to the right subtree and to the node itself), we also save an integer value of 1 before moving to the left subtree and an integer value of 2 before moving to the right subtree. Whenever the stack is popped, the integer value associated with

that pointer is popped as well. This integer value tells whether the left and right subtrees of a node have been visited. The general algorithm is as follows:

1. current = root; //start the traversal at the root node
2. v = 0;
3. if (current is NULL) the binary tree is empty
4. if (current is not NULL)
    a. push current into stack;
    b. push 1 into stack;
    c. current = current->llink;
    d. while (stack is not empty)

        if (current is not NULL and v is 0)

        {

```
            push current and 1 into stack;
            current = current->llink;
    }
    else
    {
            pop stack into current and v;
            if (v == 1)
            {
                    push current and 2 into stack; current
                    = current->rlink;
                    v = 0;
            }
            else
                    visit current;
    }
```

We use two (parallel) stacks: one to save a pointer to a node and another to save the integer value (1 or 2) associated with this pointer. We leave it as an exercise for you write the definition of a C++ function to implement the preceding postorder traversal algorithm.

9.6 Binary Tree Traversal and Functions as Parameters

Suppose that you have stored employee data in a binary search tree, and at the end of the year pay increases or bonuses are to be awarded to each employee. This task requires that each node in the binary search tree be visited and that the salary of each employee be updated. The preceding sections discussed various ways to traverse a binary tree. However, in these traversal algorithms—inorder, preorder, and postorder—whenever we visited a node, for simplicity and for illustration purposes, we only output the data contained in each node. How do we use a traversal algorithm to visit each node and update the data in each node? One way to do so is to first create another binary search tree in which the data in each node is the updated data of the original binary search tree, and then destroy the old binary search tree. This would require extra computer time and perhaps extra memory, and, therefore, is not efficient. Another solution is to write separate traversal algorithms to update the data. This solution requires you to

frequently modify the definition of the class implementing the binary search tree. However, if the user can write an appropriate function to update the data of each employee and then pass this function as a parameter to the traversal algorithms, we can considerably enhance the program's flexibility. This section describes how to pass functions as parameters to other functions.

In C++, a function name without any parentheses is considered a pointer to the function. To specify a function as a formal parameter to another function, we specify the function type, followed by the function name as a pointer, followed by the parameter types of the function. For example, consider the following statements:

```
void fParamFunc1(void (*visit) (int));                    //Line 1 void
fParamFunc2(void (*visit) (elemType&));      //Line 2
```

The statement in Line 1 declares fParamFunc1 to be a function that takes as a parameter any void function that has one value parameter of type int. The statement in Line 2 declares fParamFunc2 to be a function that takes as a parameter any void function that has one reference parameter of type elemType.

We can now rewrite the inorder traversal function of the class binaryTreeType. Alternatively, we can overload the existing inorder traversal functions. To further illustrate function overloading, we will overload the inorder traversal functions. Therefore, we include the following statements in the definition of the class binaryTreeType:

```
void inorderTraversal(void (*visit) (elemType&)); //Function
to do an inorder traversal of the binary tree. //The
parameter visit, which is a function, specifies the //action
to be taken at each node.

void inorder(binaryTreeNode<elemType> *p, void (*visit) (elemType&));
//Function to do an inorder traversal of the binary
//tree, starting at the node specified by the parameter p. //The
parameter visit, which is a function, specifies the //action to
be taken at each node.
```

The definitions of these functions are as follows:

```
template <class elemType>
void binaryTreeType<elemType>::inorderTraversal(void (*visit) (elemType& item))
{ inorder(root, *visit);
}

template <class elemType>
void binaryTreeType<elemType>::inorder(binaryTreeNode<elemType>* p, void
        (*visit) (elemType& item))
{
        if (p != NULL)
        { inorder(p->llink, *visit);
```

```
            (*visit)(p->info);
            inorder(p->rlink, *visit);
        }
}
```

The statement

```
(*visit)(p->info);
```

in the definition of the function inorder makes a call to the function with one reference parameter of the type elemType pointed to by the pointer visit.

Example 9-3 further illustrates how functions are passed as parameters to other functions.

EXAMPLE 9-3

This example shows how to pass a user-defined function as a parameter to the binary tree traversal algorithms. For illustration purposes, we show how to use only the inorder traversal function.

The following program uses the class bSearchTreeType, which is derived from the class binaryTreeType, to build the binary tree. The traversal functions are included in the class binaryTreeType, which are then inherited by the class bSearchTreeType.

```
//*************************************************************
* // This program illustrates how to pass a user-defined function // as a
parameter to the binary tree traversal algorithms.
//*************************************************************
#include <iostream>                                                    //Line 1
#include "binarySearchTree.h"                                          //Line 2


using namespace std;                                                   //Line 3


void print(int& x);                                                    //Line 4
void update(int& x);                                                   //Line 5


int main()                                                             //Line 6
{                                                                      //Line 7
    bSearchTreeType<int> treeRoot;                                     //Line 8


    int num;                                                           //Line 9


    cout << "Line 10: Enter numbers ending with -999"<< endl;          //Line 10
    cin >> num;                                                        //Line 11
```

```
while (num != -999)                                                    //Line 12
{                                                                      //Line 13
        treeRoot.insert(num);                                          //Line 14
        cin >> num;                                                    //Line 15
}                                                                      //Line 16


cout << endl << "Line 17: Tree nodes in inorder: ";                    //Line 17
treeRoot.inorderTraversal(print);                                      //Line 18
```

```
        cout << endl << "Line 19: Tree Height: "<< treeRoot.treeHeight()<< endl << endl;  //Line 19

        cout << "Line 20: ******* Update Nodes *******"<< endl;          //Line 20
        treeRoot.inorderTraversal(update);      //Line 21

        cout << "Line 22: Tree nodes in inorder after "<< "the update: " << endl << " ";  //Line 22
        treeRoot.inorderTraversal(print);      //Line 23 cout << endl <<"Line 24: Tree Height: "<<
        treeRoot.treeHeight() << endl;  //Line 24 return 0;
                                        //Line 25
}                                                                                       //Line 26

void print(int& x)                                                                      //Line 27
{       //Line 28 cout << x << " ";       //Line 29
}                                                                                       //Line 30

void update(int& x)                                                                     //Line 31
{       //Line 32 x = 2 * x;        //Line 33
}                                                                                       //Line 34
```

Sample Run: In this sample run, the user input is shaded.

Line 10: Enter numbers ending with -999

56 87 23 65 34 45 12 90 66 -999

Line 17: Tree nodes in inorder: 12 23 34 45 56 65 66 87 90

Line 19: Tree Height: 4

Line 20: ******* Update Notes *******

Line 22: Tree nodes in inorder after the update:

24 46 68 90 112 130 132 174 180

Line 24: Tree Height: 4

This program works as follows. The statement in Line 8 declares treeRoot to be a binary search tree object, in which the data in each node is of type int. The statements in Lines 11 through 16 build the binary search tree. The statement in Line 18 uses the member function inorderTraversal of treeRoot to traverse the

binary search tree treeRoot. The parameter to the function inorderTraversal, in Line 18, is the function print (defined at Line 27). Because the function print outputs the value of its argument, the statement in Line 18 outputs the data of the nodes of the binary search tree treeNode. The statement in Line 19 outputs the height of the binary search tree.

The statement in Line 21 uses the member function inorderTraversal to traverse the binary search tree treeRoot. In Line 21, the actual parameter of the function inorderTraversal is the function update (defined at Line 31). The function update doubles the value of its argument. Therefore, the statement in Line 21 updates the data of each node of the binary search tree by doubling the value. The statements in Lines 23 and 24 output the nodes and the height of the binary search tree.

9.7 AVL (Height-Balanced) Trees

In the previous sections, you learned how to build and manipulate a binary search tree. The performance of the search algorithm on a binary search tree depends on how the binary tree is built. The shape of the binary search tree depends on the data set. If the data set is sorted, the binary search tree is linear and so the search algorithm would not be efficient. On the other hand, if the tree is nicely built, the search would be fast. In fact, the smaller the height of the tree, the faster the search. Therefore, we want the height of the binary search tree to be as small as possible. This section describes a special type of binary search tree, called the AVL tree (also called the height-balanced tree) in which the resulting binary search is nearly balanced. AVL trees are due to the mathematicians G. M. Adel´sonVel´skii and E. M. Landis and are so named in their honor. The methods of building such binary trees were given by them in 1962.

We begin by defining the following terms.

Definition: A perfectly balanced binary tree is a binary tree such that

i. The heights of the left and right subtrees of the root are equal.
ii. The left and right subtrees of the root are perfectly balanced binary trees.

Figure 9-12 shows a perfectly balanced binary tree.

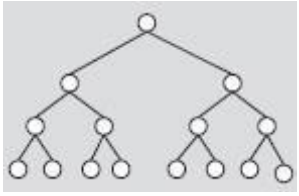


FIGURE 9-12 Perfectly balanced binary tree

Let *T* be a binary tree and *x* be a node in *T*. If *T* is perfectly balanced, then from the definition of the perfectly balanced tree, it follows that the height of the left subtree of *x* is the same as the height of the right subtree of *x*.

It can be proved that, if *T* is a perfectly balanced binary tree of height *h*, then the number of nodes in *T* is 2^h -1. From this, it follows that if the number of items in the data set is not equal to 2^h -1 for some nonnegative integer *h*, then we cannot construct a perfectly balanced binary tree. Moreover, perfectly balanced binary trees are a too stringent refinement.

Definition: An AVL tree (or height-balanced tree) is a binary search tree such that

i. The heights of the left and right subtrees of the root differ by at most 1.

ii. The left and right subtrees of the root are AVL trees.

Figure 9-13 gives examples of AVL and non-AVL trees.

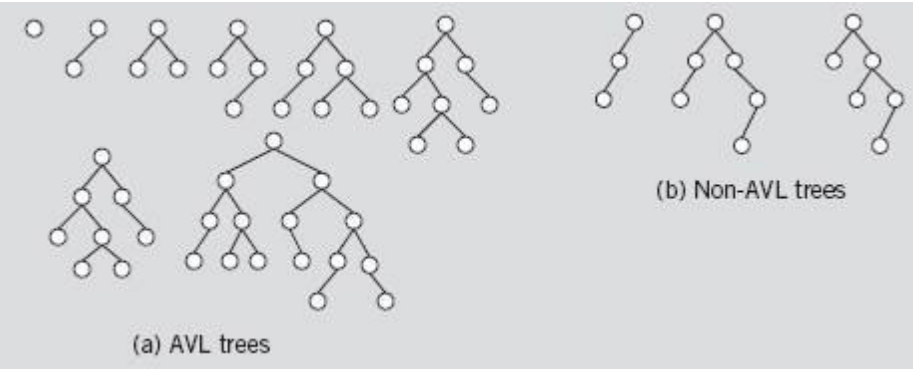


FIGURE 9-13 AVL and non-AVL trees

Let *x* be a node in a binary tree. Let x_l denote the height of the left subtree of *x*, and x_h denote the height of the right subtree of *x.*

Proposition: Let T be an AVL tree and x be a node in T. Then $|x_h - x_l| \leq 1$, where $|x_h - x_l|$ denotes the absolute value of $x_h - x_l$.

Let x be a node in the AVL tree T.

1. If $x_l > x_h$, we say that x is left high. In this case, $x_l = x_h + 1$.
2. If $x_l = x_h$, we say that x is equal high.
3. If $x_h > x_l$, we say that x is right high. In this case, $x_h = x_l + 1$.

Definition: The balance factor of x, written bf(x), is defined by $bf(x) = x_h - x_l$.

Let x be a node in the AVL tree T. Then,

1. If x is left high, bf(x) = -1.
2. If x is equal high, bf(x) = 0.
3. If x is right high, bf(x) = 1.

Definition: Let x be a node in a binary tree. We say that the node x violates the balance criteria if $| x_h - x_l| > 1$, that is, the heights of the left and right subtrees of x differ by more than 1.

From the preceding discussion, it follows that in addition to the data and pointers to the left and right subtrees, one more thing is associated with each node x in the AVL tree T, which is the balance factor of

x. Thus, every node must keep track of its balance factor. To make the algorithms efficient, we store the balance factor of each node in the node itself. Hence, the definition of a node in the AVL tree is as follows:

```
template<class elemType> struct
AVLNode
{
        elemType info;
        int bfactor;                    //balance factor
        AVLNode<elemType> *llink;
        AVLNode<elemType> *rlink;
};
```

Because an AVL tree is a binary search tree, the search algorithm for an AVL tree is the same as the search algorithm for a binary search tree. Other operations, such as finding the height, determining the number of nodes, checking whether the tree is empty, tree traversal, and so on, on AVL trees can be implemented exactly the same way they are implemented on binary trees. However, item insertion and deletion operations on AVL trees are somewhat different from the ones discussed for binary search trees. This is because after inserting (or deleting) a node from an AVL tree, the resulting binary tree must be an AVL tree. Next, we describe these operations.

9.1.11 Insertion

To insert an item in an AVL tree, first we search the tree and find the place where the new item is to be inserted. Because an AVL tree is a binary search tree, to find the place for the new item we can search the AVL tree using a search algorithm similar to the search algorithm designed for binary search trees. If the item to be inserted is already in the tree, the search ends at a nonempty subtree. Because duplicates are not allowed, in this case we can output an appropriate error message. Suppose that the item to be inserted is not in the AVL tree. Then, the search ends at an empty subtree and we insert the item in that subtree. After inserting the new item in the tree, the resulting tree might not be an AVL tree. Thus, we must restore the tree's balance criteria. This is accomplished by traveling the same path, back to the root node, which was followed when the new item was inserted in the AVL tree. The nodes on this path (back to the root node) are visited and either their balance factor is changed, or we might have to reconstruct part of the tree. We illustrate these cases with the help of the following examples.

Consider the AVL tree of Figure 9-14 (a). Let us insert 90 into this AVL tree.

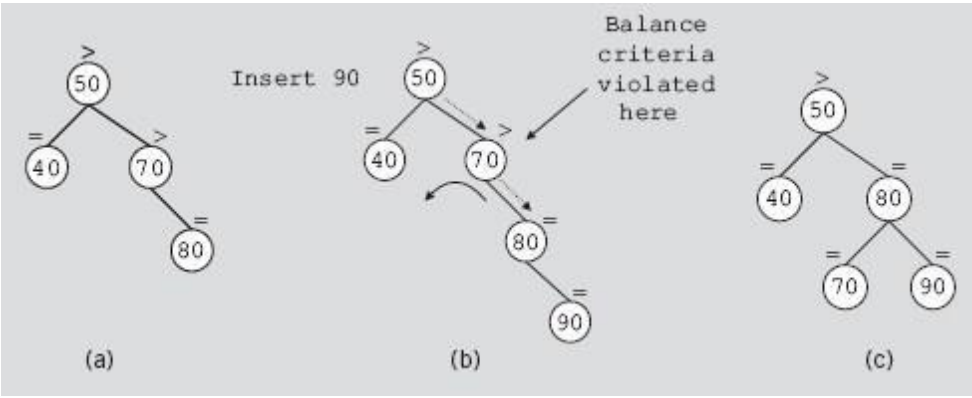


FIGURE 9-14 AVL tree before and after inserting 90

We search the tree starting at the root node to find the place for 90. The dotted arrow shows the path. We insert a node with info 90 and obtain the tree of Figure 9-14(b). The binary tree of Figure 9-14(b) is

not an AVL tree. So we backtrack and go to node 80. Prior to insertion, bf(80) was 0. Because the new node was inserted into the (empty) right subtree of 80, we change its balance factor to 1 (not shown in the figure). Now we go back to node 70. Prior to insertion, bf(70) was 1. Because after insertion the height of the right subtree of 70 is increased, we see that the subtree with root node 70 is not an AVL tree. In this case, we reconstruct this subtree (this is called rotating the tree at root node 70). Therefore, we obtain the tree shown in Figure 9-14(c). The binary tree of Figure 9-14 is an AVL tree.

Now consider the AVL tree of Figure 9-15(a). Let us insert 75 into the AVL tree of Figure 9-15(a).

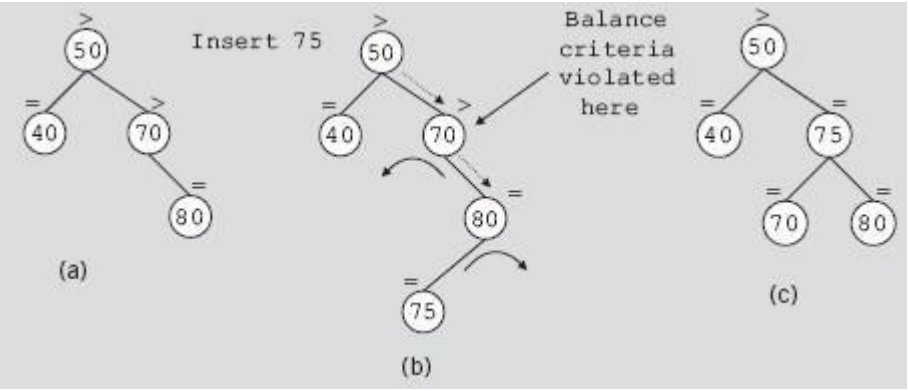


FIGURE 9-15 AVL tree before and after inserting 75

As before, we search the tree starting at the root node. The dotted arrows show the path traversed. After inserting 75, the resulting tree is shown in Figure 9-15(b). After inserting 75, we backtrack. First we go to node 80 and change its balance factor to -1. The subtree with root node 80 is an AVL tree. Now we go back to 70. Clearly, the subtree with root node 70 is not an AVL tree. So we construct this subtree. In this case, we first reconstruct the subtree at root node 80, and then reconstruct the subtree at root node 70 to obtain the tree shown in Figure 9-15(c). (These constructions, that is, rotations, are explained in the next section, ‘‘AVL Tree Rotations.’’)

Notice that in Figures 9-14(c) and 9-15(c), after reconstructing the subtree at the node, the subtree no longer grew in height. At this point, we usually send the message that overall the tree did not gain any height to the remaining nodes on the path back to the root node of the tree, and so the remaining nodes on the path do not need to do anything.

Next, consider the AVL tree of Figure 9-16. Let us insert 95 into this AVL tree.

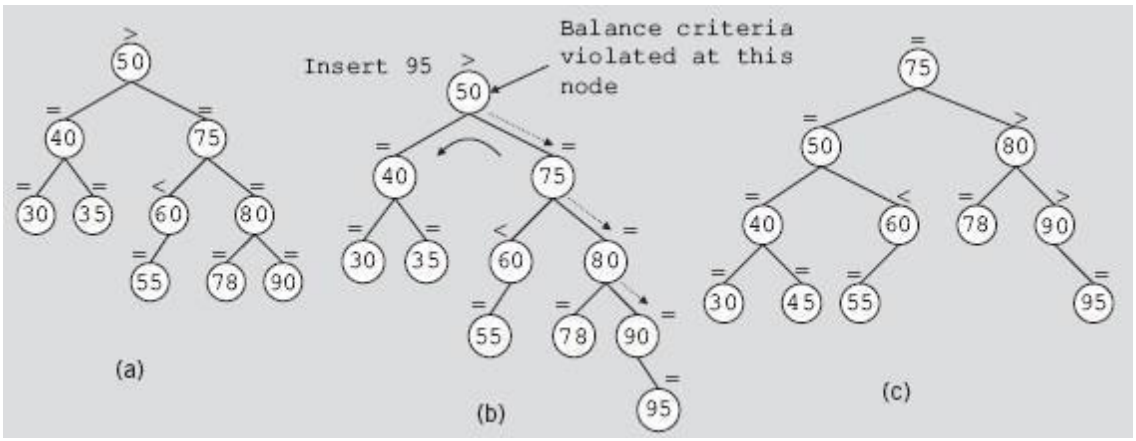


FIGURE 9-16 AVL tree before and after inserting 95

We search the tree and insert 95, as shown in Figure 9-16(b). After inserting 95, we see that the subtrees

with root nodes 90, 80, and 75 are still AVL trees. When backtracking the path, we simply adjust the balance factor of these nodes (if needed). However, when we backtrack to the root node, we discover that the tree at this node is no longer an AVL tree. Prior to insertion, bf(50) was 1, that is, its right subtree was higher than its left subtree. After insertion, the subtree grew in height, thus violating the balance criteria at 50. So we construct the tree at node 50. In this case, the tree will be reconstructed as shown in Figure 9-16(c).

Before discussing the general algorithms for reconstructing (rotating) a subtree, let us consider one more case. Consider the AVL tree as shown in Figure 9-17(a). Let us insert 88 in this tree.

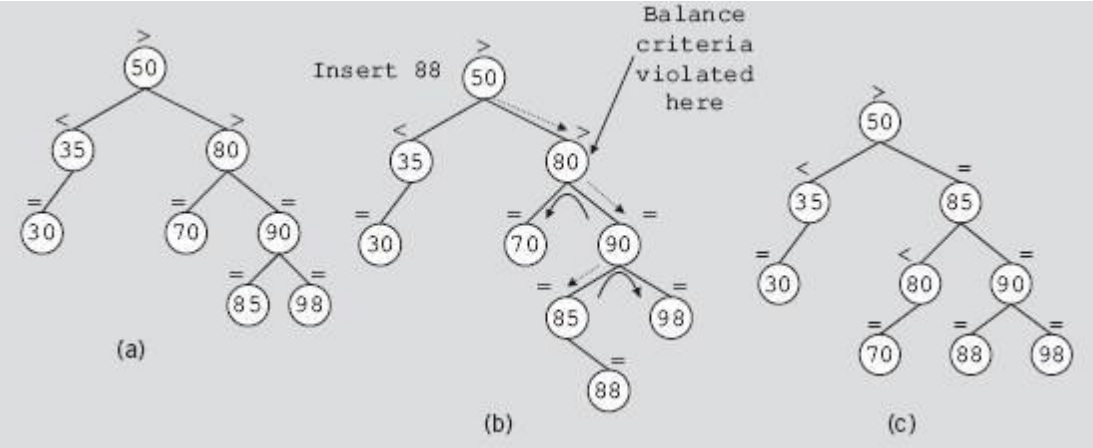


FIGURE 9-17 AVL tree before and after inserting 88

Following the insertion procedure as described previously, we obtain the tree as shown in Figure 917(b). As before, we now backtrack to the root node. We adjust the balance factor of nodes 85 and 90.
When we visit node 80, we discover that at this node we need to reconstruct the subtree. In this case, the subtree is reconstructed as shown in Figure 9-17(c). As before, after reconstructing the subtree, the entire tree is balanced. So for the remaining nodes on the path back to the root node, we would not do anything.

The examples described previously indicate that if part of the tree requires reconstruction, then after reconstructing that part of the tree, we can ignore the remaining nodes on the path back to the root node. (This is, indeed, the case.) Also, after inserting the node, the reconstruction can occur at any node on the path back to the root node.

9.8 AVL Tree Rotations

We now describe the reconstruction procedure, called rotating the tree. There are two types of rotations: left rotation and right rotation. Suppose the rotation occurs at a node x. If it is a left rotation, then certain nodes fromthe right subtree of x move to its left subtree; the root of the right subtree of x becomes the new root of the reconstructed subtree. Similarly, if it is a right rotation at x, certain nodes from the left subtree of x move to its right subtree; the root of the left subtree of x becomes the new root of the reconstructed subtree.

Case 1: Consider Figure 9-18.

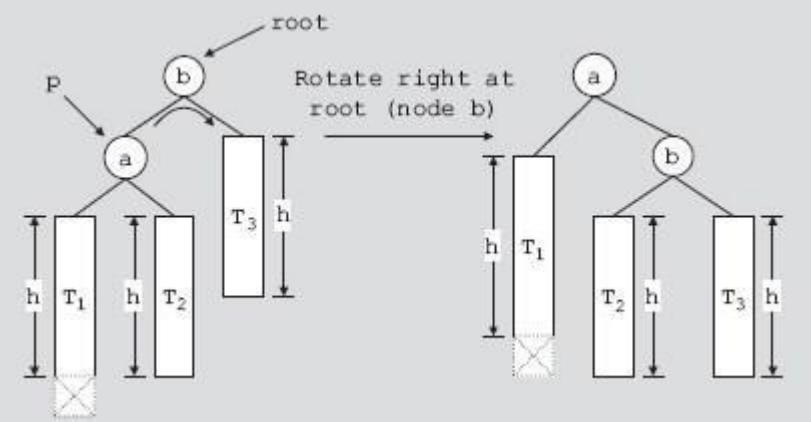


FIGURE 9-18 Right rotation at *b*

In Figure 9-18, subtrees T_1, T_2, and T_3 are of equal height, say h. The dotted rectangle shows an item insertion in T_1, causing the height of the subtree T_1 to increase by 1. The subtree at node *a* is still an AVL tree, but the balance criteria is violated at the root node. We note the following in this tree. Because the tree is a binary search tree,

- Every key in subtree T_1 is smaller than the key in node *a*. Every
- key in subtree T_2 is larger than the key in node *a*.
- Every key in subtree T_2 is smaller than the key in node *b*.

Therefore,

1. We make T2 (the right subtree of node *a*) the left subtree of node *b*.
2. We make node *b* the right child of node *a*.
3. Node *a* becomes the root of the reconstructed tree, as shown in Figure 9-18.

Case 2: This case is a mirror image of Case 1. See Figure 9-19.

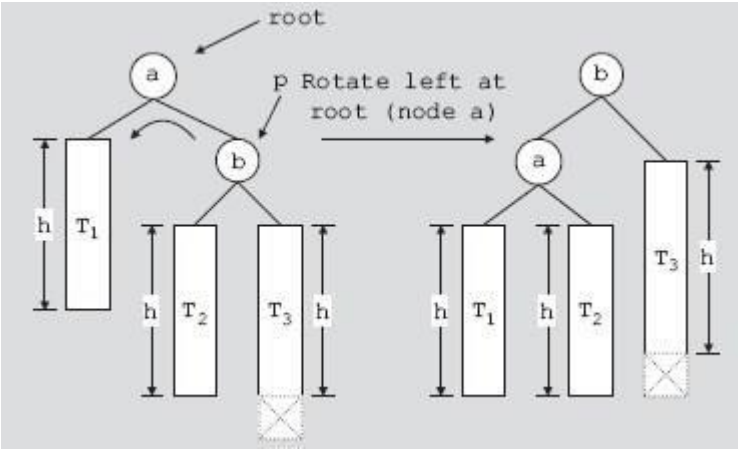


FIGURE 9-19 Left rotation at *a*

Case 3: Consider Figure 9-20.

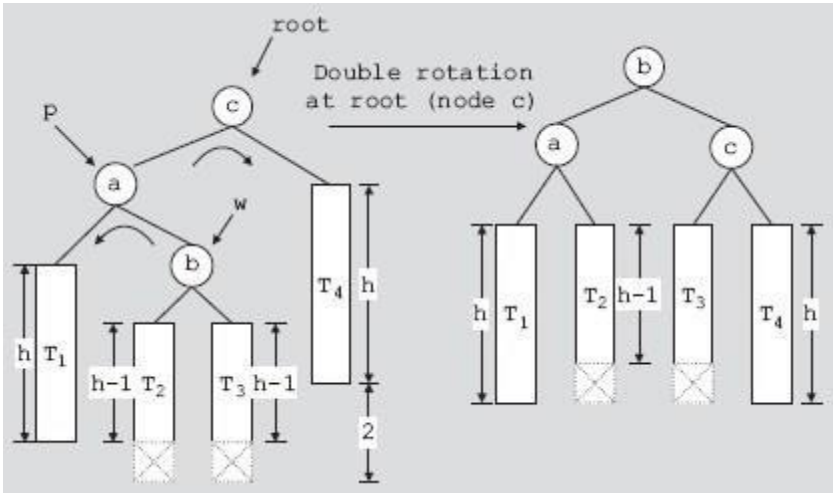


FIGURE 9-20 Double rotation: First rotate left at *a* and then right at *c*

In Figure 9-20, the tree on the left is the tree prior to the reconstruction. The heights of the subtrees are shown in the figure. The dotted rectangle shows that a new item is inserted in the subtree, causing the

subtree to grow in height. The new item is inserted either in T_2 or T_3. We note the following (in the tree prior to reconstruction):

- All keys in T_3 are smaller than the key in node *c*.
- All keys in T_3 are larger than the key in node *b*.
- All keys in T_2 are smaller than the key in node *b*.
- All keys in T_2 are larger than the key in node *a*.
- After insertion, the subtrees with root nodes *a* and *b* are still AVL trees.
- The balance criteria is violated at the root node, *c*, of the tree.
- The balance factors of node *c*, *b*f(*c*) = -1, and node *a*, *b*f(*a*) = 1 are opposite.

This is an example of double rotation. One rotation is required at node *a*, and another rotation is required at node *c*. If the balance factor of the node, where the tree is to be reconstructed, and the balance factor of the higher subtree are opposite, that node requires a double rotation. First, we rotate the tree at node *a* and then at node *c*. Now the tree at node *a* is right high and so we make a left rotation at *a*. Next, because the tree at node *c* is left high, we make a right rotation at *c*. Figure 9-20 shows the resulting tree (which is to the right of the tree after insertion). Figure 9-21, however, shows both rotations in sequence.

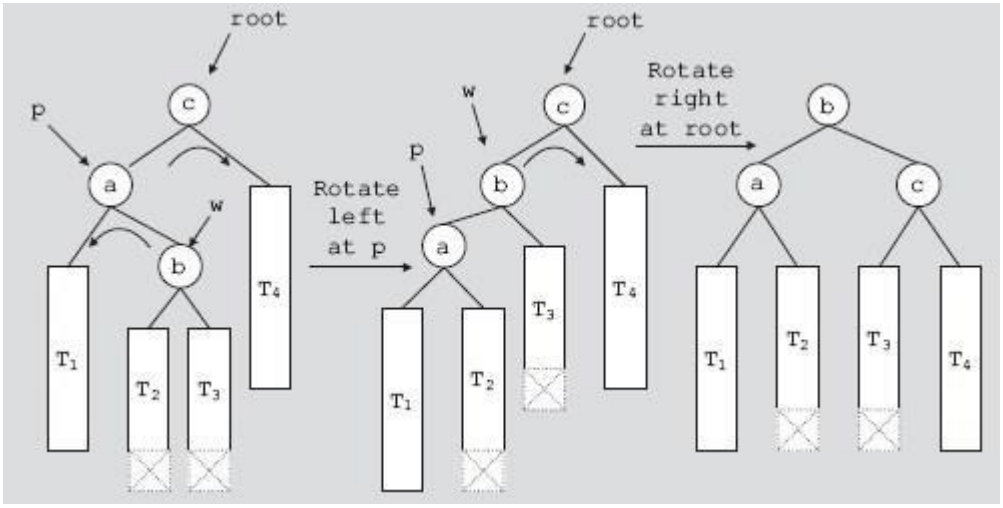


FIGURE 9-21 Left rotation at a followed by *a* right rotation at *c* Case 4: This is a mirror image of Case

3. We illustrate this with the help of Figure 9-22.

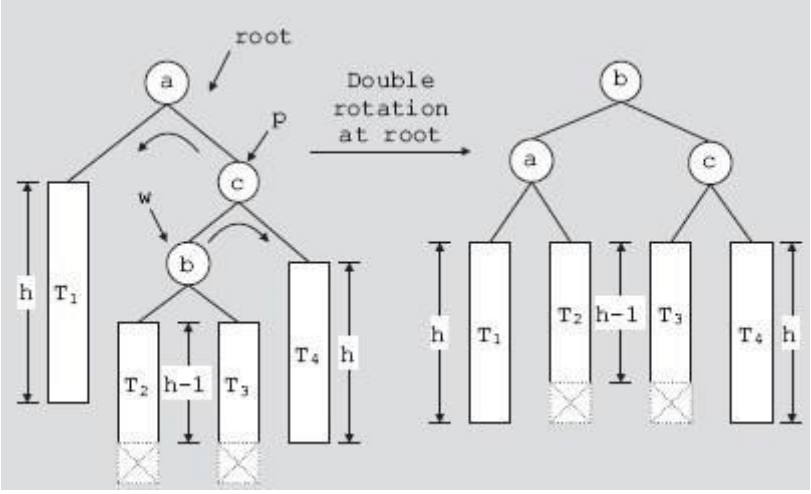


FIGURE 9-22 Double rotation: First rotate right at *c*, then rotate left at *a*

Using these four cases, we now describe what type of rotation might be required at a node.

Suppose that the tree is to be reconstructed, by rotation, at node x. Then, the subtree with root node x requires either a single or a double rotation.

1. Suppose that the balance factor of the node x and the balance factor of the root node of the higher subtree of x have the same sign, that is, both positive or both negative.

1. If these balance factors are positive, make a single left rotation at x. (Prior to insertion, the right subtree of x was higher than its left subtree. The new item was inserted in the right subtree of x, causing the height of the right subtree to increase in height, which in turn violated the balance criteria at x.)
2. If these balance factors are negative, make a single right rotation at x. (Prior to insertion, the left right subtree of x was higher than its right subtree. The new item was inserted in the left subtree of x, causing the height of the left subtree to increase in height, which in turn violated the balance criteria at x.)

2. Suppose that the balance factor of the node x and the balance factor of the higher subtree of x are opposite in sign. To be specific, suppose that the balance factor of node x prior to insertion was 1 and suppose that y is the root node of the left subtree of x. After insertion, the balance factor of node y is 1. That is, after insertion, the right subtree of node y grew in height. In this case, we require a double rotation at x. First, we make a left rotation at y (because y is right high). Then, we make a right rotation at x. The other case, which is a mirror image of this case, is handled similarly.

The following C++ functions implement the left and right rotations of a node. The pointer of the node requiring the rotation is passed as a parameter to the function.

```
template <class elemT>
void rotateToLeft(AVLNode<elemT>* &root)
{
        AVLNode<elemT> *p;                              //pointer to the root of the
                                                        //right subtree of root
        if (root == NULL) cerr << "Error in the tree" << endl; else if (root->rlink ==
        NULL) cerr << "Error in the tree:"<<" No right subtree to rotate." << endl;
        else
```

```
        { p = root->rlink;
                root->rlink = p->llink;                 //the left subtree of p becomes
                                                        //the right subtree of root
                p->llink = root;
                root = p;                               //make p the new root node
        }
}//rotateLeft

template <class elemT>
void rotateToRight(AVLNode<elemT>* &root)
{
        AVLNode<elemT> *p;                              //pointer to the root of
                                                        //the left subtree of root
```

```
    if (root == NULL) cerr << "Error in the tree" << endl; else if (root->llink ==
    NULL) cerr << "Error in the tree:"<< " No left subtree to rotate." << endl;
    else
    { p = root->llink;
        root->llink = p->rlink;                    //the right subtree of p becomes
                                                   //the left subtree of root
        p->rlink = root;
        root = p;                                  //make p the new root node
    }
}//end rotateRight
```

Now that we know how to implement both rotations, we next write the C++ functions, balanceFromLeft and balanceFromRight, which are used to reconstruct the tree at a particular node. The pointer of the node where reconstruction occurs is passed as a parameter to this function. These functions use the functions rotateToLeft and rotateToRight to reconstruct the tree, and also adjust the balance factors of the nodes affected by the reconstruction. The function balanceFromLeft is called when the subtree is left double high and certain nodes need to be moved to the right subtree. The function balanceFromRight has similar conventions.

```
template <class elemT>
void balanceFromLeft(AVLNode<elemT>* &root)
{
    AVLNode<elemT> *p;
    AVLNode<elemT> *w;
    p = root->llink;                               //p points to the left subtree of root
    switch (p->bfactor)
    { case -1: root->bfactor = 0; p->bfactor = 0; rotateToRight(root);
        break; case 0: cerr << "Error: Cannot balance from the left."
        << endl; break; case 1:
            w = p->rlink;
```

```
switch (w->bfactor)                    //adjust the balance factors
{ case -1: root->bfactor = 1; p-
          >bfactor = 0; break;
          case 0: root->bfactor =
                    0; p->bfactor = 0;
                    break;
          case 1: root->bfactor =
                    0; p->bfactor = -
                    1;
```

```
                    }                                   //end switch
                    w->bfactor = 0;
                    rotateToLeft(p); root->llink
                    = p; rotateToRight(root);
        }                                               //end switch;
}                                                       //end balanceFromLeft
```

For the sake of completeness, we also give the definition of the function balanceFromRight:

```
template <class elemT>
void balanceFromRight(AVLNode<elemT>* &root)
{
        AVLNode<elemT> *p;
        AVLNode<elemT> *w;
        p = root->rlink;                                //p points to the left subtree of root
        switch (p->bfactor)
        { case -1: w = p->llink;
                    switch (w->bfactor)                 //adjust the balance factors
                    { case -1: root->bfactor = 0; p-
                            >bfactor = 1; break; case
                            0: root->bfactor = 0; p-
                            >bfactor = 0; break; case
                            1: root->bfactor = -1;
                                    p->bfactor = 0;
                    }                                   //end switch

                    w->bfactor = 0; rotateToRight(p); root->rlink = p;
                rotateToLeft(root); break; case 0: cerr << "Error: Cannot
                balance from the left." << endl; break; case 1: root->bfactor
                = 0; p->bfactor = 0;
                    rotateToLeft(root);
        }                                               //end switch;
}                                                       //end balanceFromRight
```

We now focus our attention on the function insertIntoAVL. The function insertIntoAVL inserts the new

item in the AVL tree. The item to be inserted and the pointer of the root node of the AVL tree are passed as parameters to this function.

The following steps describe the function insertIntoAVL:

1. Create a node and copy the item to be inserted into the newly created node.
2. Search the tree and find the place for the new node in the tree.
3. Insert the new node in the tree.

4. Backtrack the path, which was constructed to find the place for the new node in the tree, to the root node. If necessary, adjust the balance factors of the nodes, or reconstruct the tree at a node on the path.

Because Step 4 requires us to backtrack the path to the root node, and in the binary tree we have links only from the parent to the children, the easiest way to implement the function insertIntoAVL is to use recursion. (Recall that recursion automatically takes care of the backtracking.) This is exactly what we do. The function insertIntoAVL also uses a reference bool parameter, isTaller, to indicate to the parent whether the subtree grew in height or not.

```
template <class elemT>
void insertIntoAVL(AVLNode<elemT>* &root, AVLNode<elemT> *newNode, bool& isTaller)
{
        if (root == NULL)
        {
                root = newNode; isTaller
                = true;
        }
        else if (root->info == newNode->info) cerr << "No
        duplicates are allowed." << endl; else if (root->info >
        newNode->info) //newItem goes in //the left subtree
        {
                insertIntoAVL(root->llink,      newNode,      isTaller);      if
                (isTaller) //after insertion, the subtree grew in height
                        switch (root->bfactor)
                        {
                                case -1:
                                        balanceFromLeft(root)
                                ; isTaller = false; break; case 0:
                                root->bfactor = -1; isTaller =
                                true; break;  case 1: root-
                                >bfactor = 0;
                                        isTaller = false;
```

```
                }//end switch
}//end if
else
{
        insertIntoAVL(root->rlink, newNode, isTaller); if
        (isTaller) //after insertion, the subtree grew in
                        //height switch
                (root->bfactor)
                { case -1: root-
                        >bfacto
                        r = 0;
                        isTaller
                        = false;
                        break;
                        case 0:
```

```
                    root-
                    >bfacto
                    r = 1;
                    isTaller
                    = true;
                    break;
                    case 1:
                            balanceFromRight(root); isTaller
                            = false;
                }//end switch
        }//end else
}//insertIntoAVL
```

Next, we illustrate the insertIntoAVL function and build an AVL tree from scratch. Initially the tree is empty. Each figure shows the item to be inserted as well as the balance factor of each node. An equal sign, =, on the top of a node indicates that the balance factor of this node is 0, the less-than sign, <, indicates that the balance factor of this node is –1, and the greater-than sign, >, indicates that the balance factor of this node is 1.

Figure 9-23 shows how items are inserted into an initially empty AVL tree.

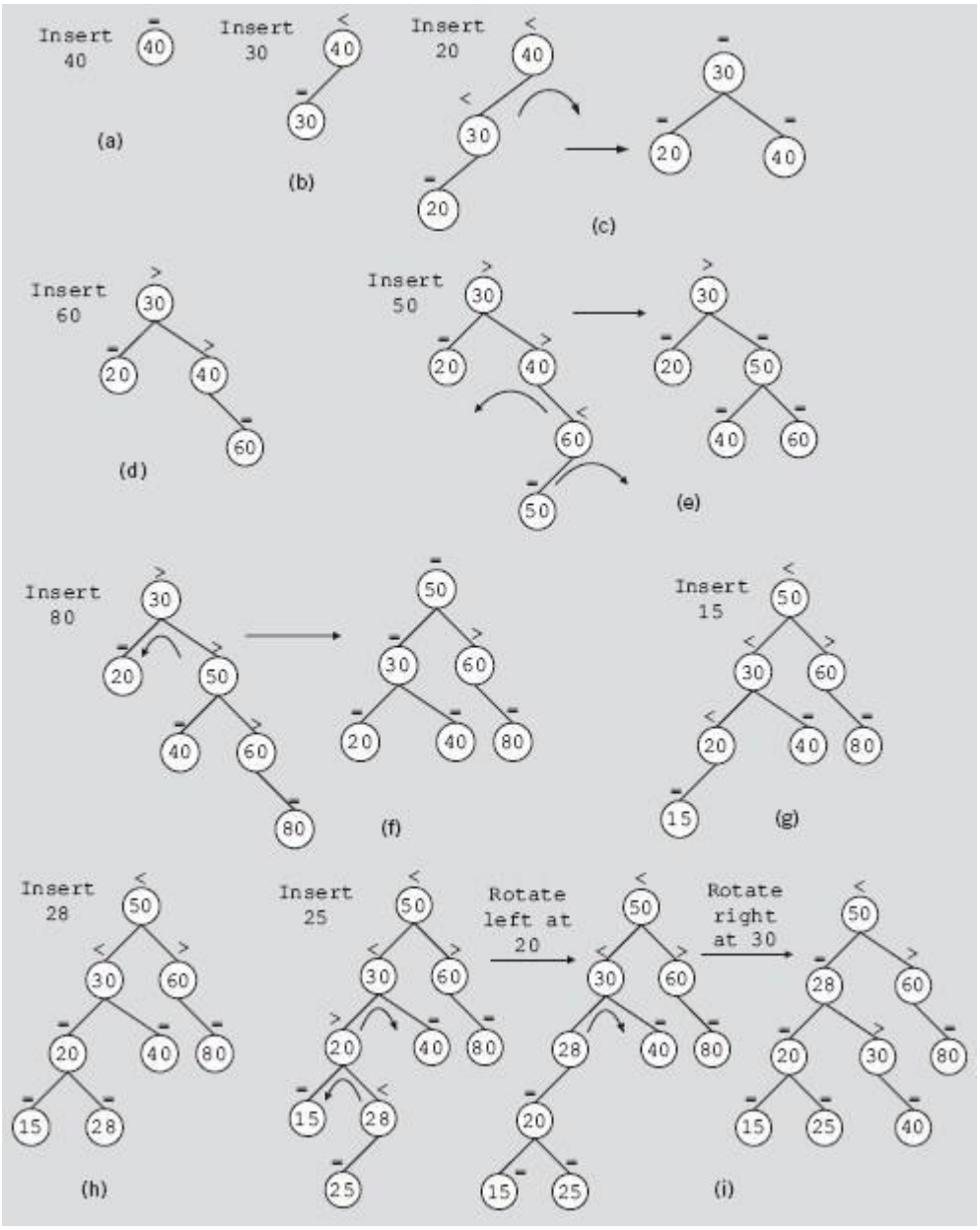


FIGURE 9-23 Item insertion into an initially empty AVL tree

First, we insert 40; see Figure 9-23(a). Next, we insert 30 into the AVL tree. Item 30 is inserted into the left subtree of node 40, causing the left subtree of 40 to grow in height. After insertion, the balance factor of node 40 is –1; see Figure 9-23(b).

Next, we insert 20 into the AVL tree. See Figure 9-23(c). The insertion of 20 violates the balance criteria at node 40. The tree is reconstructed at node 40 by making a single right rotation.

Next, we insert 60 into the AVL tree. See Figure 9-23(d). The insertion of 60 does not require reconstruction; only the balance factor is adjusted at nodes 40 and 30.

Next, we insert 50. See Figure 9-23(e). The insertion of 50 requires the tree to be reconstructed at 40. Notice that a double rotation is made at node 40.

Next, we insert 80; see Figure 9-23(f). The insertion of 80 requires the tree to be reconstructed at node 30.

Next, we insert 15; see Figure 9-23(g). The insertion of node 15 does not require any part of the tree to be reconstructed. We need to only adjust the balance factor of nodes 20, 30, and 50.

Next, we insert 28; see Figure 9-23(h). The insertion of node 28 also does not require any part of the tree to be reconstructed. We need only to adjust the balance factor of node 20.

Next, we insert 25. The insertion of 25 requires a double rotation at node 30. Figure 9-23(i) shows both rotations in sequence. In Figure 9-23(i), the tree is first rotated left at node 20 and then right at node 30.

The following function creates a node, stores the info in the node, and calls the function insertIntoAVL to insert the new node in the AVL tree:

```
template <class elemT>
void insert(const elemT &newItem)
{
        bool isTaller = false;
        AVLNode<elemT> *newNode;

        newNode = new AVLNode<elemT>;
        newNode->info = newItem; newNode-
        >bfactor = 0; newNode->llink = NULL;
        newNode->rlink = NULL;

        insertIntoAVL(root, newNode, isTaller);
}
```

We leave it as an exercise for you to design the class to implement AVL trees as an ADT. (Notice that because the structure of the node of an AVL tree is different from the structure of the node of a binary tree discussed in the beginning of this chapter, you cannot use inheritance to derive the class to implement AVL trees from the class binaryTreeType.)

9.1.12 Deletion from AVL Trees

To delete an item from an AVL tree, first we find the node containing the item to be deleted. The following four cases arise:

Case 1:	The node to be deleted is a leaf.
Case 2:	The node to be deleted has no right child, that is, its right subtree is empty.
Case 3:	The node to be deleted has no left child, that is, its left subtree is empty.
Case 4:	The node to be deleted has a left child and a right child.

Cases 1–3 are easier to handle than Case 4. Let us first discuss Case 4.

Suppose that the node to be deleted, say x, has a left and a right child. As in the case of deletion from a binary search tree, we reduce Case 4 to Case 2. That is, we find the immediate predecessor, say y of x. Then, the data of y is copied into x and now the node to be deleted is y. Clearly, y has no right child.

To delete the node, we adjust one of the pointers of the parent node. After deleting the node, the resulting tree might no longer be an AVL tree. As in the case of insertion into an AVL tree, we traverse the path (from the parent node) back to the root node. For each node on this path, sometimes we need to change only the balance factor, while other times the tree at a particular node is reconstructed. The following steps describe what to do to a node on the path back to the root node. (As in the case of insertion, we use the bool variable shorter to indicate whether the height of the subtree is reduced.) Let p be a node on the path back to the root node. We look at the current balance factor of p.

1. If the current balance factor of p is equal high, the balance factor of p is changed according to if the left subtree of p was shortened or the right subtree of p was shortened. The variable shorter is set to false.
2. Suppose that the balance factor of p is not equal and the taller subtree of p is shortened. The balance factor of p is changed to equal high, and the variable shorter is left as true.
3. Suppose that the balance factor of p is not equal and the shorter subtree of p is shortened. Further suppose that q points to the root of the taller subtree of p.
    a. If the balance factor of q is equal, a single rotation is required at p and shorter is set to false.
    b. If the balance factor of q is the same as p, a single rotation is required at p and shorter is set to true.
    c. Suppose that the balance factors of p and q are opposite. A double rotation is required at p (a single rotation at q and then a single rotation at p). We adjust the balance factors and set shorter to true.

9.1.13 Analysis: AVL Trees

Consider all possible AVL trees of height h. Let T_h be an AVL tree of height h such that Th has the fewest number of nodes. Let T_{hl} denote the left subtree of T_h and T_{hr} denote the right subtree of T_h. Then

$$|T_h| = |T_{hl}| + |T_{hr}| + 1;$$

where $|T_h|$ denotes the number of nodes in T_h.

Because T_h is an AVL tree of height h such that T_h has the fewest number of nodes, it follows that one of the subtrees of T_h is of height h - 1 and the other is of height h - 2. To be specific, suppose T_{hl} is of height h - 1 and T_{hr} is of height h - 2. From the definition of T_h, it follows that T_{hl} is an AVL tree of height h - 1 such

that T_{hl} has the fewest number of nodes among all AVL trees of height h - 1. Similarly, T_{hr} is an AVL tree of height h - 2 that has the fewest number of nodes among all AVL trees of height h - 2. Thus, T_{hl} is of the form T_{h-1} and T_{hr} is of the form T_{h-2}. Hence,

$$|T_h| = |T_{h-1}| + |T_{h-2}| + 1.$$

Clearly,

$$|T_0| = 1$$
$$|T_1| = 2$$

Let $F_{h+2} = |T_h| + 1$. Then,

$$F_{h+2} = F_{h+1} + F_h$$
$$F_2 = 2$$
$$F_3 = 3.$$

This is called a Fibonacci sequence. The solution to Fh is given by

$$F_h \approx \frac{\phi^h}{\sqrt{5}}, \text{ where } \phi = \frac{1+\sqrt{5}}{2}.$$

Hence,

$$|T_h| \approx \frac{\phi^{h+2}}{\sqrt{5}} = \frac{1}{\sqrt{5}}\left[\frac{1+\sqrt{5}}{2}\right]^{h+2}.$$

From this, it can be concluded that

$$h \approx (1.44)\log_2 |T_h|.$$

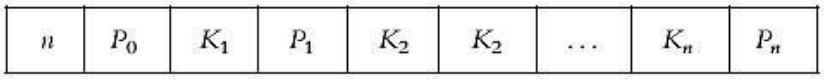
This implies that, in the worst case, the height of an AVL tree with n nodes is approximately $(1.44)\log_2 n$. Because the height of a perfectly balanced binary tree with n nodes is $\log_2 n$, it follows that, in the worst case, the time to manipulate an AVL tree is no more than 44% of the optimum time. However, in general, AVL trees are not as sparse as in the worst case. It can be shown that the average search time of an AVL tree is about 4% more than the optimum.

9.9 B-Trees

In the previous sections of this chapter, we discussed how to build binary search trees and in particular AVL trees to effectively organize data dynamically and effectively search the data. However, the performance of the search depends on the height of the tree. In this section, we discuss B-trees in which the leaves are on the same level and are not too far from the root.

Definition: (m-way search tree) An m-way search tree is a tree in which each node has at most m children, and if the tree is nonempty, it has the following properties:

1. Each node has the following form:



where P_0, P_1, P_2, . . . , P_n are pointers to the subtrees of the node, K_1, K_2, . . . , K_n are keys such that $K_1 < K_2 < \ldots < K_n$, and $n \leq m - 1$.

2. All keys, if any, in the node to which P*i* points are less than K_{i+1}.
3. All keys, if any, in the node to which P*i* points are greater than K_i.

4. The subtrees, if any, to which each P*i* points are m-way search trees

Figure 9-24 shows a 5-way search tree.

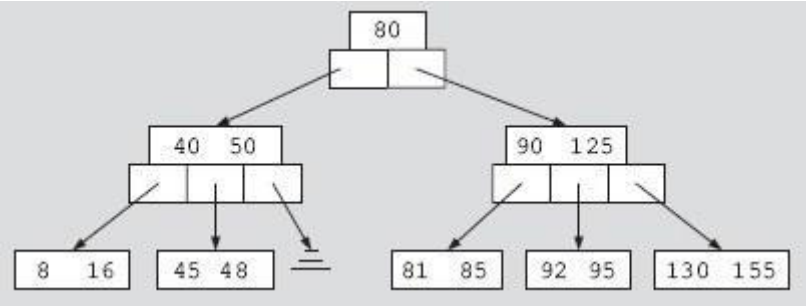


FIGURE 9-24 A 5-way search tree

Definition: (B-tree of order m) A B-tree of order m is an m-way search tree that is either empty, or has the following properties:

1. All leaves are on the same level.
2. 2. All internal nodes except the root have at most m (nonempty) children and at least children. (Note that denotes the ceiling of m/2.)
3. The root has at least 2 children if it is not a leaf, and at most m children.

Figure 9-25 shows a B-tree of order 5.

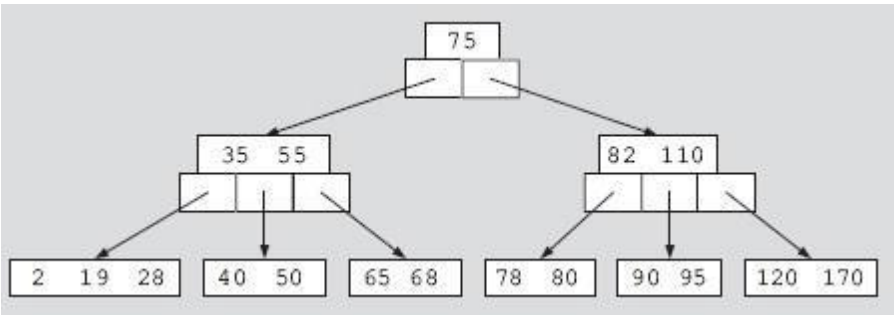


FIGURE 9-25 A B-tree of order 5

Note that the 5-way search tree in Figure 9-24 is not a B-tree of order 5.

The basic operations performed on a B-tree are search the tree, insert an item in the tree, delete an item from the tree, and traverse the tree. In the remainder of this section, we discuss how to implement some of these operations.

Before discussing these properties and describing the structure of a node and the class to implement the

properties of a B-tree, we note the following: Until now, we have passed only data types as parameters to templates. Just like types (data types), constant expressions can also be passed as parameters to templates. For example, consider the following class template:

```
template<class elemType, int size>
class listType
{
        public:
                .
```

```
        .
        .
    private: int
        maxSize; int
        length;
        elemType listElem[size];
};
```

This class template contains an array data member. The array element type and the size of the array are passed as parameters to the class template. To create a list of 100 components of int elements, we use the following statement:

```
listType<int, 100> intList;
```

Next, we give the definitions of the B-tree node and the class that implements the properties of a Btree.

Each node should store the number of keys in the node, the records, and the pointer to subtrees. We use an array to store the records and an array to store the pointers to the subtrees. Thus, the definition of a B-tree node is as follows:

```
template <class recType, int bTreeOrder> struct
bTreeNode
{
    int     recCount;     recType
    list[bTreeOrder - 1];
    bTreeNode *children[bTreeOrder];
};
```

The class implementing the properties of a B-tree must, among others, implement the search, traversal, insertion, and deletion algorithms. The following class implements the basic properties of a B-tree as an ADT:

```
//***********************************************************
// class bTree
// This class specifies the basic operations to implement a B-tree.
```

```
//**********************************************************
* template <class recType, int bTreeOrder> class bTree
{ public:
                bool search(const recType& searchItem);
                //Function to determine if searchItem is in the B-tree.
                //Postcondition: Returns true if searchItem is found in the //
                B-tree; otherwise, returns false.

                void insert(const recType& insertItem); //Function
                to insert insertItem in the B-tree.
                //Postcondition: If insertItem is not in the the B-tree, it //
                is inserted in the B-tree.

                void inOrder();
                //Function to do an inorder traversal of the B-tree.
```

```
        bTree();
        //constructor
        //Add additional members as needed.

    protected:
        bTreeNode<recType, bTreeOrder> *root;
};
```

9.1.14 Search

The function search searches the binary search tree for a given item. If the item is found in the binary search tree, it returns true; otherwise, it returns false. The search must start at the root node. Because, usually, there is more than one item in a node, we must search the array containing the data. Therefore, in addition to the function search, we also write the function searchNode that searches a node sequentially. If item is found, the function searchNode returns true and the location in the array where item is found. If item is not in the node, the function returns false and location points to either the first item that is larger than the search item or one past the last item in the node. The definitions of these functions are as follows:

```
template <class recType, int bTreeOrder> void
bTree<recType, bTreeOrder>::searchNode
    (bTreeNode<recType, bTreeOrder> *current, const recType& item, bool& found, int& location)
{ location = 0;
    while (location < current->recCount && item > current->list[location]) location++;

    if (location < current->recCount && item == current->list[location])
    found = true; else
        found = false;
} //end searchNode

template <class recType, int bTreeOrder>
bool bTree<recType, bTreeOrder>::search(const recType& searchItem)
```

```
{
    bool found = false;
    int location;

    bTreeNode<recType, bTreeOrder> *current;
    current = root;
    while (current != NULL && !found)
    {
        searchNode(current, item, found, location);

        if (!found) current = current-
            >children[location];
    }
```

```
        return found;
} //end search
```

Note that the function searchNode searches the node sequentially. However, because the data in the node is ordered, we can also use a binary search algorithm to search the node. We leave it as an exercise for you to modify the definition of the function searchNode so that it uses a binary search algorithm to search the node.

9.1.15 Traversing a B-Tree

As in the case of a binary tree, a B-tree can be traversed in three ways: inorder, preorder, and postorder.We only discuss the inorder traversal algorithm and leave the others as an exercise.

```
template <class recType, int bTreeOrder>
void bTree<recType, bTreeOrder>::inOrder()
{
        recInorder(root);
} // end inOrder

template <class recType, int bTreeOrder>
void bTree<recType, bTreeOrder>::recInorder(bTreeNode<recType, bTreeOrder> *current)
{
        if (current != NULL)
        {
                recInorder(current->children[0]);       for
                (int i = 0; i < current->recCount; i++)
                { cout << current->list[i] << " ";
                        recInorder(current->children[i + 1]);
                }
        }
} //end recInorder
```

9.1.16 Insertion into a B-Tree

The general algorithm to insert an item in a B-tree is as follows.

Insert Algorithm: Search the tree to see if the key is already in the tree. If the key is already in the tree, output an error messsage. If the key is not in the tree, the search terminates at a leaf. The record is inserted into the leaf if there is room. If the leaf is full, split the node into two nodes and the median key is moved to the parent node. (Note that the median is determined by considering all the keys in the node and the new key to be inserted.) The splitting can propogate upward even as far as the root, causing the tree to increase in height.

Next, we illustrate how the insert algorithm works.

Figures 9-26 to 9-29 shows the insertion of items into an initially empty B-tree of order 5.

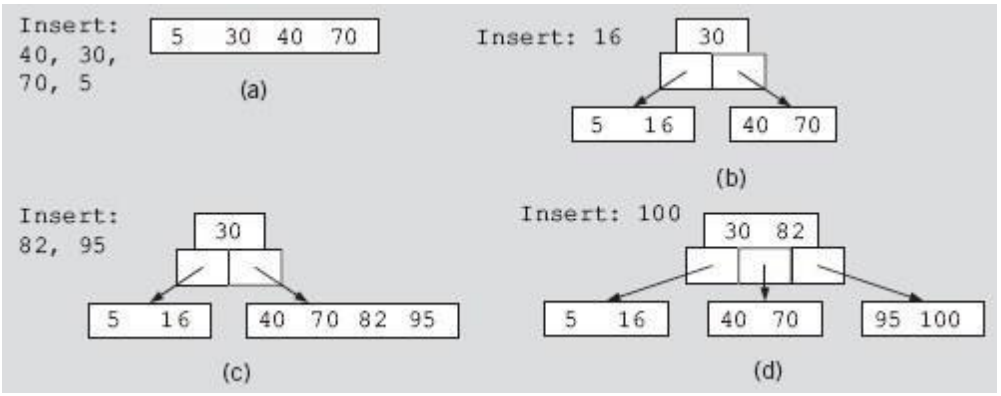


FIGURE 9-26 Item insertion into a B-tree of order 5

The insertion of 40, 30, 70, and 5 is shown in Figure 9-26(a). The insertion of 16 requires to split the root node, causing the height of the tree to increase; see Figure 9-26(b). The insertion of 82 and 95 is shown in Figure 9-26(c). The next item inserted is 100; see Figure 9-26(d). The item 100 is to be inserted in the right child of the root node. However, the right child of the root node is full. So we split this node and move the median key, which is 82, to the parent node. Because the parent node is not full, we can insert 82 in the parent node; see Figure 9-26(d).

Figure 9-27 shows the insertion of 73, 54, 98, and 37.

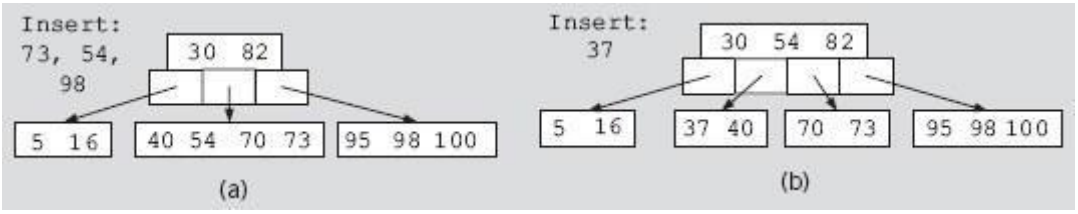


FIGURE 9-27 Insertion of 73, 54, 98, and 37

Note that 73, 54, and 98 are inserted without splitting any node; see Figure 9-27(a). However, the insertion of 37 requires the splitting of a node. The item 37 is to be inserted in the right child of 30. However, the right child of 30 is full, as shown in Figure 9-27(a). So we split the right child of 30, insert 37, and move the median key, which is 54, to the parent node. Because the parent node is not full, the median key 54 is inserted into the parent node; see Figure 9-27(b).

Figure 9-28 shows the insertion of 25, 62, 81, 150, and 79 into the B-tree of Figure 9-27(b).

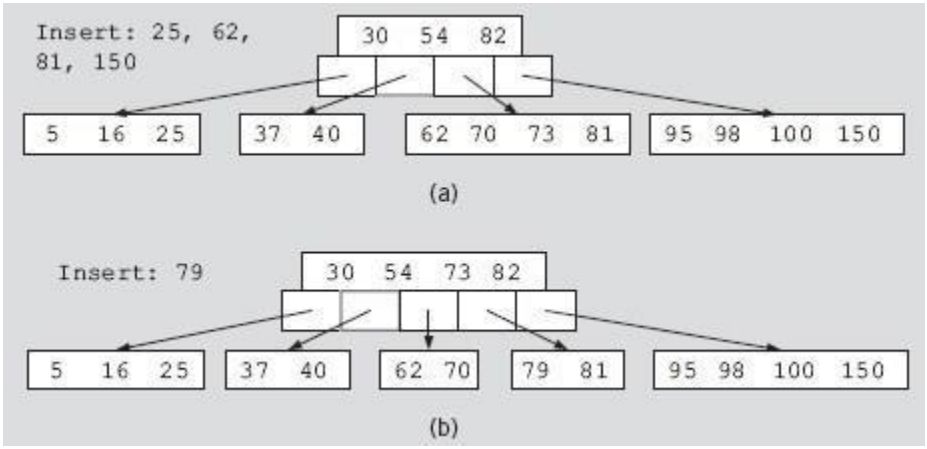


FIGURE 9-28 Insertion of 25, 62, 81, 150, and 79

Note that 25, 62, 81, and 150 are inserted without splitting any node; see Figure 9-28(a). However, the insertion of 79 requires the splitting of a node. The item 79 is to be inserted in the right child of 54. However, the right child of 54 is full; see Figure 9-28(a). So we split the right child of 54, insert 79, and move the median key, which is 73, to the parent node. Because the parent node is not full, the median key 73 is inserted into the parent node; see Figure 9-28(b).

Next, we insert 200, as shown in Figure 9-29.

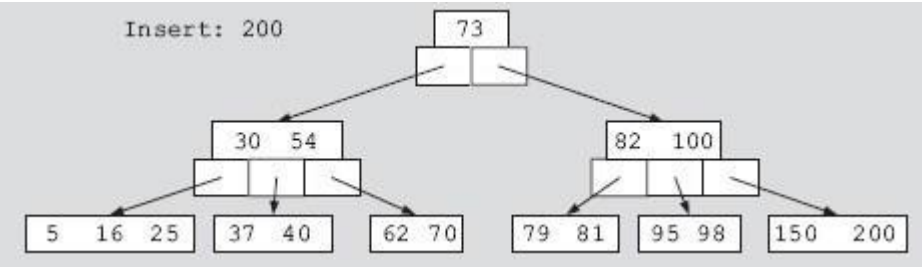


FIGURE 9-29 Insertion of 200

The item 200 is to be inserted in the right child of 82; see Figure 9-28(b). However, the right child of 82 is full. So we split the right child of 82, insert 200 in the node, and move the median key, which is 100, to the parent node. However, the parent node, which is the root node in Figure 9-28(b), is also full. So we split the parent node and move the median key, which is 73, to the new root node; see Figure 9-29. From Figure 9-29, it is clear that the height of the B-tree is increased.

From the previous discussion, it follows that to implement the insertion algorithm, we need algorithms to split a node, insert an item into a node, and move the median key to the parent node. Furthermore, because the insertion of an item might require the splitting of a node and moving the median key to the parent node, the simplest way to implement the insertion algorithm is to use recursion. To trigger the recursion, we will write another function, insertBTree. The definition of the function insert, which uses the function insertBTree, is as follows:

```
template <class recType, int bTreeOrder>
void bTree<recType, bTreeOrder>::insert(const recType& insertItem)
{
        bool isTaller = false;
        recType median;
```

```
bTreeNode<recType, bTreeOrder> *rightChild;
insertBTree(root, insertItem, median, rightChild, isTaller);
 if (isTaller)                                    //the tree is initially empty or the root
                                                  //was split by the function insertBTree
{
        bTreeNode<recType, bTreeOrder> *tempRoot;
        tempRoot = new bTreeNode<recType, bTreeOrder>;
        tempRoot->recCount = 1; tempRoot->list[0] =
        median; tempRoot->children[0] = root; tempRoot-
        >children[1] = rightChild; root = tempRoot;
```

```
        }
} //insert
```

The function insertBTree recursively inserts an item into a B-tree. After inserting an item, it returns true if the height of the tree is to be increased. If the root node is to be split, this function splits the root node, sets isTaller to true, and sends the median key, median, and a pointer, rightChild, of the right child of median to the function insert. The function insert adjusts the root of the B-tree. This function has five parameters: current—a pointer to the B-tree in which to insert an item, insertItem—item to be inserted in the B-tree, median—to return the median key, rightChild—pointer to the right child of median, and isTaller—to indicate if the height of a B-tree is to be increased. In pseudocode, the algorithm is as follows:

```
        if (current is NULL)
        {
                Either the B-tree is empty or the search ends at an empty subtree.
                Set median to insertItem
                Set rightChild to NULL
                Set isTaller to true
        }
        else
        {       Call function searchNode to search the current node
                if insertItem is in the node output
                an error message else
                {
                        call the function insertBTree with appropriate parameters
                        if isTaller is true if current node is not full insert item into
                        current node else call the function splitNode to split the
                        node
                }
}
```

We leave it as an exercise for you to write the definition of the function insertBTree.

The function insertNode inserts an item in the node. Because the keys in the node are in order, the

algorithm to insert a new item is similar to the insertAt function. The function has four parameters: current—a pointer to the node in which to insert the new item, insertItem—the item to be inserted, rightChild—a pointer to the right subtree of the item to be inserted, and insertPosition—the position in the array where to insert the item. The definition of this function is as follows:

```
template <class recType, int bTreeOrder> void
bTree<recType, bTreeOrder>::insertNode
        (bTreeNode<recType, bTreeOrder> *current,const recType& insertItem,
                bTreeNode<recType, bTreeOrder>* &rightChild,int insertPosition)
{
        int index;

        for (index = current->recCount; index > insertPosition;index--)
```

```
        { current->list[index] = current->list[index - 1];
                current->children[index + 1] = current->children[index];
        }

        current->list[index] = insertItem; current->children[index
        + 1] = rightChild;
        current->recCount++;
} //end insertNode
```

The function splitNode splits a node into two nodes and inserts the new item in the relevant node. It returns the median key and a pointer to the second half of the node. The parameter current points to the node to be split, insertItem is the item to be inserted, newChild is a pointer to the right child of the item to be inserted, insertPosition specifies the position where to insert the new item, after splitting the node the parameter rightNode returns a pointer to the right half of the node, and median returns the median key of the node.

```
template <class recType, int bTreeOrder> void
bTree<recType, bTreeOrder>::splitNode
        (bTreeNode<recType, bTreeOrder> *current,const recType& insertItem,
                bTreeNode<recType, bTreeOrder>* rightChild,int insertPosition, bTreeNode<recType,
                bTreeOrder>* &rightNode,recType &median)
{
        rightNode = new bTreeNode<recType, bTreeOrder>;
        int mid = (bTreeOrder - 1) / 2;
        if (insertPosition <= mid)                    //new item goes in the first half of the node
        { int index = 0; int i
                = mid;
                while (i < bTreeOrder - 1)
                { rightNode->list[index] = current->list[i];
                        rightNode->children[index + 1] = current->children[i + 1]; index++;
                        i++;
                }
```

```
        current->recCount = mid;
        insertNode(current, insertItem, rightChild, insertPosition);
        (current->recCount)--;

        median = current->list[current->recCount];

        rightNode->recCount = index;
        rightNode->children[0] = current->children[current->recCount + 1];
}
else                                        //new item goes in the second half of the node
{ int i = mid + 1;
        int index = 0;
```

```
            while (i < bTreeOrder - 1)
            { rightNode->list[index] = current->list[i];
                    rightNode->children[index + 1] = current->children[i + 1];
                    index++; i++;
            }

            current->recCount = mid;
            rightNode->recCount = index;

            median = current->list[mid]; insertNode(rightNode, insertItem,
            rightChild, insertPosition - mid - 1); rightNode->children[0] = current-
            >children[current->recCount + 1];
        }
} //split Node
```

We leave it as an exercise for you to include the functions to insert an item in a B-tree and the functions to search and traverse a B-tree in the class BTree, and write a program to perform these operations on a B-tree.

9.1.17 Deletion from a B-Tree

To delete an item from a B-tree, we search the tree for the item to be deleted, say deleteItem. The following cases arise:

1. If deleteItem is not in the tree, output an appropriate error message.
2. If deleteItem is in the tree, find the node containing the deleteItem. If the node containing the deleteItem is not a leaf, its immediate predecessor (or successor) is in a leaf. So we can swap the immediate predecessor (or successor) with the deleteItem to move the deleteItem into a leaf. We consider the cases to delete an item from a leaf.
    a. If the leaf contains more than the minimum number of keys, delete the deleteItem from the leaf. (In this case, no further action is required.)
    b. If the leaf contains only the minimum number of keys, look at the sibling nodes that are adjacent to the leaf. (Note that the sibling nodes and the leaf have the same parent node.)

i. If one of the sibling nodes has more than the minimum number of keys, move one of the keys from that sibling node to the parent and one key from the parent to the leaf, and then delete deleteItem.

ii. If the adjacent siblings have only the minimum number of keys, then combine one of the siblings with the leaf and the median key from the parent. If this action does not leave the minimum number of keys in the parent node, this process of combining the nodes propogates upward, possibly as far as the root node, which could result in reducing the height of the B-tree.

Next, we illustrate how the deletion process works. Consider the B-tree of order 5 shown in Figure 9-30.

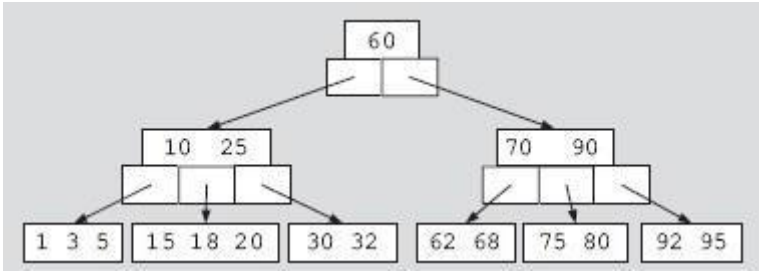


FIGURE 9-30 A B-tree of order 5

Let us delete 18 from this B-tree. Because 18 is in a leaf and the leaf has more than the minimum number of keys, we simply delete 18 from the leaf; see Figure 9-31.

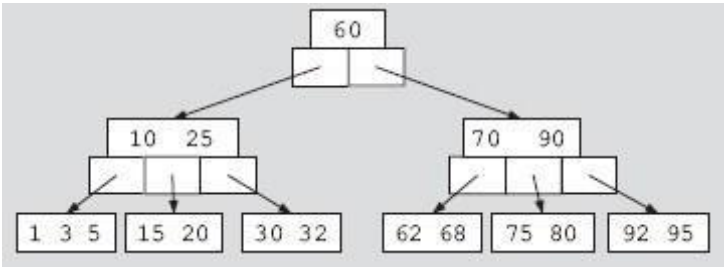


FIGURE 9-31 Deleting 18 from a B-tree of order 5

Next, let us delete 30. Figure 9-32 shows the B-tree before and after deleting 30.

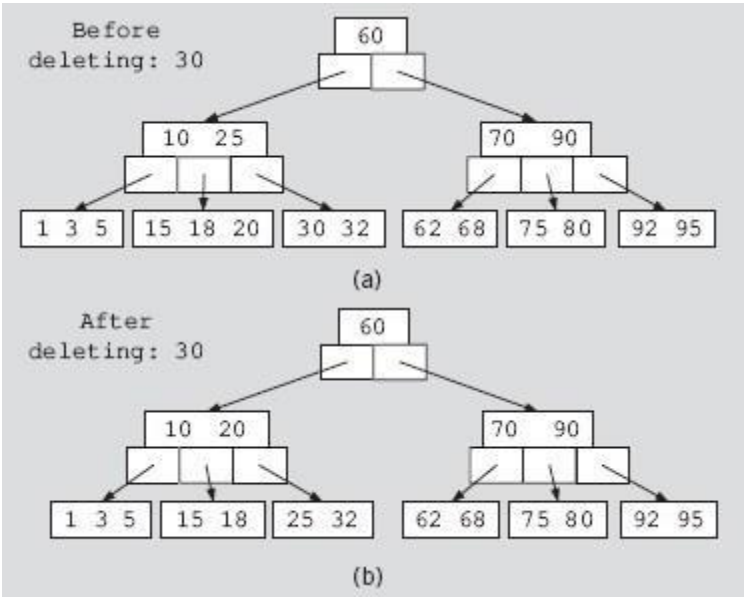


FIGURE 9-32 B-tree before and after deleting 30

The leaf containing 30 has only the minimum number of keys. However, its adjacent sibling has more than the minimum number of keys. So we move 20 from the adjacent sibling to the parent node and then move 25 from the parent node to the leaf; see Figure 9-32(b).

Next, let us delete 70. Figure 9-33 shows the process of deleting 70.

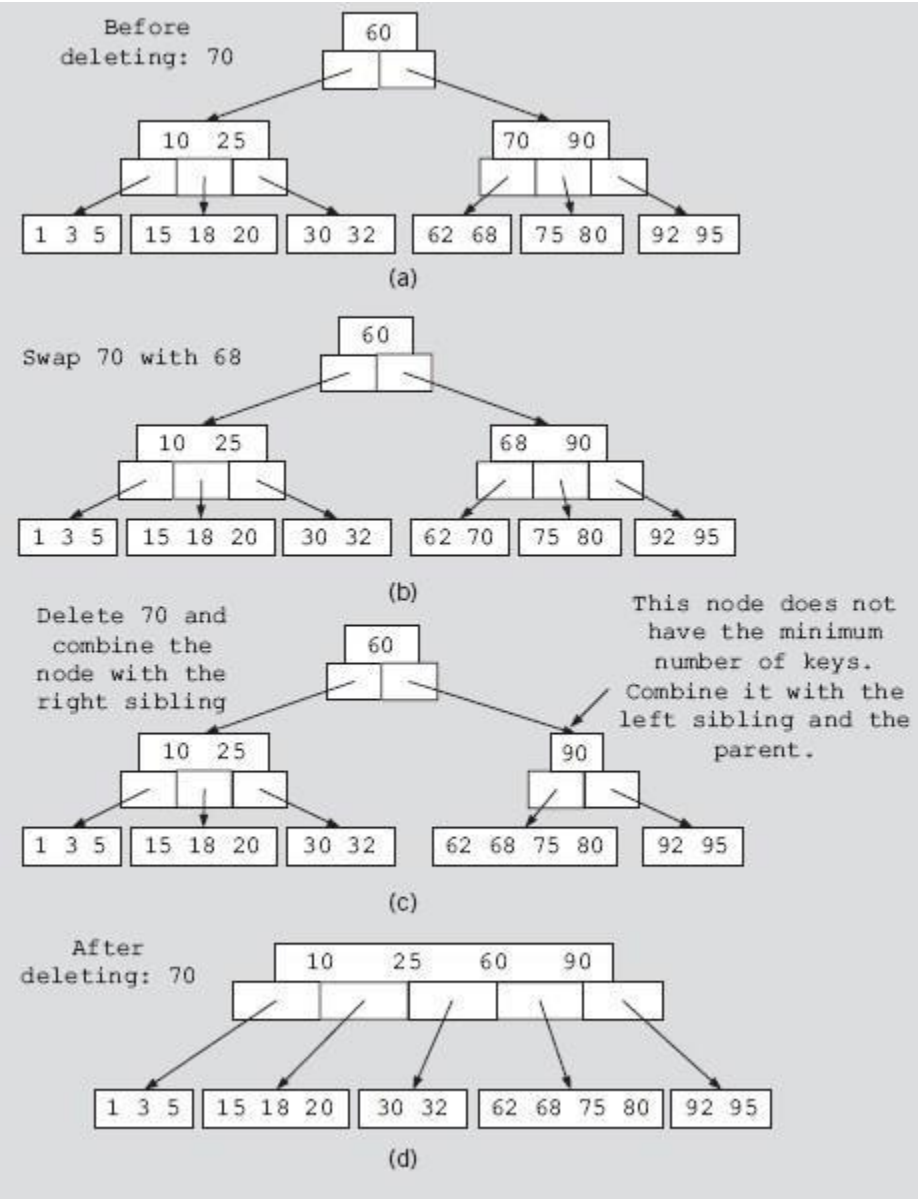


FIGURE 9-33 Deletion of 70 from the B-tree

The node containing 70 is not a leaf. So we swap 70 with its immediate predecessor, which is 68; see Figure 9-33(b). After deleting 70 from the leaf, because the leaf does not have the minimum number of keys, it is combined with its adjacent sibling; see Figure 9-33(c). However, this process does not leave the minimum number of keys in the parent node, which is 90. So we combine this node with its left sibling and their parent, which in this case is the root node; see Figure 9-33(d). Note that the deletion of 70 resulted in reducing the height of the B-tree.

We leave it as an exercise for you to develop the necessary algorithms to delete a record from a B-tree.

Key Terms

1. A binary tree is either empty or it has a special node called the root node. If the tree is nonempty, the root node has two sets of nodes, called the left and right subtrees, such that the left and right subtrees

are also binary trees.

2. The node of a binary tree has two links in it.
3. A node in a binary tree is called a leaf if it has no left and right children.
4. A node U is called the parent of a node V if there is a branch from U to V.

5. A path from a node X to a node Y in a binary tree is a sequence of nodes $X_0, X_1, \ldots, X_n$ such that (a) $X = X_0$, $X_n = Y$ and (b) X_{i-1} is the parent of X_i for all $i = 1, 2, \ldots, n$. That is, there is a branch from X_0 to X_1, X_1 to X_2, . . . , X_{i-1} to X_i, . . . , X_{n-1} to X_n.
6. The level of a node in a binary tree is the number of branches on the path from the root to the node.
7. The level of the root node of a binary tree is 0; the level of the children of the root node is 1.
8. The height of a binary tree is the number of nodes on the longest path from the root to a leaf.
9. In an inorder traversal, the binary tree is traversed as follows: (a) Traverse the left subtree; (b) visit the node; (c) traverse the right subtree.
10. In a preorder traversal, the binary tree is traversed as follows: (a) Visit the node; (b) traverse the left subtree; (c) traverse the right subtree.
11. In a postorder traversal, the binary tree is traversed as follows: (a) Traverse the left subtree; (b) traverse the right subtree; (c) visit the node.
12. A binary search tree T is either empty or
    i. T has a special node called the root node.
    ii. T has two sets of nodes, L_T and R_T, called the left subtree and the right subtree of T, respectively.
    iii. The key in the root node is larger than every key in the left subtree and smaller than every key in the right subtree.
    iv. L_T and R_T are binary search trees.
13. To delete a node from a binary search tree that has both left and right nonempty subtrees, first its immediate predecessor is located, then the predecessor's info is copied into the node, and finally the predecessor is deleted.
14. A perfectly balanced binary tree is a binary tree such that
    i. The heights of the left and right subtrees of the root are equal.
    ii. The left and right subtrees of the root are perfectly balanced binary trees.
15. An AVL (or height-balanced) tree is a binary search tree such that
    i. The heights of the left and right subtrees of the root differ by at most 1.
    ii. The left and right subtrees of the root are AVL trees.
16. Let x be a node in a binary tree. x_l denotes the height of the left subtree of x; x_h denotes the height of the right subtree of x.
17. Let T be an AVL tree and x be a node in T. Then $|x_h - x_l| \leq 1$, where $|x_h - x_l|$ denotes the absolute value of $x_h - x_l$.
18. Let x be a node in the AVL tree T.
    a. If $x_l > x_h$, we say the x is left high. In this case, $x_l = x_h + 1$.

b. If $x_l = x_h$, we say the x is equal high.

c. If $x_h > x_l$, we say the x is right high. In this case, $x_h = x_l + 1$.

19. The balance factor of x, written bf(x), is defined as $bf(x) = x_h - x_l$.

20. Let x be a node in the AVL tree T. Then,

    a. If x is left high, bf(x) = -1.

    b. If x is equal high, bf(x) = 0.

    c. If x is right high, bf(x) = 1.

21. Let x be a node in a binary tree. We say that node x violates the balance criteria if $|x_h - x_l| > 1$, that is, the heights of the left and right subtrees of x differ by more than 1.

22. Every node x in the AVL tree T, in addition to the data and pointers to the left and right subtrees, must keep track of its balance factor.

23. In an AVL tree, there are two types of rotations: left rotation and right rotation. Suppose that the rotation occurs at node, say x. If it is a left rotation, certain nodes from the right subtree of x move to its left subtree; the root of the right subtree of x becomes the new root of the reconstructed subtree.

Similarly, if it is a right rotation at x, certain nodes from the left subtree of x move to its right subtree; the root of the left subtree of x becomes the new root of the reconstructed subtree.

24. A B-tree of order m is an m-way search tree that is either empty, or has the following properties: (1) All leaves are on the same level; (2) All internal nodes except the root have at most m (nonempty) children and at least children. (Note that denotes the ceiling of m/2.); (3) The root has at least 2 children if it is not a leaf, and at most m children.
25. To insert an item into a B-tree, search the tree to see if the record is already in the tree. If the record is already in the tree, output an error messsage. If the record is not in the tree, the search terminates at a leaf. The record is inserted into the leaf if there is room. If the leaf is full, split the node into two nodes and the median record is moved to the parent node. The splitting can propogate upward even as far as the root, causing the tree to increase in height.

Review Questions

1. Mark the following statements as true or false.

a. A binary tree must be nonempty.

b. The level of the root node is 0.

c. If a tree has only one node, the height of this tree is 0 because the number of levels is 0.

d. The inorder traversal of a binary tree always outputs the data in ascending order.

2. There are 14 different binary trees with four nodes. Draw all of them. The binary tree of Figure 9-34 is to be used for Exercises 3 through 8.

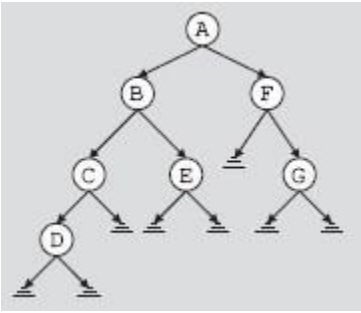


FIGURE 9-34 Binary tree for Exercises 3 through 8

3. Find LA, the node in the left subtree of A.

4. Find RA, the node in the right subtree of A.

5. Find RB, the node in the right subtree of B.

6. List the nodes of this binary tree in an inorder sequence.

7. List the nodes of this binary tree in a preorder sequence.

8. List the nodes of this binary tree in a postorder sequence.

The binary tree of Figure 9-35 is to be used for Exercises 9 through 13.

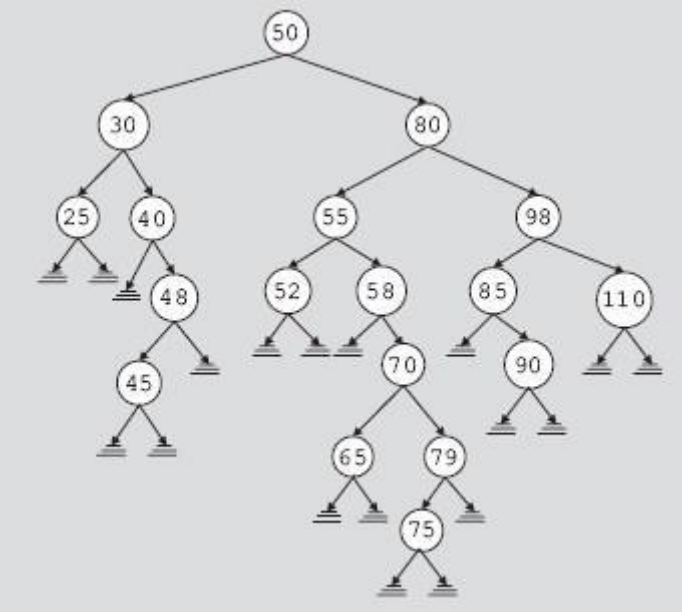


FIGURE 9-35 Binary tree for Exercises 9 through 13

9. List the path from the node with info 80 to the node with info 79.

10. A nodewith info 35 is to be inserted in the tree. List the nodes that are visited by the function insert to insert 35. Redraw the tree after inserting 35.

11. Delete node 52 and redraw the binary tree.

12. Delete node 40 and redraw the binary tree.

13. Delete nodes 80 and 58 in that order. Redraw the binary tree after each deletion.

14. Suppose that you are given two sequences of elements corresponding to the inorder sequence and the preorder sequence. Prove that it is possible to reconstruct a unique binary tree.

15. The nodes in a binary tree in preorder and inorder sequences are as follows:

    preorder: ABCDEFGHIJKLM inorder:

    CEDFBAHJIKGML

Draw the binary tree.

16. Given the preorder sequence and the postorder sequence, show that it might not be possible to reconstruct the binary tree.

17. Insert 100 in the AVL tree of Figure 9-36. The resulting tree must be an AVL tree. What is the balance factor at the root node after the insertion?

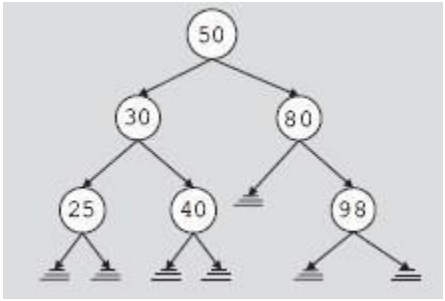


FIGURE 9-36 AVL tree for Exercise 17

18. Insert 45 in the AVL tree of Figure 9-37. The resulting tree must be an AVL tree. What is the balance factor at the root node after the insertion?

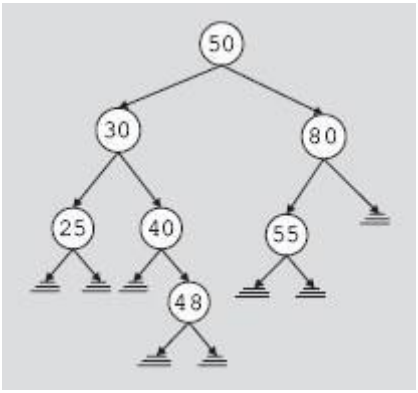


FIGURE 9-37 AVL tree for Exercise 18

19. Insert 42 in the AVL tree of Figure 9-38. The resulting tree must be an AVL tree. What is the balance factor at the root node after the insertion?

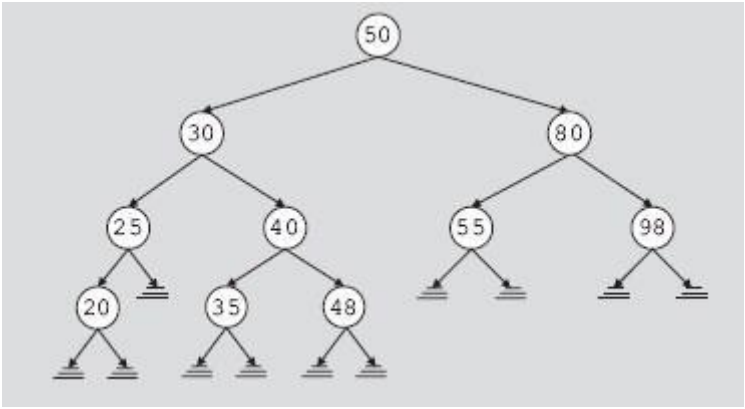


FIGURE 9-38 AVL tree for Exercise 19

20. The keys 24, 39, 31, 46, 48, 34, 19, 5, and 29 are inserted (in the order given) into an initially empty

AVL tree. Show the AVL tree after each insertion. The binary tree of Figure 9-39 is to be used for Exercises 21 to 23.

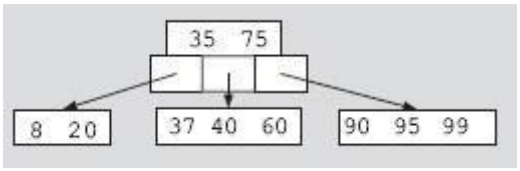


FIGURE 9-39 B-tree of order 5 for Exercises 21 to 23

21. Insert the keys 72, 30, and 50, in this order, into the B-tree of order 5 of Figure 9-39. Show the resulting tree.
22. Insert the keys 38, 45, 55, 80, and 85 into the B-tree of order 5 of Figure 9-39. Show the resulting tree.

23. Insert the keys 2, 30, 42, 10, 96, 15, 50, 82, and 98 into the B-tree of order 5 of Figure 9-39. Show the resulting tree.

The binary tree of Figure 9-40 is to be used for Exercises 24 to 27.

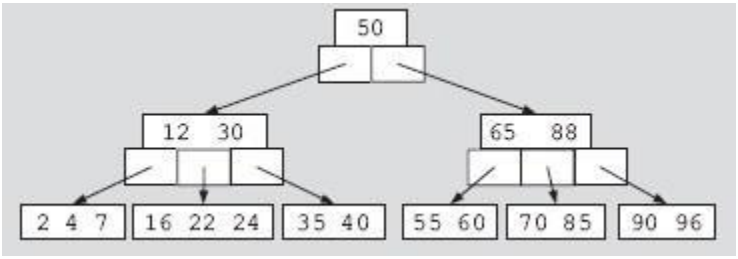


FIGURE 9-40 B-tree of order 5 for Exercises 24 to 27

24. Delete 7 from the B-tree of order 5 of Figure 9-40. Show the resulting tree.

25. Delete 40 from the B-tree of order 5 of Figure 9-40. Show the resulting tree.

26. Delete 88 from the B-tree of order 5 of Figure 9-40. Show the resulting tree.

27. Delete 12 from the B-tree of order 5 of Figure 9-40. Show the resulting tree.

28. Suppose that you have the keys 40, 30, 70, 5, 16, 82, 95, 100, 73, 54, 98, 37, 25, 62, 81, 150, 79, and 87.

a. Insert these keys into an initially empty B-tree of order 5.

b. Insert these keys into an initially empty B-tree of order 6.

10 TOPIC 10:GRAPHS



LEARNING OUTCOMES:

After studying this topic you should be able to:

- Learn about graphs
- Become familiar with the basic terminology of graph theory
- Discover how to represent graphs in computer memory
- Examine and implement various graph traversal algorithms
- Learn how to implement a shortest path algorithm
- Examine and implement the minimum spanning tree algorithm
- Explore topological sort
- Learn how to find Euler circuits in a graph

In previous chapters, you learned various ways to represent and manipulate data. This chapter discusses how to implement and manipulate graphs, which have numerous applications in computer science.

10.1 Introduction

In 1736, the following problem was posed. In the town of Königsberg (now called Kaliningrad), the river Pregel (Pregolya) flows around the island Kneiphof and then divides into two. See Figure 10-1.

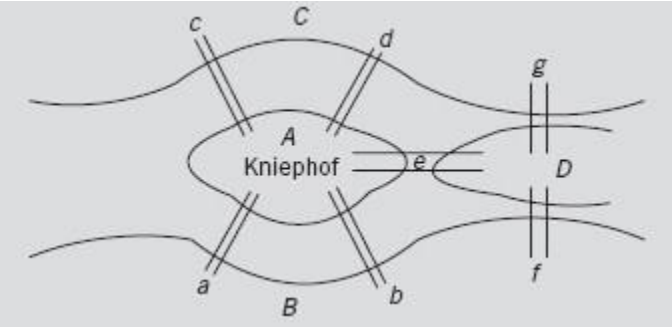


FIGURE 10-1 The Königsberg bridge problem

The river has four land areas (A, B, C, D), as shown in the figure. These land areas are connected using seven bridges, as shown in Figure 10-1. The bridges are labeled a, b, c, d, e, f, and g. The Ko¨nigsberg bridge problem is as follows: Starting at one land area, is it possible to walk across all the bridges exactly once and return to the starting land area? In 1736, Euler represented the Ko¨nigsberg bridge problem as a graph, as shown in Figure 10-2, and answered the question in the negative. This marked (as recorded) the birth of graph theory.

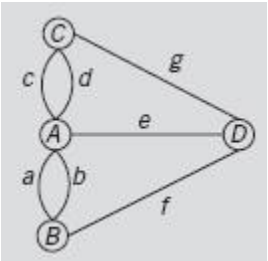


FIGURE 10-2 Graph representation of the Königsberg bridge problem

Over the past 200 years, graph theory has been applied to a variety of applications. Graphs are used to model electrical circuits, chemical compounds, highway maps, and so on. They are also used in the analysis of electrical circuits, finding the shortest route, project planning, linguistics, genetics, social science, and so forth. In this chapter, you learn about graphs and their applications in computer science.

10.2 Graph Definitions and Notations

To facilitate and simplify our discussion, we borrow a few definitions and terminology from set theory. Let X be a set. If a is an element of X, we write a X. (The symbol '' '' means ''belongs to.'') A set Y is called a subset of X if every element of Y is also an element of X. If Y is a subset of X, we write Y X. (The symbol '' '' means ''is a subset of.'') The intersection of sets A and B, written A B, is the set of all elements that are in A and B; that is, A B = {x | x A and x B}. (The symbol '' '' means ''intersection.'') The union of sets A and B, written A B, is the set of all elements that are in A or in B; that is, A B = {x | x A or x B}. (The symbol '' '' means ''union.'') For sets A and B, the set A x B is the set of all ordered pairs of elements of A and B; that is, A x B = {(a, b) | a A, b B}. (The symbol ''x'' means ''Cartesian product.'')

A graph G is a pair, G = (V, E), where V is a finite nonempty set, called the set of vertices of G and E V x V . That is, the elements of E are pairs of elements of V. E is called the set of edges of G. G is called trivial if it has only one vertex.

Let V(G) denote the set of vertices, and E(G) denote the set of edges of a graph G. If the elements of E are ordered pairs, G is called a directed graph or digraph; otherwise, G is called an undirected graph. In an undirected graph, the pairs (u, v) and (v, u) represent the same edge.

Let G be a graph. A graph H is called a subgraph of G if V(H) V(G) and E(H) E(G); that is, every vertex of H is a vertex of G, and every edge in H is an edge in G.

A graph can be shown pictorially. The vertices are drawn as circles, and a label inside the circle represents the vertex. In an undirected graph, the edges are drawn using lines. In a directed graph, the edges are drawn using arrows.

EXAMPLE 10-1

Figure 10-3 shows some examples of undirected graphs.

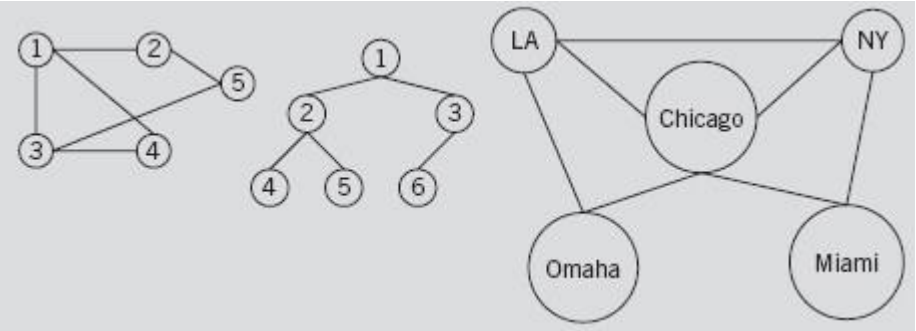


FIGURE 10-3 Various undirected graphs

EXAMPLE 10-2

Figure 10-4 shows some examples of directed graphs.

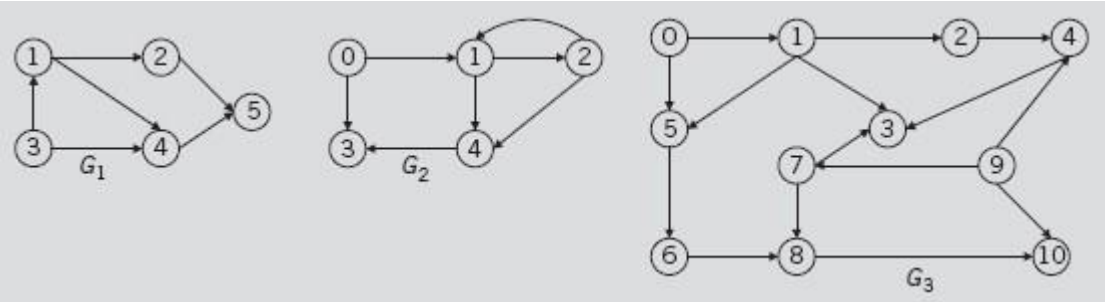


FIGURE 10-4 Various directed graphs

For the graphs of Figure 10-4, we have

$V(G_1) = \{1, 2, 3, 4, 5\}$ $E(G_1) = \{(1, 2), (1, 4), (2, 5), (3, 1), (3, 4), (4, 5)\}$

$V(G_2) = \{0, 1, 2, 3, 4\}$ $E(G_2) = \{(0, 1), (0, 3), (1, 2), (1, 4), (2, 1), (2, 4), (4, 3)\}$

$V(G_3) = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ $E(G_3) = \{(0, 1), (0, 5), (1, 2), (1, 3), (1, 5), (2, 4), (4, 3), (5, 6), (6, 8),$ $(7, 3), (7, 8), (8, 10), (9, 4), (9, 7), (9, 10)\}$

Let G be an undirected graph. Let u and v be two vertices in G. Then u and v are called adjacent if there is an edge from one to the other; that is, (u, v) E. An edge incident on a single vertex is called a loop. If two edges, e_1 and e_2, are associated with the same pair of vertices {u, v}, then e_1 and e_2 are called parallel edges. A graph is called a simple graph if it has no loops and no parallel edges. Let e = (u, v) be an edge in G. We then say that edge e is incident on the vertices u and v. The degree of u, written deg(u) or d(u), is the number of edges incident with u. We make the convention that each loop on a vertex u contributes 2 to the degree of u. u is called an even (odd) degree vertex if the degree of u is even (odd). There is a path from u to v if there is a sequence of vertices $u_1, u_2, \ldots, u_n$ such that $u = u_1$, $u_n = v$ and (u_i, u_{i+1}) is an edge for all i = 1, 2, . . ., n - 1. Vertices u and v are called connected if there is a path from u to v. A simple path is a path in which all the vertices, except possibly the first and last vertices, are distinct. A cycle in G is a simple path in which the first and last vertices are the same. G is called connected if there is a path from any vertex to any other vertex. A maximal subset of connected vertices is called a component of G.

Let G be a directed graph, and let u and v be two vertices in G. If there is an edge from u to v, that is, (u, v) E, we say that u is adjacent to v and v is adjacent from u. The definitions of the paths and cycles in G are similar to those for undirected graphs. G is called strongly connected if any two vertices in G are connected.

Consider the directed graphs of Figure 10-4. In G_1, 1-4-5 is a path from vertex 1 to vertex 5. There are no cycles in G_1. In G_2, 1-2-1 is a cycle. In G_3, 0-1-2-4-3 is a path from vertex 0 to vertex 3; 1-5-6-8-10 is a path from vertex 1 to vertex 10. There are no cycles in G_3.

10.1 Graph Representation

To write programs that process and manipulate graphs, the graphs must be stored—that is, represented—in computer memory. A graph can be represented (in computer memory) in several ways. We now discuss two commonly used ways: adjacency matrices and adjacency lists.

10.1.1 Adjacency Matrices

Let G be a graph with n vertices, where n > 0. Let V(G) = {v_1, v_2, ..., v_n}. The adjacency matrix A_G is a twodimensional n x n matrix such that the (i, j)th entry of A_G is 1 if there is an edge from v_i to v_j; otherwise, the (i, j)th entry is 0. That is,

$$A_G(i,j) = \begin{cases} 1 & \text{if}(v_i, v_j) \in E(G) \\ 0 & \text{otherwise} \end{cases}$$

In an undirected graph, if (v_i, v_j) E(G), then (v_j, v_i) E(G), so A_G(i, j) = 1 = A_G(j, i). It follows that the adjacency matrix of an undirected graph is symmetric.

EXAMPLE 10-3

Consider the directed graphs of Figure 10-4. The adjacency matrices of the directed graphs G_1 and G_2 are as follows:

$$A_{G_1} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad A_{G_2} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

10.1.2 Adjacency Lists

Let G be a graph with n vertices, where n>0. Let V(G) = {v_1, v_2, ..., v_n}. In the adjacency list representation, corresponding to each vertex, v, there is a linked list such that each node of the linked list contains the vertex, u, such that (v, u) E(G). Because there are n nodes, we use an array, A, of size n, such that A[i] is a reference variable pointing to the first node of the linked list containing the vertices to which vi is adjacent. Clearly, each node has two components, say vertex and link. The component vertex contains the index of the vertex adjacent to vertex i.

EXAMPLE 10-4

Consider the directed graphs of Figure 10-4. Figure 10-5 shows the adjacency list of the directed graph G_2.

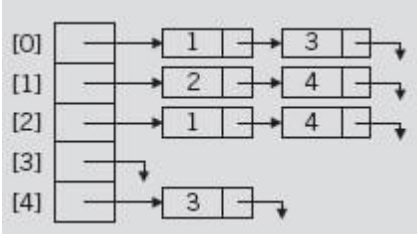


FIGURE 10-5 Adjacency list of graph G_2 of Figure 10-4

Figure 10-6 shows the adjacency list of the directed graph G_3.

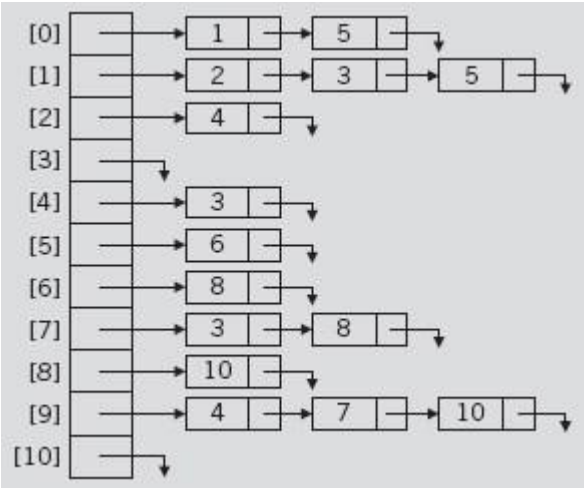


FIGURE 10-6 Adjacency list of graph G_3 of Figure 10-4

10.3 Operations on Graphs

Now that you know how to represent graphs in computer memory, the next obvious step is to learn the basic operations on a graph. The operations commonly performed on a graph are as follows:

1. Create the graph. That is, store the graph in computer memory using a particular graph representation.
2. Clear the graph. This operation makes the graph empty.
3. Determine whether the graph is empty.
4. Traverse the graph.
5. Print the graph.

We will add more operations on a graph when we discuss a specific application or a particular graph later in this chapter.

How a graph is represented in computer memory depends on the specific application. For illustration purposes, we use the adjacency list (linked list) representation of graphs. Therefore, for each vertex v the vertices adjacent to v (in a directed graph, also called the immediate successors) are stored in the linked list associated with v.

To manage the data in a linked list, we use the class unorderedLinkedList, discussed in Chapter 3.

The labeling of the vertices of a graph depends on a specific application. If you are dealing with the graph of cities, you could label the vertices by the names of the cities. However, to write algorithms to manipulate a graph as well as to simplify the algorithm, there must be some ordering to the vertices. That is, we must specify the first vertex, the second vertex, and so on. Therefore, for simplicity, throughout this chapter we assume that the n vertices of the graphs are numbered 0, 1, . . ., n - 1. Moreover, it follows that the class we will design to implement the graph algorithm will not be a template.

10.4 Graphs as ADTs

In this section, we describe the class to implement graphs as an abstract data type (ADT) and provide the definitions of the functions to implement the operations on a graph.

The following class defines a graph as an ADT:

```
//*************************************************************
// class graphType
// This class specifies the basic operations to implement a graph.
//************************************************************* class
graphType
{
        public:

                bool isEmpty() const;
                //Function to determine whether the graph is empty.
                //Postcondition: Returns true if the graph is empty; //
                otherwise, returns false.

                void createGraph(); //Function
                to create a graph.
                //Postcondition: The graph is created using the //
                adjacency list representation.

                void clearGraph(); //Function
                to clear graph.
                //Postcondition: The memory occupied by each vertex //
                is deallocated.

                void printGraph() const;
                //Function to print graph.
                //Postcondition: The graph is printed.

                void depthFirstTraversal();
                //Function to perform the depth first traversal of //the
                entire graph.
                //Postcondition: The vertices of the graph are printed //
                using the depth first traversal algorithm.
```

```
void dftAtVertex(int vertex);
//Function to perform the depth first traversal of
//the graph at a node specified by the parameter vertex.
//Postcondition: Starting at vertex, the vertices are //
printed using the depth first traversal algorithm.

void breadthFirstTraversal();
//Function to perform the breadth first traversal of //the
entire graph.
//Postcondition: The vertices of the graph are printed //
using the breadth first traversal algorithm.

graphType(int size = 0);
//Constructor
```

```
        //Postcondition: gSize = 0; maxSize = size; //
        graph is an array of pointers to linked lists.

        ~graphType();
        //Destructor
        //The storage occupied by the vertices is deallocated.

    protected:
        int maxSize; //maximum number of vertices int
        gSize; //current number of vertices
        unorderedLinkedList<int> *graph; //array to create //adjacency
        lists

    private: void dft(int v, bool
        visited[]);
        //Function to perform the depth first traversal of
        //the graph at a node specified by the parameter vertex. //This
        function is used by the public member functions
        //depthFirstTraversal and dftAtVertex.
        //Postcondition: Starting at vertex, the vertices are //
        printed using the depth first traversal algorithm.
};
```

We leave the UML class diagram of the class graphType as an exercise.

The definitions of the functions of the class graphType are discussed next.

A graph is empty if the number of vertices is 0—that is, if gSize is 0. Therefore, the definition of the function isEmpty is as follows:

```
bool graphType::isEmpty() const
{ return (gSize == 0);
}
```

The definition of the function createGraph depends on how the data is input into the program. For

illustration purposes, we assume that the data to the program is input from a file. The user is prompted for the input file. The data in the file appears in the following form:

```
5
0 2 4 ... -999 1
3 6 8 ... -999
…
```

The first line of input specifies the number of vertices in the graph. The first entry in the remaining lines specifies the vertex, and all of the remaining entries in the line (except the last) specify the vertices that are adjacent to the vertex. Each line ends with the number –999.

Using these conventions, the definition of the function createGraph is as follows:

```
void graphType::createGraph()
{
        ifstream infile;
        char fileName[50];

        int vertex;
        int adjacentVertex;

        if (gSize != 0)                         //if the graph is not empty, make it empty
                clearGraph();

        cout << "Enter input file name: ";
        cin >> fileName; cout << endl;
        infile.open(fileName);
        if (!infile)
        {
                cout << "Cannot open input file." << endl; return;
        }
        infile >> gSize;                        //get the number of vertices
        for (int index = 0; index < gSize; index++)
        { infile >> vertex;
                infile >> adjacentVertex;

                while (adjacentVertex != -999)
                {
                        graph[vertex].insertLast(adjacentVertex);  infile
                        >> adjacentVertex;
                }                               //end while
        }                                       // end for

        infile.close();
}                                               //end createGraph
```

The function clearGraph empties the graph by deallocating the storage occupied by each linked list and then setting the number of vertices to 0.

```
void graphType::clearGraph()
{ for (int index = 0; index < gSize; index++)
        graph[index].destroyList(); gSize = 0;
}                                                    //end clearGraph
```

The definition of the function printGraph is given next:

```
void graphType::printGraph() const
{ for (int index = 0; index < gSize; index++)
        { cout << index << " ";
                graph[index].print()
                ;
                cout << endl;
        }
        cout << endl;
}                                                       //end printGraph
```

The definitions of the constructor and the destructor are as follows:

```
//Constructor
graphType::graphType(int size)
{
        maxSize = size; gSize
        = 0;
        graph = new unorderedLinkedList<int>[size];
}

//Destructor
graphType::~graphType()
{ clearGraph();
}
```

10.5 Graph Traversals

Processing a graph requires the ability to traverse the graph. This section discusses the graph traversal algorithms.

Traversing a graph is similar to traversing a binary tree, except that traversing a graph is a bit more complicated. Now a binary tree has no cycles and starting at the root node we can traverse the entire tree. On the other hand, a graph might have cycles and we might not be able to traverse the entire graph

from a single vertex (for example, if the graph is not connected). Therefore, we must keep track of the vertices that have been visited. We must also traverse the graph from each vertex (that has not been visited) of the graph. This ensures that the entire graph is traversed.

The two most common graph traversal algorithms are the depth-first traversal and breadth-first traversal, which are described next. For simplicity, we assume that when a vertex is visited, its index is output. Moreover, each vertex is visited only once. We use the bool array visited to keep track of the visited vertices.

10.1.3 Depth-First Traversal

The depth-first traversal is similar to the preorder traversal of a binary tree. The general algorithm is as follows: for each vertex, v, in the graph if v is not visited start the depth first traversal at v

Consider the graph G_3 of Figure 10-4. It is shown here again as Figure 10-7 for easy reference.

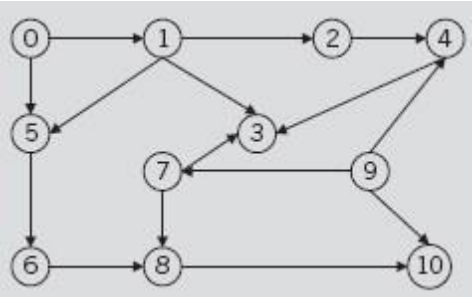


FIGURE 10-7 Directed graph G_3

The depth-first ordering of the vertices of graph G_3 in Figure 10-7 is as follows:

0 1 2 4 3 5 6 8 10 7 9

For the graph of Figure 10-7, the depth-first search starts at the vertex 0. After visiting all the vertices that can be reached starting at the vertex 0, the depth-first search starts at the next vertex that is not visited. There is a path from the vertex 0 to every other vertex except the vertices 7 and 9. Therefore, when the depth-first search starts at the vertex 0, all the vertices except 7 and 9 are visited before these vertices. After completing the depth-first search that started at the vertex 0, the depth-first search starts at the vertex 7 and then at the vertex 9. Note that there is no path from the vertex 7 to the vertex 9. Therefore, after completing the depth-first search that started at the vertex 7, the depth-first search starts at the vertex 9. The general algorithm to do a depth-first traversal at a given node, v, is as follows:

1. mark node v as visited
2. visit the node
3. for each vertex u adjacent to v if u is not visited start the depth first traversal at u

Clearly, this is a recursive algorithm. We use a recursive function, dft, to implement this algorithm. The vertex at which the depth-first traversal is to be started, and the bool array visited, are passed as parameters to this function.

```
void graphType::dft(int v, bool visited[])
{ visited[v] = true;
        cout << " " << v << " ";                          //visit the vertex linkedListIterator<int>
        graphIt;
```

```
                                                    //for each vertex adjacent to v
        for (graphIt = graph[v].begin(); graphIt != graph[v].end();++graphIt)
        {
                int w = *graphIt; if
                (!visited[w]) dft(w,
                visited);
        }                                           //end loop
}                                                   //end dft
```

In the preceding code, note that the statement

```
linkedListIterator<int> graphIt;
```

declares graphIt to be an iterator. In the for loop, we use it to traverse a linked list (adjacency list) to which the pointer graph[v] points. Next, let us look at the statement

```
int w = *graphIt;
```

The expression *graphIt returns the label of the vertex, adjacent to the vertex v, to which graphIt points.

Next, we give the definition of the function depthFirstTraversal to implement the depth-first traversal of the graph.

```
void graphType::depthFirstTraversal()
{
    bool *visited;                                  //pointer to create the array to keep
                                                    //track of the visited vertices
    visited = new bool[gSize];
    for (int index = 0; index < gSize; index++)
        visited[index] = false;
                                                    //For each vertex that is not visited, do a
                                                    //depth first traverssal

    for (int index = 0; index < gSize; index++) if
        (!visited[index]) dft(index,visited);
    delete [] visited;
}                                                   //end depthFirstTraversal
```

The function depthFirstTraversal performs a depth-first traversal of the entire graph. The definition of the function dftAtVertex, which performs a depth-first traversal at a given vertex, is as follows:

```
void graphType::dftAtVertex(int vertex)
{
    bool *visited;
```

```
        visited = new bool[gSize];

        for (int index = 0; index < gSize; index++) visited[index]
                = false;
        dft(vertex, visited);
        delete [] visited;
}                                                    // end dftAtVertex
```

10.1.4 Breadth-First Traversal

The breadth-first traversal of a graph is similar to traversing a binary tree level-by-level (the nodes at each level are visited from left to right). All the nodes at any level, i, are visited before visiting the nodes at level i + 1.

The breadth-first ordering of the vertices of the graph G_3 in Figure 10-7 is as follows:

0 1 5 2 3 6 4 8 10 7 9

For the graph G_3, we start the breadth traversal at vertex 0. After visiting the vertex 0, we visit the vertices that are directly connected to it and are not visited, which are 1 and 5. Next, we visit the vertices that are directly connected to 1 and are not visited, which are 2 and 3. After this, we visit the vertices that are directly connected to 5 and are not visited, which is 6. After this, we visit the vertices that are directly connected to 2 and are not visited, and so on.

As in the case of the depth-first traversal, because it might not be possible to traverse the entire graph from a single vertex, the breadth-first traversal also traverses the graph from each vertex that is not visited. Starting at the first vertex, the graph is traversed as much as possible; we then go to the next vertex that has not been visited. To implement the breadth-first search algorithm, we use a queue. The general algorithm is as follows:

1. for each vertex v in the graph if v is not visited
   add v to the queue //start the breadth first search at v
2. Mark v as visited
3. while the queue is not empty
   3.1. Remove vertex u from the queue
   3.2. Retrieve the vertices adjacent to u 3.3.
   for each vertex w that is adjacent to u
   if w is not visited
   3.3.1 Add w to the queue
   3.3.2 Mark w as visited

The following C++ function, breadthFirstTraversal, implements this algorithm:

```
void graphType::breadthFirstTraversal()
{
    linkedQueueType<int> queue;
```

```
bool *visited;
visited = new bool[gSize];

for (int ind = 0; ind < gSize; ind++)
        visited[ind] = false;                    //initialize the array
                                                 //visited to false

linkedListIterator<int> graphIt;

for (int index = 0; index < gSize; index++)
```

```
            if (!visited[index])
            {
                queue.addQueue(index); visited[index]
                = true;
                cout << " " << index << " ";

                while (!queue.isEmptyQueue())
                {
                    int u = queue.front();
                    queue.deleteQueue();

                    for (graphIt = graph[u].begin(); graphIt != graph[u].end(); ++graphIt)
                    { int w = *graphIt;
                        if (!visited[w])
                        {
                            queue.addQueue(w); visited[w]
                            = true;
                            cout << " " << w << " ";
                        }
                    }
                }                                   //end while
            }
        delete [] visited;
}                                                   //end breadthFirstTraversal
```

As we continue to discuss graph algorithms, we will be writing C++ functions to implement specific algorithms, and so we will derive (using inheritance) new classes from the class graphType.

10.6 Shortest Path Algorithm

Graph theory has many applications. For example, we can use graphs to show how different chemicals are related or to show airline routes. Graphs also be used to show the highway structure of a city, state, or country. The edges connecting two vertices can be assigned a nonnegative real number, called the weight

of the edge. If the graph represents a highway structure, the weight can represent the distance between two places or the travel time from one place to another. Such graphs are called weighted graphs.

Let G be a weighted graph. Let u and v be two vertices in G, and let P be a path in G from u to v. The weight of the path P is the sum of the weights of all the edges on the path P, which is also called the weight of v from u via P.

Let G be a weighted graph representing a highway structure. Suppose that the weight of an edge represents the travel time. For example, to plan monthly business trips, a salesperson wants to find the shortest path (that is, the path with the smallest weight) from her or his city to every other city in the graph. Many such problems exist in which we want to find the shortest path from a given vertex, called the source, to every other vertex in the graph.

This section describes the shortest path algorithm, also called a greedy algorithm, developed by Dijkstra.

Let G be a graph with n vertices, where n ≥ 0. Let V(G) = {v_1, v_2, . . ., v_n}. Let W be a two-dimensional n x n

$$W(i,j) = \begin{cases} w_{ij} & \text{if } (v_i, v_j) \text{ is an edge in } G \text{ and } w_{ij} \text{ is the weight of the edge } (v_i, v_j) \\ \infty & \text{if there is no edge from } v_i \text{ to } v_j \end{cases}$$

matrix such that

The input to the program is the graph and the weight matrix associated with the graph. To make inputting the data easier, we extend the definition of the class graphType (using inheritance), and add the function createWeightedGraph to create the graph and the weight matrix associated with the graph. Let us call this class weightedGraphType. The functions to implement the shortest path algorithm will also be added to this class.

```
//*************************************************************
// class weightedGraphType
// This class specifies the operations to find the weight of the
// shortest path from a given vertex to every other vertex in a graph.
//************************************************************* class
weightedGraphType: public graphType
{
    public:
        void createWeightedGraph();
        //Function to create the graph and the weight matrix.
        //Postcondition: The graph using adjacency lists and
        // its weight matrix is created.

        void shortestPath(int vertex);
        //Function to determine the weight of a shortest path
        //from vertex, that is, source, to every other vertex
        //in the graph.
        //Postcondition: The weight of the shortest path from vertex //
        to every other vertex in the graph is determined.

        void printShortestDistance(int vertex);
        //Function to print the shortest weight from the vertex
        //specified by the parameter vertex to every other vertex in //the
```

graph.
//Postcondition: The weight of the shortest path from vertex //
to every other vertex in the graph is printed.

weightedGraphType(int size = 0);
//Constructor
//Postcondition: gSize = 0; maxSize = size; //
graph is an array of pointers to linked lists.
// weights is a two-dimensional array to store the weights //
of the edges.
// smallestWeight is an array to store the smallest weight //
from source to vertices.

~weightedGraphType();

```
        //Destructor
        //The storage occupied by the vertices and the arrays //weights
        and smallestWeight is deallocated.

    protected:
        double **weights;           //pointer to create weight matrix double
        *smallestWeight;    //pointer to create the array to store
                                    //the smallest weight from source to vertices
};
```

We leave the UML class diagram of the class weightedGraphType and the inheritance hierarchy as an exercise. The definition of the function createWeightedGraph is also left as an exercise for you. Next, we describe the shortest path algorithm.

10.7 Shortest Path

Given a vertex, say vertex (that is, a source), this section describes the shortest path algorithm. The general algorithm is as follows:

1. Initialize the array smallestWeight so that smallestWeight[u] = weights[vertex, u]
2. Set smallestWeight[vertex] = 0.
3. Find the vertex v that is closest to vertex for which the shortest path has not been determined.
4. Mark v as the (next) vertex for which the smallest weight is found.
5. For each vertex w in G, such that the shortest path from vertex to w has not been determined and an edge (v, w) exists, if the weight of the path to w via v is smaller than its current weight, update the weight of w to the weight of v + the weight of the edge (v, w).

Because there are n vertices, Steps 3 through 5 are repeated n - 1 times. Example 10-5 illustrates the shortest path algorithm. (We use the Boolean array weightFound to keep track of the vertices for which the smallest weight from the source vertex has been found. If the smallest weight for a vertex, from the source, has been found, then this vertex's corresponding entry in the array weightFound is set to true; otherwise the corresponding entry is false.)

EXAMPLE 10-5

Let G be the graph shown in Figure 10-8.

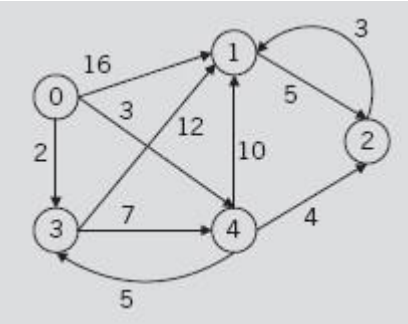


FIGURE 10-8 Weighted graph G

Suppose that the source vertex of G is 0. The graph shows the weight of each edge. After Steps 1 and 2 execute, the resulting graph is as shown in Figure 10-9.

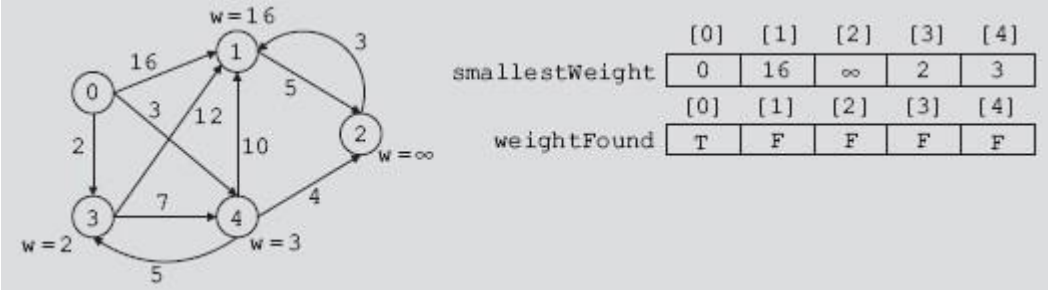


FIGURE 10-9 Graph after Steps 1 and 2 execute

Iteration 1 of Steps 3 through 5: At Step 3, we select a vertex that is closest to the vertex 0 and for which the shortest path has not been found. We do this by finding a vertex in the array smallestWeight that has the smallest weight and its corresponding entry in the array weightFound is false. Therefore, in this iteration, we select the vertex 3. At Step 4, we mark weightFound[3] as true. Next at Step 5, we consider vertices 1 and 4 because these are the vertices for which there is an edge from the vertex 3 and the shortest path from 0 to these vertices has not been found. We then check if the path from the vertex 0 to the vertices 1 and 4 via the vertex 3 can be improved. The weight of the path 0-3-1 from 0 to 1 is less than the weight of the path 0-1. So we update smallestWeight[1] to 14. The weight of the path 0-3-4, which is 2 + 7 = 9, is greater than the weight of the path 0-4, which is 3. So we do not update the weight of the vertex 4. Figure 10-10 shows the resulting graph. (The dotted arrow shows the shortest path from the source—that is, from 0—to the vertex.)

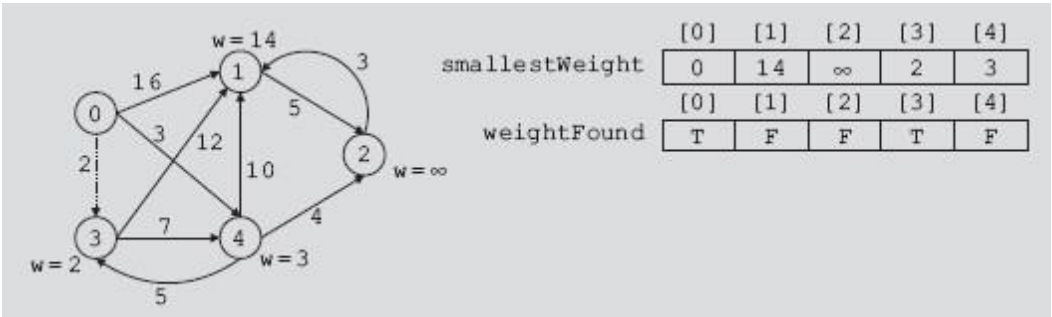


FIGURE 10-10 Graph after the first iteration of Steps 3 to 5

Iteration 2 of Steps 3 through 5: At Step 3, we select vertex 4 because this is the vertex in the array smallestWeight that has the smallest weight and its corresponding entry in the array weightFound is false. Next we execute Steps 4 and 5. At Step 4, we set weightFound[4] to true. At Step 5, we consider vertices 1 and 2 because these are the vertices for which there is an edge from the vertex 4 and the shortest path from 0 to these vertices has not been found.We then check if the path from the vertex 0 to the vertices 1 and 2 via the vertex 4 can be improved. Clearly, the weight of the path 0-4-1, which is 13, is smaller than the current weight of 1, which is 14. So we update smallestWeight[1]. Similarly, we update

smallestWeight[2]. Figure 10-11 shows the resulting graph.

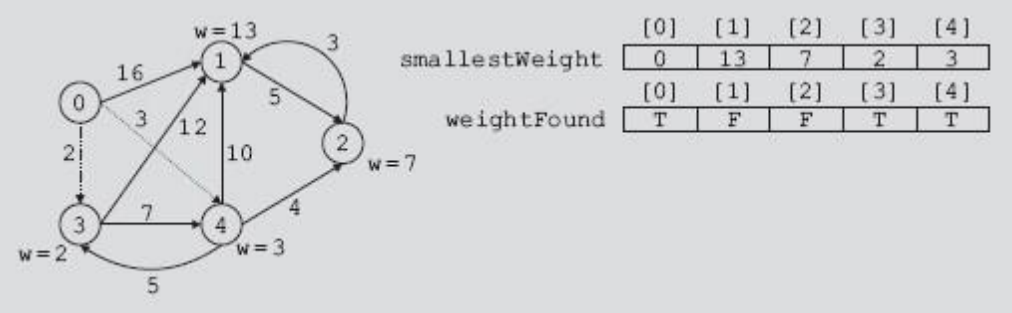


FIGURE 10-11 Graph after the second iteration of Steps 3 to 5

Iteration 3 of Steps 3 through 5: At Step 3, the vertex selected is 2. At Step 4, we set weightFound[2] to true. Next at Step 5, we consider the vertex 1 because this is the vertex for which there is an edge from the vertex 2 and the shortest path from 0 to this vertex has not been found. We then check if the path from the vertex 0 to the vertex 1 via the vertex 2 can be improved. Clearly, the weight of the path 0-4-21, which is 10, from 0 to 1 is smaller than the current weight of 1 (which is 13). So we update smallestWeight[1]. Figure 10-12 shows the resulting graph.

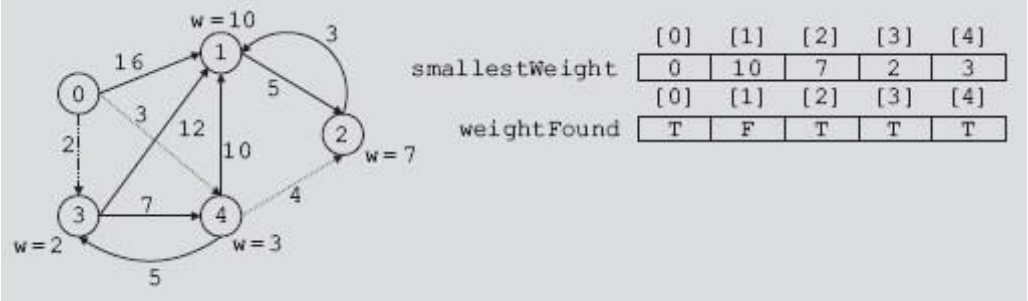


FIGURE 10-12 Graph after the third iteration of Steps 3 to 5

Iteration 4 of Steps 3 through 5: At Step 3, the vertex 1 is selected and at Step 4, weightFound[1] is set to true. In this iteration, the action of Step 5 is null because the shortest path from the vertex 0 to every other vertex in the graph has been determined. Figure 10-13 shows the final graph.

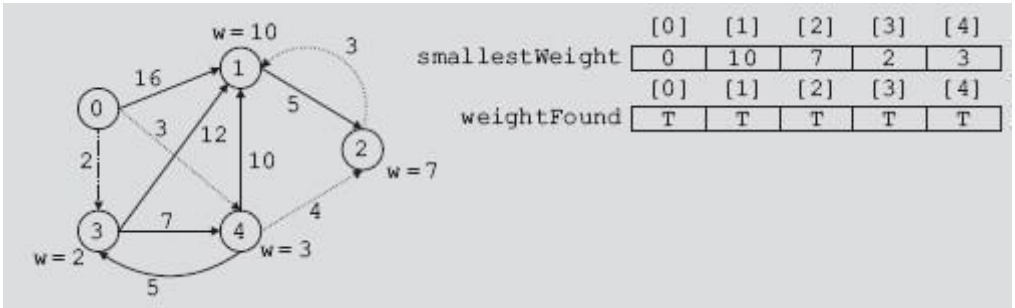


FIGURE 10-13 Graph after the fourth iteration of Steps 3 through 5

The following C++ function, shortestPath, implements the previous algorithm:

```
void weightedGraphType::shortestPath(int vertex)
{
        for (int j = 0; j < gSize; j++) smallestWeight[j] =
                weights[vertex][j];
```

```
    bool *weightFound; weightFound
    = new bool[gSize]; for (int j = 0; j
    < gSize; j++)
        weightFound[j] = false;

    weightFound[vertex] = true;
    smallestWeight[vertex] = 0;

    for (int i = 0; i < gSize - 1; i++)
    {
        double minWeight = DBL_MAX; int
        v;

        for (int j = 0; j < gSize; j++) if (!weightFound[j]) if
            (smallestWeight[j] < minWeight)
                {
                    v = j;
                    minWeight = smallestWeight[v];
                }
            weightFound[v] = true;
            for (int j = 0; j < gSize; j++)
                if (!weightFound[j])
                    if (minWeight + weights[v][j] < smallestWeight[j]) smallestWeight[j]
                        = minWeight + weights[v][j];
    }                                   //end for
}                                       //end shortestPath
```

Note that the function shortestPath records only the weight of the shortest path from the source to a vertex. We leave it for you to modify this function so that the shortest path from the source to a vertex is also recorded. Moreover, this function uses the named constant DBL_MAX, which is defined in the header file cfloat.

Let G be a graph with n vertices. In the function shortestPath, the first for loop executes n times and the

second for loop also executes n times. The third for loop executes n – 1 times. The body of the third for loop contains two for loops, in sequence, and each of these for loops executes n times. Thus, the total number of iterations of the for loops is n + n + (n – 1)(n + n) = 2n + 2n(n – 1) = $O(n^2)$. Hence, the function shortestPath, that is, the shortest path algorithm is of order $O(n^2)$.

The definitions of the function printShortestDistance and the constructor and destructor are as follows:

```
void weightedGraphType::printShortestDistance(int vertex)
{
        cout << "Source Vertex: " << vertex << endl;
        cout << "Shortest distance from source to each vertex."<< endl; cout
        << "Vertex Shortest_Distance" << endl;
 for (int j = 0; j < gSize; j++) cout << setw(4) << j << setw(12) <<
        smallestWeight[j]<< endl; cout << endl;
} //end printShortestDistance
```

```
//Constructor
weightedGraphType::weightedGraphType(int  size):graphType(size)
{
        weights = new double*[size];
        for (int i = 0; i < size; i++) weights[i] =
                new double[size];
        smallestWeight = new double[size];
}
//Destructor
weightedGraphType::~weightedGraphType()
{ for (int i = 0; i < gSize; i++) delete
        [] weights[i]; delete []
        weights;
        delete smallestWeight;
}
```

10.8 Minimum Spanning Tree

Consider the graph of Figure 10-14, which represents the airline connections of a company, between seven cities. The number on each edge represents some cost factor of maintaining the connection between cities.

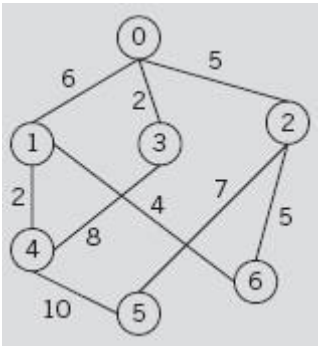


FIGURE 10-14 Airline connections between cities and the cost factor of maintaining the connections

Due to financial hardship, the company needs to shut down the maximum number of connections and still be able to fly (may be not directly) from one city to another. The graphs of Figure 10-15(a), (b), and (c) show three different solutions.

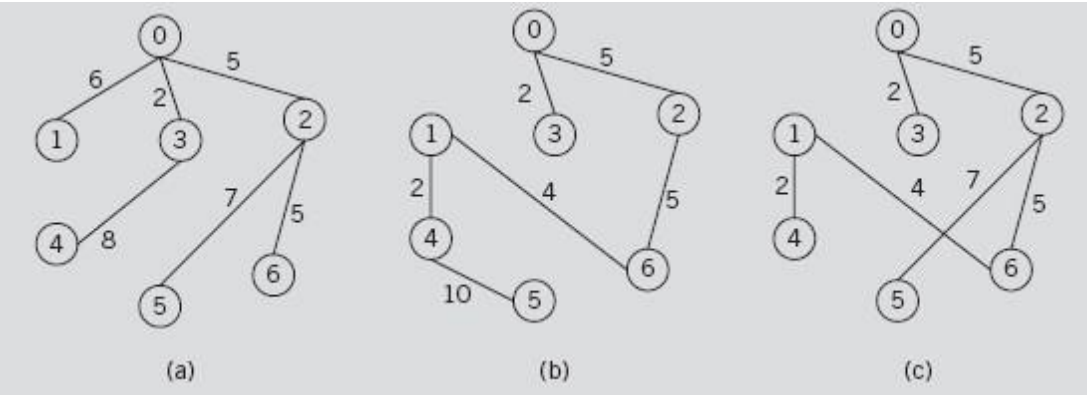


FIGURE 10-15 Possible solutions to the graph of Figure 10-14

The total cost factor of maintaining the remaining connections in Figure 10-15(a) is 33, in Figure 1015(b) it is 28, and in Figure 10-15(c) it is 25. Out of these three solutions, obviously, the desired solution is the one shown by the graph of Figure 10-15(c) because it gives the lowest cost factor. Graphs of Figure 10-15 are called spanning trees of the graph of Figure 10-14.

Let us note the following from the graphs of Figure 10-15. Each of the graphs of Figure 10-15 is a subgraph of the graph of Figure 10-14, and there is a unique path from a node to any other node. Such graphs are called trees. There are many other situations, where given a weighted graph, we need to determine a graph such as in Figure 10-15 with the smallest weight. In this section, we give an algorithm to determine such graphs. However, first we introduce some terminology.

A (free) tree T is a simple graph such that if u and v are two vertices in T, there is a unique path from u to v. A tree in which a particular vertex is designated as a root is called a rooted tree. If a weight is assigned to the edges in T, T is called a weighted tree. If T is a weighted tree, the weight of T, denoted by W(T), is the sum of the weights of all the edges in T.

A tree T is called a spanning tree of graph G if T is a subgraph of G such that V(T) = V(G), that is, all the vertices of G are in T.

Suppose G denotes the graph of Figure 10-14. Then the graphs of Figure 10-15 show three spanning trees of G. Let us note the following theorem.

Theorem 10-1: A graph G has a spanning tree if and only if G is connected.

From this theorem, it follows that to determine a spanning tree of a graph, the graph must be connected.

Definition: Let G be a weighted graph. A minimum (minimal) spanning tree of G is a spanning tree with the minimum weight.

There are two well-known algorithms, Prim's algorithm and Kruskal's algorithm, for finding a minimum spanning tree of a graph. This section discusses Prim's algorithm to find a minimum spanning tree.

Prim's algorithm builds the tree iteratively by adding edges until a minimum spanning tree is obtained. We start with a designated vertex, which we call the source vertex. At each iteration, a new edge that does not form a cycle is added to the tree.

Let G be a weighted graph such that V(G) = {v_0, v_1,. . .,v_{n-1}}, where n, the number of vertices, is nonnegative. Let v0 be the source vertex. Let T be the partially built tree. Initially V(T) contains the source vertex and E(T) is empty. At the next iteration, a new vertex that is not in V(T) is added to V(T), such that an edge exists from a vertex in T to the new vertex so that the corresponding edge has the smallest weight. The corresponding edge is added to E(T).

The general form of Prim's algorithm is as follows. (Let n be the number of vertices in G.)

1. Set V(T) = {source}
2. Set E(T) = empty

3. for i = 1 to n

   3.1. minWeight = infinity;

   3.2. for j = 1 to n

        if v_j is in V(T) for k = 1 to n if v_k is not in T and weight[v_j, v_k]

        < minWeight

        {

        endVertex = vk; edge

        = (v_j, v_k);

        minWeight = weight[v_j, v_k];

        }

   3.3. V(T) = V(T) {endVertex};

   3.4. E(T) = E(T) {edge};

Let us illustrate Prim's algorithm using the graph G of Figure 10-16 (which is same as the graph of Figure 10-14).

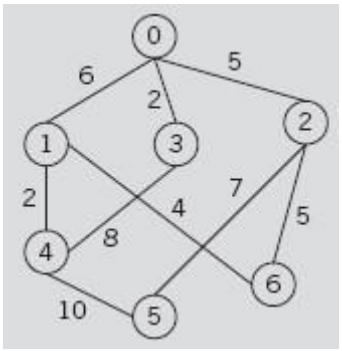


FIGURE 10-16 Weighted graph G

Let N denote the set of vertices of G that are not in T. Suppose that the source vertex is 0. Figure 10-17 shows how Prim's algorithm works.

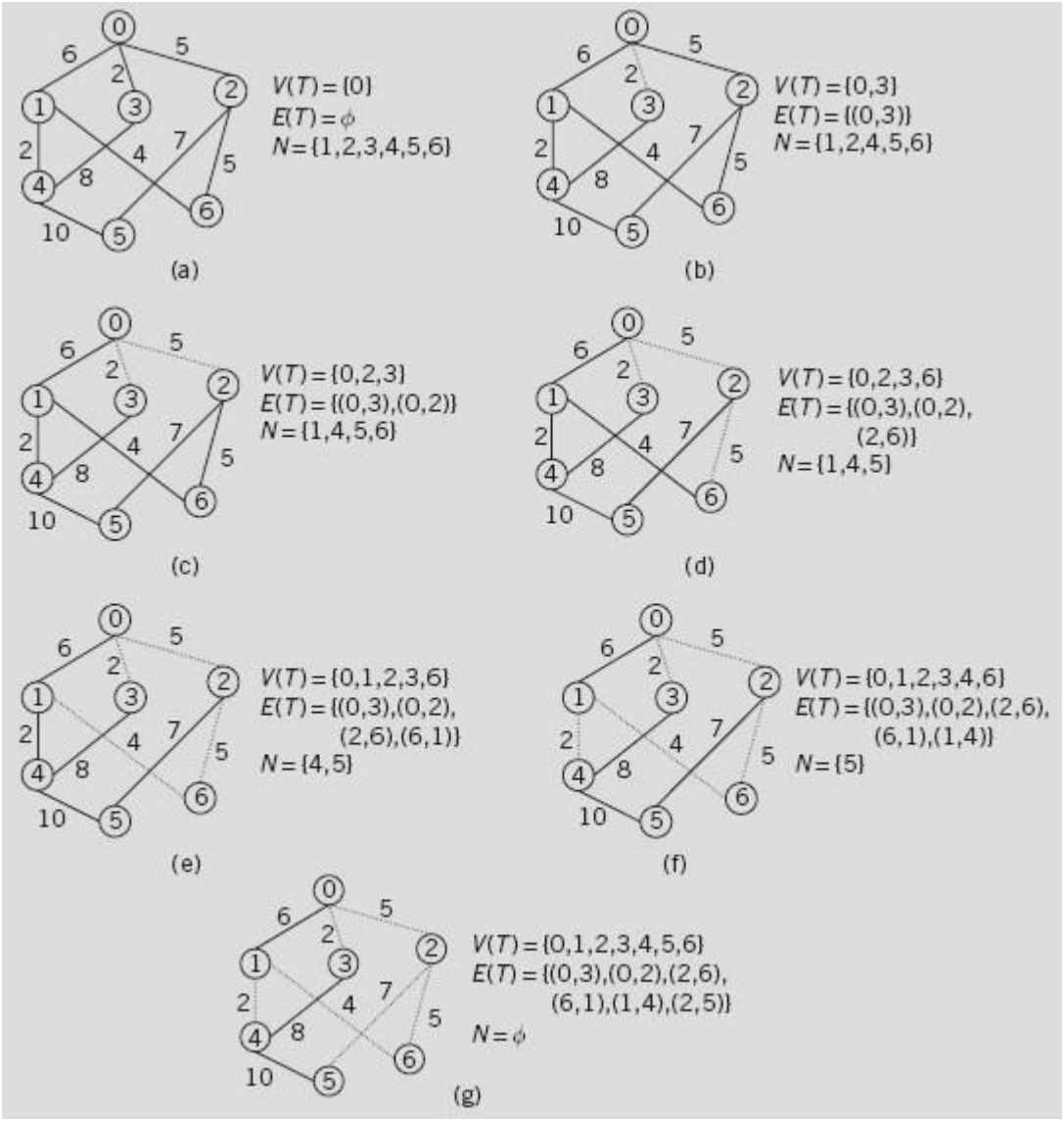


FIGURE 10-17 Graph G, V(T), E(T), and N after Steps 1 and 2 execute

After Steps 1 and 2 execute, V(T), E(T), and N are as shown in Figure 10-17(a). Step 3 checks the following edges: (0,1), (0,2), and (0,3). The weight of the edge (0,1) is 6, the weight of the edge (0,2) is 5, and the weight of the edge (0,3) is 2. Clearly, the edge (0,3) has the smallest weight; see Figure 10-17(b). Therefore, vertex 3 is added to V(T) and the edge (0,3) is added to E(T). Figure 10-17(b) shows the resulting graph, V(T), E(T), and N. (The dotted line shows the edge in T.)

Next, Step 3 checks the following edges: (0,1), (0,2), and (3,4). The weight of the edge (0,1) is 6, the weight of the edge (0,2) is 5, and the weight of the edge (3,4) is 8. Clearly, the edge (0,2) has the smallest weight. Therefore, vertex 2 is added to V(T) and the edge (0,2) is added to E(T). Figure 10-17(c) shows the resulting graph, V(T), E(T), and N.

At the next iteration, Step 3 checks the following edges: (0,1), (2,5), (2,6), and (3,4). The weight of the edge (0,1) is 6, the weight of the edge (2,5) is 7, the weight of the edge (2,6) is 5, and the weight of the

edge (3,4) is 8. Clearly, the edge (2,6) has the smallest weight. Therefore, vertex 6 is added to V(T) and the edge (2,6) is added to E(T). Figure 10-17(d) shows the resulting graph, V(T), E(T), and N. (The dotted lines show the edges in T.)

At the next iteration, Step 3 checks the following edges: (0,1), (2,5), (3,4), and (6,1). The weight of the edge (0,1) is 6, the weight of the edge (2,5) is 7, the weight of the edge (3,4) is 8, and the weight of the

edge (6,1) is 4. Clearly, the edge (6,1) has the smallest weight. Therefore, vertex 1 is added to V(T) and the edge (6,1) is added to E(T). Figure 10-17(e) shows the resulting graph, V(T), E(T), and N. (The dotted lines show the edges in T.)

At the next iteration, Step 3 checks the following edges: (1,4), (2,5), and (3,4). The weight of the edge (1,4) is 2, the weight of the edge (2,5) is 7, and the weight of the edge (3,4) is 8. Clearly, the edge (1,4) has the smallest weight. Therefore, vertex 4 is added to V(T) and the edge (1,4) is added to E(T). Figure 10-17(f) shows the resulting graph, V(T), E(T), and N. (The dotted lines show the edges in T.)

At the next iteration, Step 3 checks the following edges: (2,5) and (4,5). The weight of the edge (2,5) is 7 and the weight of the edge (4,5) is 10. Clearly, the edge (2,5) has the smallest weight. Therefore, vertex 5 is added to V(T) and the edge (2,5) is added to E(T). Figure 10-17(g) shows the resulting graph, V(T), E(T), and N. (The dotted lines show the edges in T.)

In Figure 10-17(g), the dotted lines show a minimum spanning tree of G of weight 25.

Before we give the definition of the function to implement Prim's algorithm, let us first define a spanning tree as an ADT.

Let mstv be a bool array such that mstv[j] is true if the vertex v_i is in T, and false otherwise. Let edges be an array such that edges[j] = k, if there is an edge connecting vertices v_j and v_k. Suppose that the edge (v_i, v_j) is in the minimum spanning tree. Let edgeWeights be an array such that edgeWeights[j] is the weight of the edge (v_i, v_j).
Using these conventions, the following class defines a spanning tree as an ADT:

```
//*************************************************************
// class msTreeType
// This class specifies the operations to find a minimum spanning tree in a graph.
//*************************************************************
class msTreeType: public graphType
{
        public:
                void createSpanningGraph();
                //Function to create the graph and the weight matrix.
```

```
//Postcondition: The graph using adjacency lists and
// its weight matrix is created.

void minimumSpanning(int sVertex);
//Function to create a minimum spanning tree with //root
as sVertex.
// Postcondition: A minimum spanning tree is created.
// The weight of the edges is also saved in the array //
edgeWeights.

void printTreeAndWeight();
//Function to output the edges of the minimum spanning tree
//and the weight of the minimum spanning tree.
//Postcondition: The edges of a minimum spanning tree //
and their weights are printed.
```

```
        msTreeType(int size = 0);
        //Constructor
        //Postcondition: gSize = 0; maxSize = size; //
        graph is an array of pointers to linked lists.
        // weights is a two-dimensional array to store the weights //
        of the edges.
        // edges is an array to store the edges of a minimum //
        spanning tree.
        // edgeWeights is an array to store the weights of the //
        edges of a minimum spanning tree.

        ~msTreeType();
        //Destructor
        //The storage occupied by the vertices and the arrays //weights,
        edges, and edgeWeights is deallocated.

    protected:
        int source; double
        **weights; int
        *edges;
        double *edgeWeights;
};
```

We leave the UML class diagram of the class msTreeType and the inheritance hierarchy as an exercise. The definition of the function createSpanningGraph is also left as an exercise for you. This function creates the graph and the weight matrix associated with the graph.

The following C++ function, minimumSpanning, implements Prim's algorithm, as described previously:

```
void msTreeType::minimumSpanning(int sVertex)
{
    int startVertex, endVertex;
    double minWeight;
    source = sVertex;
```

```
bool *mstv;
mstv = new bool[gSize];

for (int j = 0; j < gSize; j++)
{ mstv[j] = false; edges[j]
        = source;
        edgeWeights[j] = weights[source][j];
}

mstv[source] = true;
edgeWeights[source] = 0;
```

```
    for (int i = 0; i < gSize - 1; i++)
    {
        minWeight = DBL_MAX;

        for (int j = 0; j < gSize; j++) if
            (mstv[j])
                for (int k = 0; k < gSize; k++) if (!mstv[k] &&
                    weights[j][k] < minWeight)
                    {
                        endVertex = k; startVertex
                        = j;
                        minWeight = weights[j][k];
                    }

        mstv[endVertex] = true; edges[endVertex]
        = startVertex;
        edgeWeights[endVertex] = minWeight;
    }                                                   //end for
}                                                       //end minimumSpanning
```

The definition of the function minimumSpanning contains three nested for loops. Therefore, in the worst case, Prim's algorithm given in this section is of the order $O(n^3)$. It is possible to design Prim's algorithm so that it is of the order $O(n^2)$; Programming Exercise 5 at the end of this chapter asks you to do this.

The definition of the function printTreeAndWeight is as follows:

```
void msTreeType::printTreeAndWeight()
{
    double treeWeight = 0;
    cout << "Source Vertex: " << source << endl; cout
    << "Edges Weight" << endl;

    for (int j = 0; j < gSize; j++)
    {
```

```
            if (edges[j] != j)
            {
                treeWeight = treeWeight + edgeWeights[j];
                cout << "("<<edges[j] << ", " << j << ") " << edgeWeights[j] << endl;
            }
        }

        cout << endl;
        cout << "Minimum spanning Tree Weight: " << treeWeight << endl;
}                                                   //end printTreeAndWeight
```

The definitions of the constructor and the destructor are as follows:

```
msTreeType::msTreeType(int  size):graphType(size)
{
        weights = new double*[size];

        for (int i = 0; i < size; i++) weights[i] =
                new double[size];

        edges = new int[size];

        edgeWeights = new double[size];
}

//Destructor
msTreeType::~msTreeType()
{ for (int i = 0; i < gSize; i++)
                delete [] weights[i];

        delete [] weights; delete
        [] edges;
        delete edgeWeights;
}
```

10.9 Topological Order

In college, before taking a particular course, students, usually, must take all its prerequisite courses, if any. For example, before taking the Programming II course, the student must take the Programming I course. However, certain courses can be taken independent of each other. The courses within a department can be represented as a directed graph. A directed edge from, say vertex u to vertex v means the course represented by the vertex u is a prerequisite of the course represented by the vertex v. It would be helpful for the students to know, before starting a major, the sequence in which they can take the courses so that before taking a course they take all its prerequisite courses and fulfill the graduation requirements on time. In this section, we describe an algorithm that can be used to output the vertices of a directed graph

in such a sequence. Let us first introduce some terminology.

Let G be a directed graph and V(G) = {v_1, v_2, . . ., v_n}, where n ≥ 0. A topological ordering of V(G) is a linear ordering v_{i1}, v_{i2}, . . ., v_{in} of the vertices such that, if v_{ij} is a predecessor of v_{ik}, j ≠ k, 1 ≤ j ≤ n, 1 ≤ k ≤ n, then v_{ij} precedes v_{ik}, that is, j < k in this linear ordering.

In this section, we describe an algorithm, topological order, which outputs the vertices of a directed graph in topological order. We assume that the graph has no cycles. We leave it as an exercise for you to modify the algorithm for the graphs that have cycles.

Because the graph has no cycles, the following is true:

- There exists a vertex v in G such that v has no successor.
- There exists a vertex u in G such that u has no predecessor.

Suppose that the array topologicalOrder (of size n, the number of vertices) is used to store the vertices of G in topological order. Thus, if a vertex, say u, is a successor of the vertex v and topologicalOrder[j] = v and topologicalOrder[k] = u, then j < k.

The topological sort algorithm can be implemented either using the depth-first traversal or the breadthfirst traversal. This section discusses how to implement topological ordering using the breadth-first traversal.

We extend the definition of the class graphType (using inheritance) to implement the breadth-first topological ordering algorithm. Let us call this class topologicalOrderType. Next, we give the definition of the class that includes the functions to implement the topological ordering algorithm.

```
//*************************************************************
// class topologicalOrderType
// This class specifies the operations to find a topological ordering of a graph.
//************************************************************* class
topologicalOrderType: public graphType
{ public:
        void bfTopOrder();
        //Function to perform breadth first topological ordering of //a
        graph.
        //Postcondition: The vertices are output in a breadth first //topological
        order.

        topologicalOrderType(int size = 0);
        //Constructor
        //Postcondition: gSize = 0; maxSize = size;
        // graph is an array of pointers to linked lists.
};
```

Next, we discuss how to implement the function bfTopOrder.

10.10 Breadth-First Topological Ordering

Recall that the breadth-first traversal algorithm is similar to traversing a binary tree level-by-level, and so

the root node (which has no predecessor) is visited first. Therefore, in the breadth-first topological ordering, we first find a vertex that has no predecessor vertex and place it first in the topological ordering. We next find the vertex, say v, all of whose predecessors have been placed in the topological ordering and place v next in the topological ordering. To keep track of the number of vertices of a vertex, we use the array predCount. Initially, predCount[j] is the number of predecessors of the vertex vj. The queue used to guide the breadth-first traversal is initialized to those vertices vk such that predCount[k] is 0. In essence, the general algorithm is as follows:

4. Create the array predCount and initialize it so that predCount[i] is the number of predecessors of the vertex v_i.
5. Initialize the queue, say queue, to all those vertices v_k so that predCount[k] is 0. (Clearly, queue is not empty because the graph has no cycles.)
6. while the queue is not empty

   3.1 Remove the front element, u, of the queue.

   3.2 Put u in the next available position, say topologicalOrder[topIndex], and increment topIndex.

3.3 For all the immediate successors w of u,

3.3.1 Decrement the predecessor count of w by 1.

3.3.2 if the predecessor count of w is 0, add w to queue.

The graph G_3 of Figure 10-7 has no cycles. The vertices of G_3 in breadth-first topological ordering are as follows:

Breadth First Topological order: 0 9 1 7 2 5 4 6 3 8 10

Next, we illustrate the breadth-first topological ordering of the graph G_3.

After Steps 1 and 2 execute, the arrays predCount, topologicalOrder, and queue are as shown in Figure 10-18. (Notice that for simplicity, we show only the elements of the queue.)

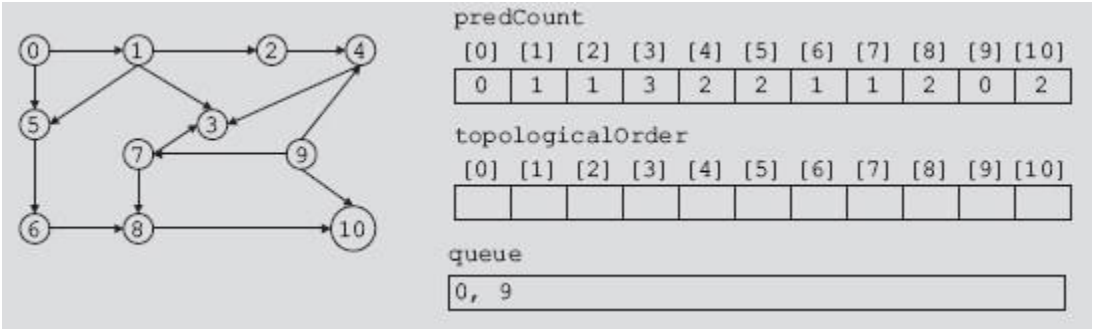


FIGURE 10-18 Arrays predCount, topologicalOrder, and queue after Steps 1 and 2 execute

Step 3 executes as long as the queue is nonempty.

Iteration 1 of Step 3: After Step 3.1 executes, the value of u is 0. Step 3.2 stores the value of u, which is 0, in the next available position in the array topologicalOrder.Notice that 0 is stored at position 0 in this array. Step 3.3 reduces the predecessor count of all the successors of 0 by 1, and if the predecessor count of any successor node of 0 reduces to 0, that node is pushed into queue. The successor nodes of the node 0 are the nodes 1 and 5. The predecessor count of the node 1 reduces to 0, and the predecessor count of the node 5 reduces to 1. The node 1 is pushed into queue. After the first iteration of Step 3, the arrays predCount, topologicalOrder, and queue are as shown in Figure 10-19.

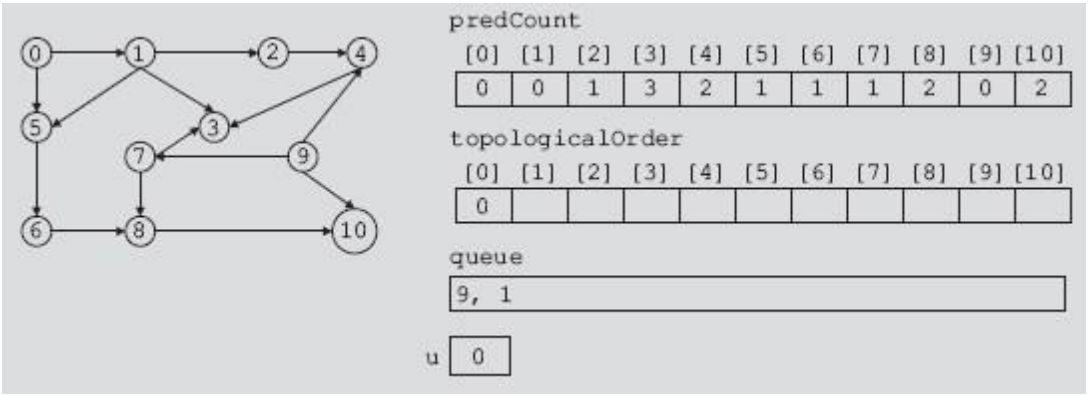
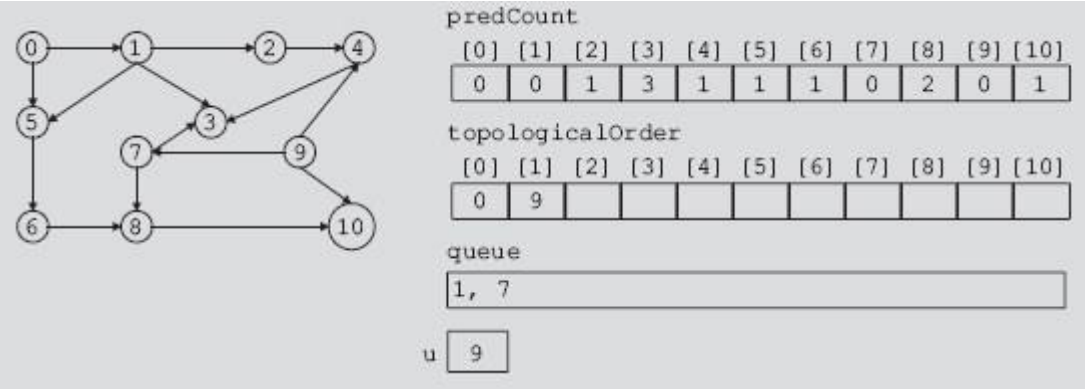


FIGURE 10-19 Arrays predCount, topologicalOrder, and queue after the first iteration of Step 3

Iteration 2 of Step 3: The queue is nonempty. After Step 3.1 executes, the value of u is 9. Step 3.2 stores the value of u, which is 9, in the next available position in the array topologicalOrder. Notice that 9 is stored at position 1 in this array. Step 3.3 reduces the predecessor count of all the successors of 9 by 1, and if the predecessor count of any successor node of 9 reduces to 0, that node is pushed into queue. The

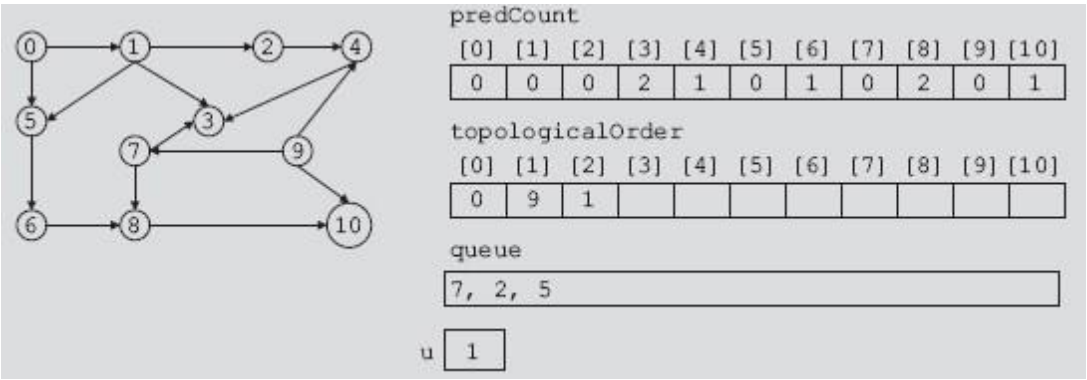
successor nodes of the node 9 are the nodes 4, 7, and 10. The predecessor count of the node 7 reduces to 0 and the predecessor count of the nodes 4 and 10 reduces to 1. The node 7 is pushed into queue. After the second iteration of Step 3, the arrays predCount, topologicalOrder, and queue are as shown in Figure



10-20.

FIGURE 10-20 Arrays predCount, topologicalOrder, and queue after the second iteration of Step 3

Iteration 3 of Step 3: The queue is nonempty. After Step 3.1 executes, the value of u is 1. Step 3.2 stores the value of u, which is 1, in the next available position in the array topologicalOrder. Notice that 1 is stored at position 2 in this array. Step 3.3 reduces the predecessor count of all the successors of 1 by 1 and if the predecessor count of any successor node of 1 reduces to 0, that node is pushed into queue. The successor nodes of the node 1 are the nodes 2, 3, and 5. The predecessor count of the nodes 2 and 5



reduces to 0 and the predecessor count of the node 3 reduces to 2. The nodes 2 and 5, in this order, are pushed into the queue. After the third iteration of Step 3, the arrays predCount, topologicalOrder, and queue are as shown in Figure 10-21.

FIGURE 10-21 Arrays predCount, topologicalOrder, andqueue after the third iteration of Step 3

If you repeat Step 3 eight more times, the arrays predCount, topologicalOrder, and queue are as shown in Figure 10-22.

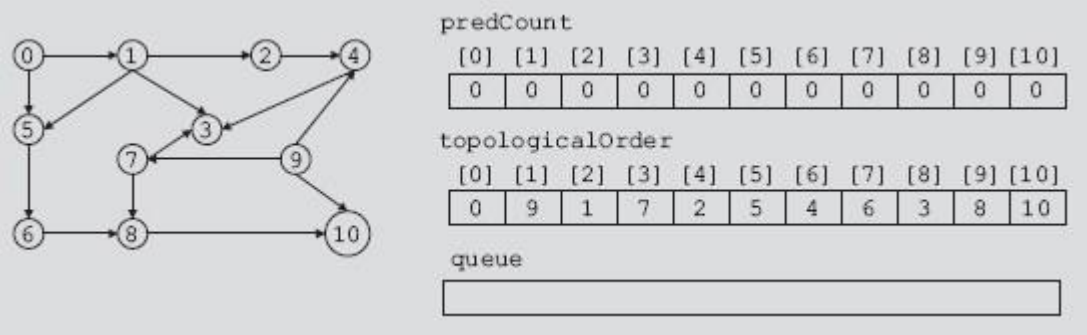


FIGURE 10-22 Arrays predCount, topologicalOrder, and queue after Step 3 executes

In Figure 10-22, the array topologicalOrder shows the breadth-first topological ordering of the nodes of the graph G_3.

The following C++ function implements this breadth-first topological ordering algorithm:

```
void topologicalOrderType::bfTopOrder()
{
    linkedQueueType<int> queue;

    int *topologicalOrder;                  //pointer to the array to store
                                            //breadth first topological ordering
    topologicalOrder = new int[gSize];
    int topIndex = 0;
    linkedListIterator<int> graphIt;        //iterator to traverse a linked list

    int *predCount;                         //pointer to the array to store the
                                            //predecessor count of a vertex.
    predCount = new int[gSize];

    for (int ind = 0; ind < gSize; ind++) predCount[ind]
        = 0;

                                            //Determine the predecessor count of all the vertices.
    for (int ind = 0; ind < gSize; ind++)
    {
        for (graphIt = graph[ind].begin(); graphIt != graph[ind].end(); ++graphIt)
        {
            int w = *graphIt;
            predCount[w]++;
        }
    }

                                                    (predCount[ind] == 0) queue.addQueue(ind);

    for (int ind = 0; ind < gSize; ind++) if                    while (!queue.isEmptyQueue())
```

```
{                                   //Initialize queue: If the predecessor count of //vertex
                                    is 0, put this node into the queue.
    int u = queue.front(); queue.deleteQueue();
    topologicalOrder[topIndex++] = u;
```

```
                                        //Reduce the predecessor count of all the successors //of
                                        u by 1. If the predecessor count of a vertex //becomes 0,
                                        put this vertex into the queue.
            for (graphIt = graph[u].begin();graphIt != graph[u].end(); ++graphIt)
            {
                int w = *graphIt; predCount[w]--;
                if (predCount[w] == 0) queue.addQueue(w);
            }
        }//end while

                                        //output the vertices in breadth first topological order
        for (int ind = 0; ind < gSize; ind++) cout <<
            topologicalOrder[ind] << " ";

        cout << endl;

        delete [] topologicalOrder;
        delete [] predCount;
}                                       //bfTopOrder
```

We leave the definition of the constructor as an exercise.

10.2 Euler Circuits

Let us consider the Königsberg bridge problem stated at the beginning of the chapter. The problem is to determine whether it is possible to take a walk that crosses each bridge exactly once before returning to the starting point; see Figure 10-1. As remarked earlier, Euler converted this problem into a graph theory problem as follows: Each of the islands A, B, C, and D is considered as a vertex of a graph and the bridges are considered as edges, as shown in Figure 10-2. Now the problem reduces to finding a circuit in the graph of Figure 10-2, such that it contains all the edges. In this section, we further describe properties of graphs, which will help us answer this question.

Definition: A circuit is a path of nonzero length from a vertex u to u with no repeated edges.

Definition: A circuit in a graph that includes all the edges of the graph is called an Euler circuit.

Definition: A graph G is said to be Eulerian if either G is a trivial graph or G has an Euler circuit.

Notice that the graph of Figure 10-2 is a connected graph and this graph has odd degree vertices as well as even degree vertices.

Let us consider a connected graph with more than one vertex such that every vertex has odd degree. For example, consider the graph of Figure 10-23. This is a connected graph and every vertex of this graph is odd degree. This graph has no circuit and so has no circuit that contains all the edges.

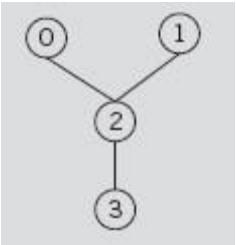


FIGURE 10-23 A graph with all vertices of odd degree

Next consider the connected graph G of Figure 10-24 such that every vertex has even degree.

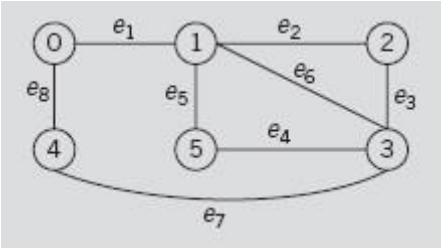


FIGURE 10-24 A graph with all vertices of even degree

The graph of Figure 10-24 has an Euler circuit. For example, (0, e_1, 1, e_2, 2, e_3, 3, e_4, 5, e_5, 1, e_6, 3, e_7, 4, e_8, 0) is an Euler circuit in the graph of Figure 10-24.

The following theorems give necessary and sufficient conditions for a connected graph to have an Euler circuit.

Theorem 10-2: If a connected graph G is Eulerian, then every vertex of G has even degree.

Theorem 10-3: Let G be a connected graph such that every vertex of G is of even degree. Then G has an Euler circuit.

We can effectively use this theorem to determine whether a connected graph G has an Euler circuit by checking whether all of its vertices are of even degree.

Let us again consider the Ko¨nigsberg bridge problem. Notice that the graph corresponding to this problem is a connected graph but has vertices of odd degree; see Figure 10-2. Hence, by Theorem 10-2, the graph of Figure 10-2 has no Euler circuit. In other words, starting at one land area, it is not possible to walk across all the bridges exactly once and return to the starting land area.

After 1736, two additional bridges have been constructed on the Pregel river—one is between the regions B and C and the other is between A and D. The graph with the additional two bridges is shown in Figure 10-25.

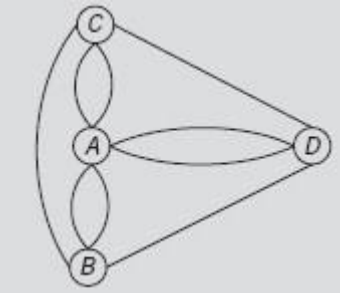


FIGURE 10-25 Graph of the Königsberg bridge problem with two additional bridges

By Theorem 10-3, every connected graph with only even degree vertices has an Euler circuit. Next, we describe an algorithm, known as Fleury's algorithm, which can be used to construct an Euler circuit in a connected graph with vertices of even degrees.

Fleury's Algorithm

Step 1. Choose a vertex v as the starting vertex for the circuit and choose an edge e with v as one of the end vertices.

Step 2. If the other end vertex u of the edge e is also v, go to Step 3. Otherwise, choose an edge e_1 different from e with u as one of the end vertices. If the other vertex u_1 of e_1 is v, go to Step 3; otherwise, choose an edge e_2 different from e and e_1 with u_1 as one of the end vertices and repeat Step
2.

Step 3. If the circuit T_1 obtained in Step 2 contains all the edges, then stop. Otherwise, choose an edge e_j different from the edges of T_1 such that one of the end vertices of e_j, say, w is a member of the circuit T_1.

Step 4. Construct a circuit T_2 with starting vertex w, as in Steps 1 and 2, such that all the edges of T_2 are different from the edges in the circuit T_1.

Step 5. Construct the circuit T_3 by inserting the circuit T_2 at w of the circuit T_1. Now go to Step 3 and repeat Step 3 with the circuit T_3.

Next, we illustrate how Fleury's algorithm works. Consider the graph of Figure 10-26.

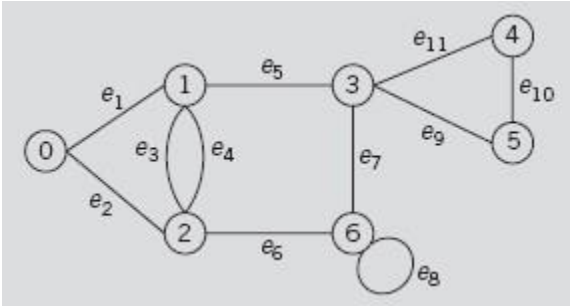


FIGURE 10-26 A graph with all vertices of even degree

Let us apply Fleury's algorithm to find an Eulerian circuit.

First, select vertex 0 and form the circuit: T_1: (0, e_1, 1, e_3, 2, e_2, 0).

Next, select vertex 1 and edge e_4. Construct the circuit: C_1: (1, e_4, 2, e_6, 6, e_7, 3, e_5, 1). Then form the circuit: T_2: (0, e_1, C_1, e_3, 2, e_2, 0).

Circuit T_2 does not contain all the edges of the given graph. Now choose vertex 6 and edge e_8 and form the circuit: C_2: (6, e_8, 6).

Now construct the circuit: T_3: (0, e_1, 1, e_4, 2, e_6, C_2, e_7, 3, e_5, 1, e_3, 2, e_2, 0).

This circuit also does not contain all the edges. Select now vertex 3 and edge e_{11}. Form the circuit: C_3: (3, e_{11}, 4, e_{10}, 5, e_9, 3).

Next, construct the circuit: T_4: (0, e_1,1, e_4, 2, e_6, C_2, e_7, C_3, e_5, 1, e_3, 2, e_2, 0).

Circuit T_4 contains all the vertices and all the edges of the given graph and, hence, it is an Euler circuit. We leave it as an exercise for you to write a program to implement Fleury's algorithm.

Key Terms

1. A graph G is a pair, G = (V, E), where V is a finite nonempty set, called the set of vertices of G and E V x V , called the set of edges.
2. In an undirected graph G = (V, E), the elements of E are unordered pairs.
3. In a directed graph G = (V, E), the elements of E are ordered pairs.
4. Let G be a graph. A graph H is called a subgraph of G if every vertex of H is a vertex of G and every edge in H is an edge in G.
5. Two vertices u and v in an undirected graph are called adjacent if there is an edge from one to the other.
6. An edge incident on a single vertex is called a loop.
7. In an undirected graph, if two edges e_1 and e_2 are associated with the same pair of vertices {u, v}, then e_1 and e_2 are called parallel edges.
8. A graph is called a simple graph if it has no loops and no parallel edges.
9. Let e = (u, v) be an edge in an undirected graph G. The edge e is said to be incident on the vertices u and v.
10. A path from a vertex u to a vertex v is a sequence of vertices u_1, u_2, . . ., u_n such that u = u_1, u_n = v, and (u_i, u_{i+1}) is an edge for all i = 1, 2, . . ., n - 1.

11. The vertices u and v are called connected if there is a path from u to v.
12. A simple path is a path in which all the vertices, except possibly the first and last vertices, are distinct.
13. A cycle in G is a simple path in which the first and last vertices are the same.
14. An undirected graph G is called connected if there is a path from any vertex to any other vertex.
15. A maximal subset of connected vertices is called a component of G.
16. Suppose that u and v are vertices in a directed graph G. If there is an edge from u to v, that is, (u, v) E, we say that u is adjacent to v and v is adjacent from u.
17. A directed graph G is called strongly connected if any two vertices in G are connected.
18. Let G be a graph with n vertices, where n > 0. Let V(G) = {v_1, v_2, . . ., v_n}. The adjacency matrix A_G is a two-dimensional n x n matrix such that the (i, j)th entry of A_G is 1 if there is an edge from v_i to v_j; otherwise, the (i, j)th entry is 0.
19. In an adjacency list representation, corresponding to each vertex v is a linked list such that each node of the linked list contains the vertex u and (v, u) E(G).

20. The depth-first traversal of a graph is similar to the preorder traversal of a binary tree.
21. The breadth-first traversal of a graph is similar to the level-by-level traversal of a binary tree.
22. The shortest path algorithm gives the shortest distance for a given node to every other node in the graph.
23. In a weighted graph, every edge has a nonnegative weight.
24. The weight of the path P is the sum of the weights of all the edges on the path P, which is also called the weight of v from u via P.
25. A (free) tree T is a simple graph such that if u and v are two vertices in T, there is a unique path from u to v.
26. A tree in which a particular vertex is designated as a root is called a rooted tree.
27. Suppose T is a tree. If a weight is assigned to the edges in T, T is called a weighted tree.
28. If T is a weighted tree, the weight of T, denoted by W(T), is the sum of the weights of all the edges in T.
29. A tree T is called a spanning tree of graph G if T is a subgraph of G such that V(T) = V(G)—that is, if all the vertices of G are in T.
30. Let G be a graph and V(G) = $\{v_1, v_2, \ldots, v_n\}$, where $n \geq 0$. A topological ordering of V(G) is a linear ordering $v_{i1}, v_{i2}, \ldots, v_{in}$ of the vertices such that if v_{ij} is a predecessor of v_{ik}, $j \neq k$, $1 \leq j, k \leq n$, then v_{ij} precedes v_{ik}, that is, $j < k$ in this linear ordering.
31. A circuit is a path of nonzero length from a vertex u to u with no repeated edges.
32. A circuit in a graph that includes all the edges of the graph is called an Euler circuit.
33. A graph G is said to be Eulerian if either G is a trivial graph or G has an Euler circuit.

Exercises

Use the graph in Figure 10-27 for Exercises 1 through 4.

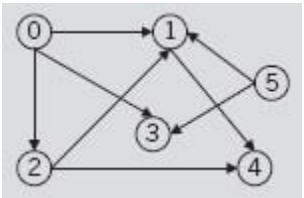


FIGURE 10-27 Graph for Exercises 1 through 4

1. Find the adjacency matrix of the graph.

2. Draw the adjacency list of the graph.

3. List the nodes of the graph in a depth-first traversal.

4. List the nodes of the graph in a breadth-first traversal.

5. Find the weight matrix of the graph in Figure 10-28.

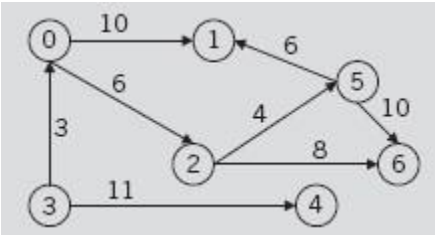


FIGURE 10-28 Graph for Exercise 5

6. Consider the graph in Figure 10-29. Find the shortest distance from node 0 to every other node in the graph.

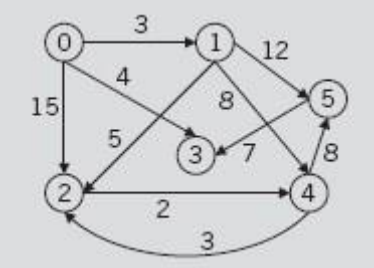


FIGURE 10-29 Graph for Exercise 6

7. Find a spanning tree in the graph in Figure 10-30.

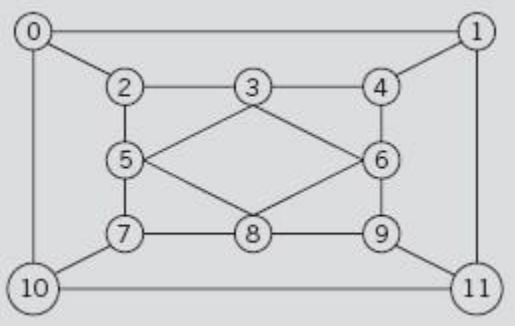


FIGURE 10-30 Graph for Exercise 7

8. Find a spanning tree in the graph in Figure 10-31.

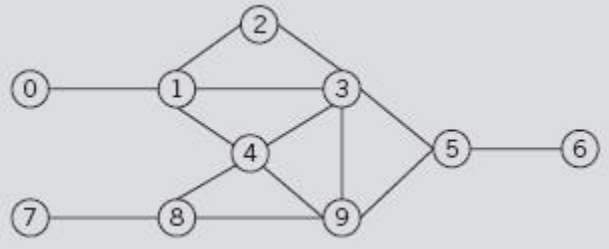


FIGURE 10-31 Graph for Exercise 8

9. Find the minimum spanning tree for the graph in Figure 10-32 using the algorithm given in this chapter.

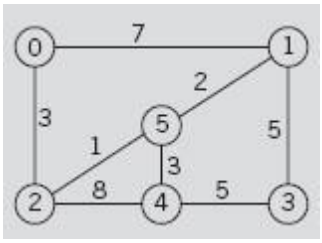


FIGURE 10-32 Graph for Exercise 9

10. List the nodes of the graph of Figure 10-33 in a breadth-first topological ordering.

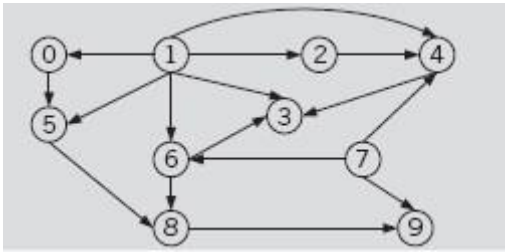


FIGURE 10-33 Graph for Exercise 10

11. Describe whether the graph in Figure 10-34 has an Euler circuit. If the graph has an Euler circuit, find one such circuit.

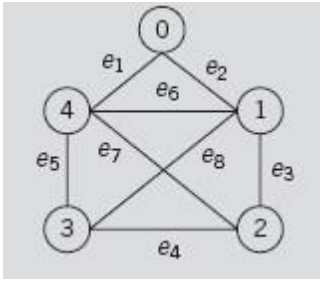


FIGURE 10-34 Graph for Exercise 11

12. Describe whether the graph in Figure 10-35 has an Euler circuit. If the graph has an Euler circuit, find one such circuit.

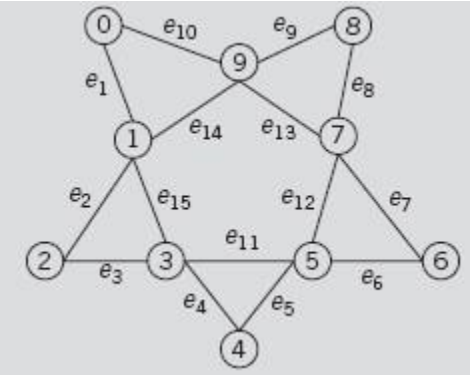


FIGURE 10-35 Graph for Exercise 12

11 ASSIGNMENT.GUIDELINES. ATTENTION STUDENTS/STAFF

If you COPY AND OR REPRODUCE SOMEBODY ELSE'S WORK WITHOUT REFERENCING, your assignment will be penalized by 30%.

11.1.1.1 TO AVOID BEING PENALISED IN YOUR ASSIGNMENTS DO NOT

- **Copy and paste information from the internet and on-line media, such as encyclopedias or journal articles without acknowledging the source of information (Referencing).**
- **Transcribe information from any textbook, encyclopedia, newspaper, journal articles, without acknowledging the source of information (Referencing)**
- **Modify information without acknowledging the source of information (Referencing)**
- **Use photographs, videos, or audio without permission or acknowledgement**
- **Get other people to write your assignment**
- **Buy assignments and submit them as your own effort**
- **Use previous work for a new assignment without citing the original assignment in your reference list.**

Types of Student Plagiarism

Academic Integrity

Good academic practice is about adopting strategies and behaviour that allow students to complete their studies independently and honestly, and writing assignments in an appropriate academic style. The task submitted is also awarded better marks! Bad academic practice includes dishonesty, cheating and plagiarism and also work that is badly prepared and

rushed.

11.1.2 Direct copying

This sort of plagiarism is taking the exact words somebody else wrote, (in a website, a book, another student's work, or any other source) and putting that into an assignment, without referencing that somebody else’s work has been cited.

11.1.3 Word-switching

This sort of plagiarism means taking someone else's writing and changing words here and there, or taking little bits of sentences, without acknowledging whose ideas have been adopted.

11.1.4 Concealing sources

Not making it obvious when drawing on somebody else's work will be regarded as plagiarism. This includes:

- Taking somebody else's ideas and putting them into your words without telling us where you got the ideas.
- Using a source several times, but only referencing it once. Using ideas from the same source several times in a piece of work, requires a citation each time the source is used.

11.1.5 Working with other students

It is preferable for students to do their assignments themselves. Hence

- Copying another student's work is plagiarism.
- Submitting all or part of another student's work as your own is plagiarism.
- Sharing written work is plagiarism.
- Paying somebody to do your work for you is plagiarism.
- In an individual assignment, writing the assignment with other people is plagiarism. (Group assignments are different!).
- Asking another student to translate your ideas into English, or getting their help to write your assignment is plagiarism.

Unless you are told to work in a group, you must work alone. If you want to talk to your friends about the work, do it before you start writing. The work you submit must be your own!

When any of the above types are identified, each case will be investigated for:

- Intention of student
- Degree of plagiarism – how much of the assignment has been plagiarized
- Previous offences by the student

References

Malik, D.S., 2018. *C++ Programming: Program Design Including Data Structures*. 8th ed. United Kingdom: Cengage Learning.

Williams, A., 2019. *C++ Concurrency in Action*. Simon and Schuster. ISBN: 1638356351.

Bishop, C.M., 2021. *Pattern Recognition and Machine Learning*. 2nd ed. New York: Springer.

Liang, Y.D., 2020. *Introduction to Java Programming and Data Structures*. 11th ed. Upper Saddle River: Pearson Education.

Laplante, P.A., 2021. *Introduction to Software Engineering*. 6th ed. Boca Raton: CRC Press.

Miller, B., 2020. *Algorithms and Data Structures: A Computer Science Perspective*. 4th ed. New York: Springer.

Wang, Y., 2022. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. 2nd ed. San Francisco: O'Reilly Media.

The IT qualification at Richfield College stands as a beacon of academic innovation and professional readiness. It equips students with the skills and credentials necessary for thriving in the IT industry. By combining foundational knowledge, practical expertise, and global recognition, the program not only prepares students for immediate employment but also sets them on a trajectory for long-term career success.