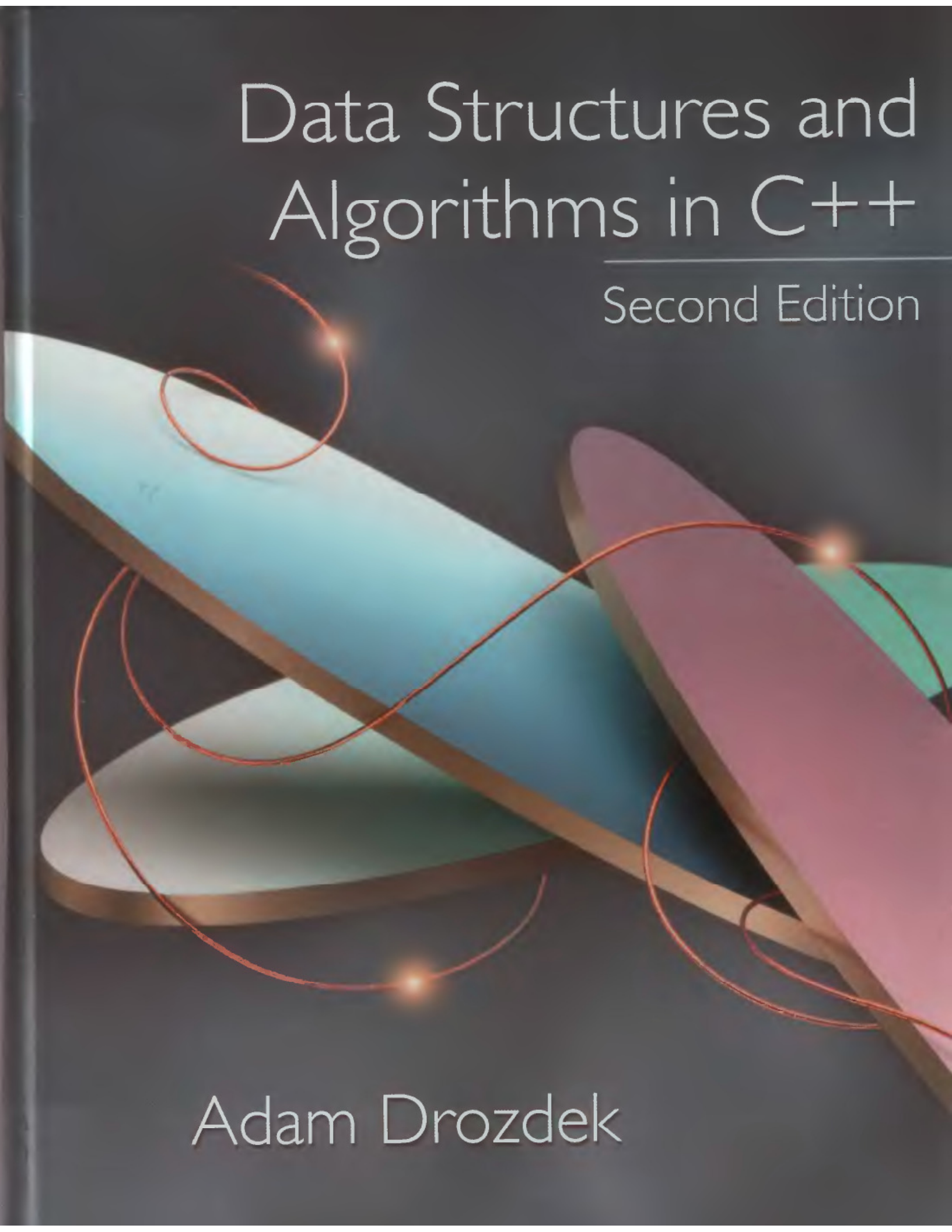
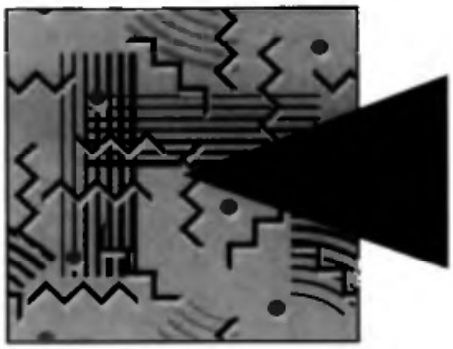
Data Structures and Algorithms in C++

Second Edition



Adam Drozdek



Data Structures and Algorithms in C++

SECOND EDITION

Adam Drozdek

Brooks/Cole
Thomson Learning™

Australia • Canada • Mexico • Singapore • Spain • United Kingdom • United States

Sponsoring Editor: *Kallie Swanson*
Marketing Team: *Chris Kelly, Samantha Cabaluna*
Editorial Assistant: *Grace Fujimoto*
Production Coordinator: *Kelsey McGee*
Production Service: *Forbes Mill Press*
Manuscript Editor: *Frank Hubert*
Permissions Editor: *Mary Kay Hancharick*
Interior Design: *Forbes Mill Press/Robin Gold*
Cover Design: *Roy R. Neuhaus*
Cover Photo: *David Bishop*
Interior Illustration: *Audrey Miller*
Print Buyer: *Vena Dyer*
Typesetting: *Wolf Creek Press & Forbes Mill Press*
Cover Printing, Printing
and Binding: *R.R. Donnelley/Crawfordsville*

For more information about this or any other Brooks/Cole product, contact:
BROOKS/COLE
511 Forest Lodge Road
Pacific Grove, CA 93950 USA
www.brookscole.com
1-800-423-0563 (Thomson Learning Academic Resource Center)

Printed in the United States of America

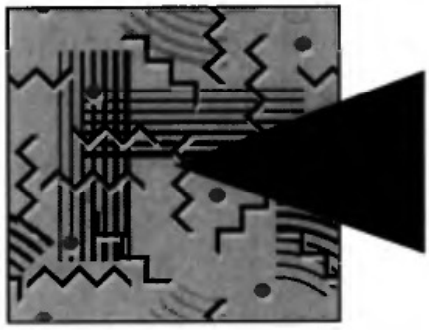
10 9 8 7 6 5 4 3

Library of Congress Cataloging-in-Publication Data

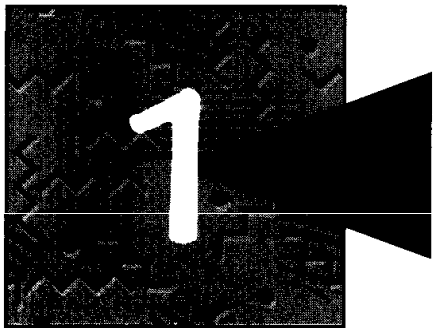
Drozdek, Adam
Data structures and algorithms in C++/Adam Drozdek.--2nd ed.
p. cm.
Includes references (p.) and index.
ISBN 0 534-37597-9 (text)
1. C++ (Computer program language) 2. Data structures (Computer science)
3. Computer algorithms. I. Title.

QA76.73.C153 D76 2000 00-025038
005.13'3--dc21

Contents



Object-Oriented Programming Using C++



1.1 Abstract Data Types

Before a program is written, we should have a fairly good idea how to accomplish the task being implemented by this program. Hence, an outline of the program containing its requirements should precede the coding process. The larger and more complex the project, the more detailed the outline phase should be. The implementation details should be delayed to the later stages of the project. In particular, the details of the particular data structures to be used in the implementation should not be specified at the beginning.

From the start, it is important to specify each task in terms of input and output. At the beginning stages, we should be more concerned with what the program should do, not how it should or could be done. Behavior of the program is more important than the gears of the mechanism accomplishing it. For example, if an item is needed to accomplish some tasks, the item is specified in terms of operations performed on it rather than in terms of its inner structure. These operations may act upon this item, for example, modifying it, searching for some details in it, or storing something in it. After these operations are precisely specified, the implementation of the program may start. The implementation decides which data structure should be used to make execution most efficient in terms of time and space. An item specified in terms of operations is called an *abstract data type*. An abstract data type is not a part of a program, since a program written in a programming language requires the definition of a data structure, not just the operations on the data structure. However, an object-oriented language (OOL) such as C++ has a direct link to abstract data types by implementing them as a class.

1.2 ENCAPSULATION

Object-oriented programming (OOP) revolves around the concept of an object. Objects, however, are created using a class definition. A *class* is a template in accordance to which objects are created. A class is a piece of software that includes data specification and functions operating on these data and possibly on the data belonging to other class instances. Functions defined in a class are called *methods*, *member functions*, or *function members*, and variables used in a class are called *data members* (more properly, they should be called datum members). This combining of the data and related operations is called data *encapsulation*. An *object* is an instance of a class, an entity created using a class definition.

In contradistinction to functions in languages that are not object-oriented, objects make the connection between data and member functions much tighter and more meaningful. In languages that are not object-oriented, declarations of data and definitions of functions can be interspersed through the entire program, and only the program documentation indicates that there is a connection between them. In OOLs, a connection is established right at the outset; in fact, the program is based on this connection. An object is defined by related data and operations, and because there may be many objects used in the same program, the objects communicate by exchanging messages which reveal to each other as little detail about their internal structure as necessary for adequate communication. Structuring programs in terms of objects allows us to accomplish several goals.

First, this strong coupling of data and operations can be used much better in modeling a fragment of the world, which is emphasized especially by software engineering. Not surprisingly, OOP has its roots in simulation, that is, in modeling real-world events. The first OOL was called Simula and it was developed in the 1960s in Norway.

Second, objects allow for easier error finding because operations are localized to the confines of their objects. Even if side effects occur, they are easier to trace.

Third, objects allow us to conceal certain details of their operations from other objects so that these operations may not be adversely affected by other objects. This is known as the *information-hiding* principle. In languages that are not object-oriented, this principle can be found to some extent in the case of local variables, or as in Pascal, in local functions or procedures, which can only be used and accessed by the function defining them. This is, however, a very tight hiding or no hiding at all. Sometimes we may need to use (again, as in Pascal) a function $f2$ defined in $f1$ outside of $f1$, but we cannot. Sometimes we may need to access some data local to $f1$ without exactly knowing the structure of these data, but we cannot. Hence, some modification is needed and it is accomplished in OOLs.

An object in OOL is like a watch. As users, we are interested in what the hands show, but not in the inner workings of the watch. We are aware that there are gears and springs inside the watch. But because we usually know very little about why all these parts are in a particular configuration, we should not have access to this mechanism so that we do not damage it, inadvertently or on purpose. This mechanism is

hidden from us, we have no immediate access to it, and the watch is protected and works better than when its mechanism is open for everyone to see.

An object is like a black box whose behavior is very well defined, and we use the object because we know what it does, not because we have an insight into how it does it. This opacity of objects is extremely useful for maintaining them independently of each other. If communication channels between the objects are well defined, then changes made inside an object can affect other objects only as much as these changes affect the communication channels. Knowing the kind of information sent out and received by an object, the object can be replaced more easily by another object more suitable in a particular situation: A new object can perform the same task differently but more quickly in a certain hardware environment. An object discloses only as much as is needed for the user to utilize it. It has a public part which can be accessed by any user when the user sends a message matching any of the member function names revealed by the object. In this public part, the object displays to the user buttons which can be pushed to invoke the object's operations. The user knows only the names of these operations and the expected behavior.

Information hiding tends to blur the dividing line between data and operations. In Pascal-like languages, the distinction between data and functions or procedures is clear and rigid. They are defined differently and their roles are very distinct. OOLs put data and methods together, and to the user of the object, this distinction is much less noticeable. To some extent, this incorporates the features of functional languages. LISP, one of the earliest programming languages, allows the user to treat data and functions similarly, since the structure of both is the same.

We have already made a distinction between particular objects and object types or classes. We write functions to be used with different variables, and by analogy, we do not like to be forced to write as many object declarations as the number of objects required by the program. Certain objects are of the same type and we would like only to use a reference to a general object specification. For single variables, we make a distinction between type declaration and variable declaration. In the case of objects, we have a class declaration and object instantiation. For instance, in the following class declaration, `C` is a class and `object1` through `object3` are objects.

```
class C {
public:
    C(char *s = "", int i = 0, double d = 1) {
        strcpy(dataMember1,s);
        dataMember2 = i;
        dataMember3 = d;
    }
    void memberFunction1() {
        cout << dataMember1 << ' ' << dataMember2 << ' '
             << dataMember3 << endl;
    }
    void memberFunction2(int i, char *s = "unknown") {
        dataMember2 = i;
```

```
        cout << i << " received from " << s << endl;
    }
protected:
    char dataMember1[20];
    int dataMember2;
    double dataMember3;
};
```

```
C object1("object1",100,2000), object2("object2"), object3;
```

Message passing is equivalent to a function call in traditional languages. However, to stress the fact that in OOLs the member functions are relative to objects, this new term is used. For example, the call to public `memberFunction()` in `object1`,

```
object1.memberFunction1();
```

is seen as message `memberFunction1()` sent to `object1`. Upon receiving the message, the object invokes its member function and displays all relevant information. Messages can include parameters so that

```
object1.memberFunction2(123);
```

is the message `memberFunction2()` with parameter 123 received by `object1`.

The lines containing these messages are either in the main program, in a function, or in a member function of another object. Therefore, the receiver of the message is identifiable, but not necessarily the sender. If `object1` receives the message `memberFunction1()`, it does not know where the message originated. It only responds to it by displaying the information `memberFunction1()` encapsulates. The same goes for `memberFunction2()`. Therefore, the sender may prefer sending a message that also includes its identification, as follows:

```
object1.memberFunction2(123, "object12");
```

A powerful feature of C++ is the possibility of declaring generic classes by using type parameters in the class declaration. For example, if we need to declare a class that uses an array for storing some items, then we may declare this class as

```
class intClass {
    int storage[50];
    ...................
};
```

However, in this way, we limit the usability of this class to integers only; if we need a class that performs the same operations as `intClass` except that it operates on float numbers, then a new declaration is needed, such as

```
class floatClass {
    float storage[50];
    ...................
};
```

If storage is to hold structures, or pointers to characters, then two more classes must be declared. It is much better to declare a generic class and decide to what type of items the object is referring only when defining the object. Fortunately, C++ allows us to declare a class in this way, and the declaration for the example is

```
template<class genType>
class genClass {
    genType storage[50];
     ..................
};
```

Later, we make the decision about how to initialize genType:

```
genClass<int> intObject;
genClass<float> floatObject;
```

This generic class manifests itself in different forms depending on the specific declaration. One generic declaration suffices for enabling such different forms.

We can go even further than that by not committing ourselves to 50 cells in storage and by delaying that decision until the object definition stage. But just in case, we may leave a default value so that the class declaration is now

```
template<class genType, int size = 50>
class genClass {
    genType storage[size];
 ..................
};
```

The object definition is now

```
genClass<int> intObject1; // use the default size;
genClass<int,100> intObject2;
genClass<float,123> floatObject;
```

This method of using generic types is not limited to classes only; we can use them in function declarations as well. For example, the standard operation for swapping two values can be defined by the function

```
template<class genType>
void swap(genType& el1, genType& el2) {
    genType tmp = el1; el1 = el2; el2 = tmp;
}
```

This example also indicates the need for adapting built-in operators to specific situations. If genType is a number, a character, or a structure, then the assignment operator, =, performs its function properly. But if genType is an array, then we can expect a problem in swap(). The problem can be resolved by overloading the assignment operator by adding to it the functionality required by a specific data type.

After a generic function has been declared, a proper function can be generated at compilation time. For example, if the compiler sees two calls,

```
int n = 5, m = 7;
float x=1.5, y = 2.5;
swap(n,m); // swap two integers;
swap(x,y); // swap two floats;
```

it generates two swap functions to be used during execution of the program.

1.3 INHERITANCE

OOLs allow for creating a hierarchy of classes so that objects do not have to be instantiations of a single class. Before discussing the problem of inheritance, consider the following class definitions:

```
class BaseClass {
public:
    BaseClass() { }
    void f(char *s = "unknown") {
        cout << "Function f() in BaseClass called from " << s << endl;
        h();
    }
protected:
    void g(char *s = "unknown") {
        cout << "Function g() in BaseClass called from " << s << endl;
    }
private:
    void h() {
        cout << "Function h() in BaseClass\n";
    }
};
class Derived1Level1 : public virtual BaseClass {
public:
    void f(char *s = "unknown") {
        cout << "Function f() in Derived1Level1 called from " << s << endl;
        g("Derived1Level1");
        h("Derived1Level1");
    }
    void h(char *s = "unknown") {
        cout << "Function h() in Derived1Level1 called from " << s << endl;
    }
};
class Derived2Level1 : public virtual BaseClass {
public:
    void f(char *s = "unknown") {
        cout << "Function f() in Derived2Level1 called from " << s << endl;
        g("Derived2Level1");
//      h();  // error: BaseClass::h() is not accessible
    }
};
```

```
class DerivedLevel2 : public Derived1Level1, public Derived2Level1 {
public:
    void f(char *s = "unknown") {
        cout << "Function f() in DerivedLevel2 called from " << s << endl;
        g("DerivedLevel2");
        Derived1Level1::h("DerivedLevel2");
        BaseClass::f("DerivedLevel2");
    }
};
```

A sample program is

```
void main() {
 1  BaseClass bc;
 2  Derived1Level1 d111;
 3  Derived2Level1 d211;
 4  DerivedLevel2 dl2;
 5  bc.f("main(1)");
//  bc.g(); // error: BaseClass::g() is not accessible
//  bc.h(); // error: BaseClass::h() is not accessible
 6  d111.f("main(2)");
//  d111.g(); // error: BaseClass::g() is not accessible
 7  d111.h("main(3)");
 8  d211.f("main(4)");
//  d211.g(); // error: BaseClass::g() is not accessible
//  d211.h(); // error: BaseClass::h() is not accessible
 9  dl2.f("main(5)");
//  dl2.g();  // error: BaseClass::h() is not accessible
10  dl2.h();
}
```

This sample produces the following output:

```
5  Function f() in BaseClass called from main(1)
   Function h() in BaseClass
6  Function f() in Derived1Level1 called from main(2)
   Function g() in BaseClass called from Derived1Level1
   Function h() in Derived1Level1 called from Derived1Level1
7  Function h() in Derived1Level1 called from main(3)
8  Function f() in Derived2Level1 called from main(4)
   Function g() in BaseClass called from Derived2Level1
9  Function f() in DerivedLevel2 called from main(5)
   Function g() in BaseClass called from DerivedLevel2
   Function h() in Derived1Level1 called from DerivedLevel2
   Function f() in BaseClass called from DerivedLevel2
   Function h() in BaseClass
10 Function h() in Derived1Level1 called from unknown
```

The class `BaseClass` is called a *base class* or a *superclass*, and other classes are called *subclasses* or *derived classes* because they are derived from the superclass in that

they can use the data members and member functions specified in `BaseClass` as `protected` or `public`. They inherit all these members from their base class so that they do not have to repeat the same definitions. However, a derived class can override the definition of a member function by introducing its own definition. In this way, both the base class and the derived class have some measure of control over their member functions.

The base class can decide which member functions and data members can be revealed to derived classes so that the principle of information hiding holds not only with respect to the user of the base class but also to the derived classes. Moreover, the derived class can decide which parts of the public and protected member functions and data members to retain and use and which to modify. For example, both `Derived1Level1` and `Derived2Level1` define their own versions of `f()`. However, the access to the member function with the same name in any of the classes higher up in the hierarchy is still possible by preceding the function with the name of the class and the scope operator, as shown in the call of `BaseClass::f()` from `f()` in `DerivedLevel2`.

A derived class can add some new members of its own. Such a class can become a base class for other classes that can be derived from it so that the inheritance hierarchy can be deliberately extended. For example, the class `Derived1Level1` is derived from `BaseClass`, but at the same time, it is the base class for `DerivedLevel2`.

Inheritance in our examples is specified as public by using the word `public` after the semicolon in the heading of the definition of a derived class. Public inheritance means that public members of the base class are also public in the derived class and protected members are also protected. In the case of protected inheritance (with the word `protected` in the heading of the definition), both public and protected members of the base class become protected in the derived class. Finally, for private inheritance, both public and protected members of the base class become private in the derived class. In all types of inheritance, private members of the base class are inaccessible to any of the derived classes. For example, an attempt to call `h()` from `f()` in `Derived2Level1` causes a compilation error, "`BaseClass::h()` is not accessible." However, a call of `h()` from `f()` in `Derived1Level1` causes no problem because it is a call to `h()` defined in `Derived1Level1`.

Protected members of the base class are accessible only to derived classes and not to nonderived classes. For this reason, both `Derived1Level1` and `Derived2Level1` can call `BaseClass`'s protected member function `g()`, but a call to this function from `main()` is rendered illegal.

A derived class does not have to be limited to one base class only. It can be derived from more than one base class. For example, `DerivedLevel2` is defined as a class derived from both `Derived1Level1` and `Derived2Level1`, inheriting in this way all the member functions of `Derived1Level1` and `Derived2Level1`. However, `DerivedLevel2` also inherits the same member functions from `BaseClass` twice because both classes used in the definition of `DerivedLevel2` are derived from `BaseClass`. This is redundant at best and at worst can cause a compilation error, "member is ambiguous `BaseClass::g()` and `BaseClass::g()`." To prevent this from happening, the definitions of the two classes include the modifier `virtual`, which means that `DerivedLevel2`

contains only one copy of each member function from `BaseClass`. A similar problem arises if `f()` in `DerivedLevel2` calls `h()` without the preceding scope operator and class name, `Derived1Level1::h()`. It does not matter that `h()` is private in `BaseClass` and inaccessible to `DerivedLevel2`. An error would be printed, "member is ambiguous `Derived1Leve1::h()` and `BaseClass::h()`."

1.4 POINTERS

Variables used in a program can be considered as boxes that are never empty; they are filled with some content either by the programmer or, if uninitialized, by the operating system. Such a variable has at least two attributes: the content or value and the location of the box or variable in computer memory. This content can be a number, character, or a compound item such as a structure or union. However, this content can also be the location of another variable, and variables with such contents are called *pointers*. Pointers are usually auxiliary variables which allow us to access the values of other variables indirectly. A pointer is analogous to a road sign that leads us to a certain location or to a slip of paper on which an address has been jotted down. They are variables leading to variables, humble auxiliaries which point to some other variables as the focus of attention.

For example, in the declaration

```
int i = 15, j, *p, *q;
```

`i` and `j` are numerical variables and `p` and `q` are pointers to numbers where the star in front of `p` and `q` indicates their function. Assuming that the addresses of the variables `i`, `j`, `p`, and `q` are 1080, 1082, 1084, and 1086, then after assigning 15 to `i` in the declaration, the positions and values of the variables in computer memory are as in Figure 1.1a.

Now, we could make the assignment `p = i` (or `p = (int*) i` if the compiler does not accept it), but the variable `p` was created to store the address of an integer variable, not its value. Therefore, the proper assignment is `p = &i` where the ampersand in front of `i` means that the address of `i` is meant and not its content. Figure 1.1b illustrates this situation. In Figure 1.1c, the arrow from `p` to `i` indicates that `p` is a pointer that holds the address of `i`.

& = address

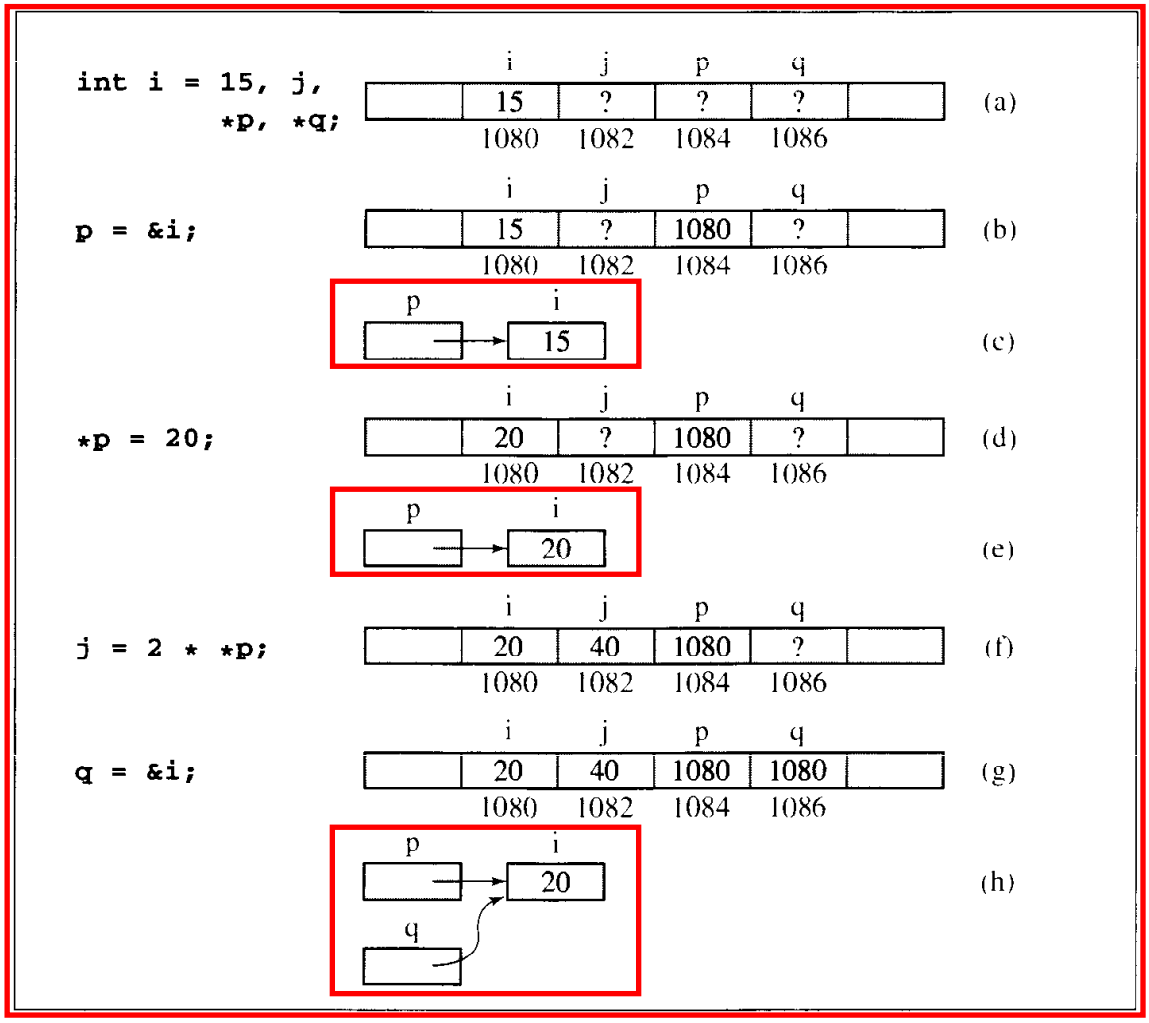
We have to be able to distinguish the value of `p`, which is an address, from the value of the location whose address the pointer holds. For example, to assign 20 to the variable pointed to by `p`, the assignment statement is

* = value

```
*p = 20;
```

The star `*` here is an indirection operator which forces the system to first retrieve the contents of `p`, then access the location whose address has just been retrieved from `p`, and only afterward, assign 20 to this location (Figure 1.1d). Figures 1.1e through 1.1n give more examples of assignment statements and how the values are stored in computer memory.

FIGURE **1.1** Changes of values after assignments are made using pointer variables. Note that (b) and (c) show the same situation and so do (d) and (e), (g) and (h), (i) and (j), (k) and (l), and (m) and (n).



In fact, pointers—like all variables—also have two attributes: a content and a location. This location can be stored in another variable, which then becomes a pointer to a pointer.

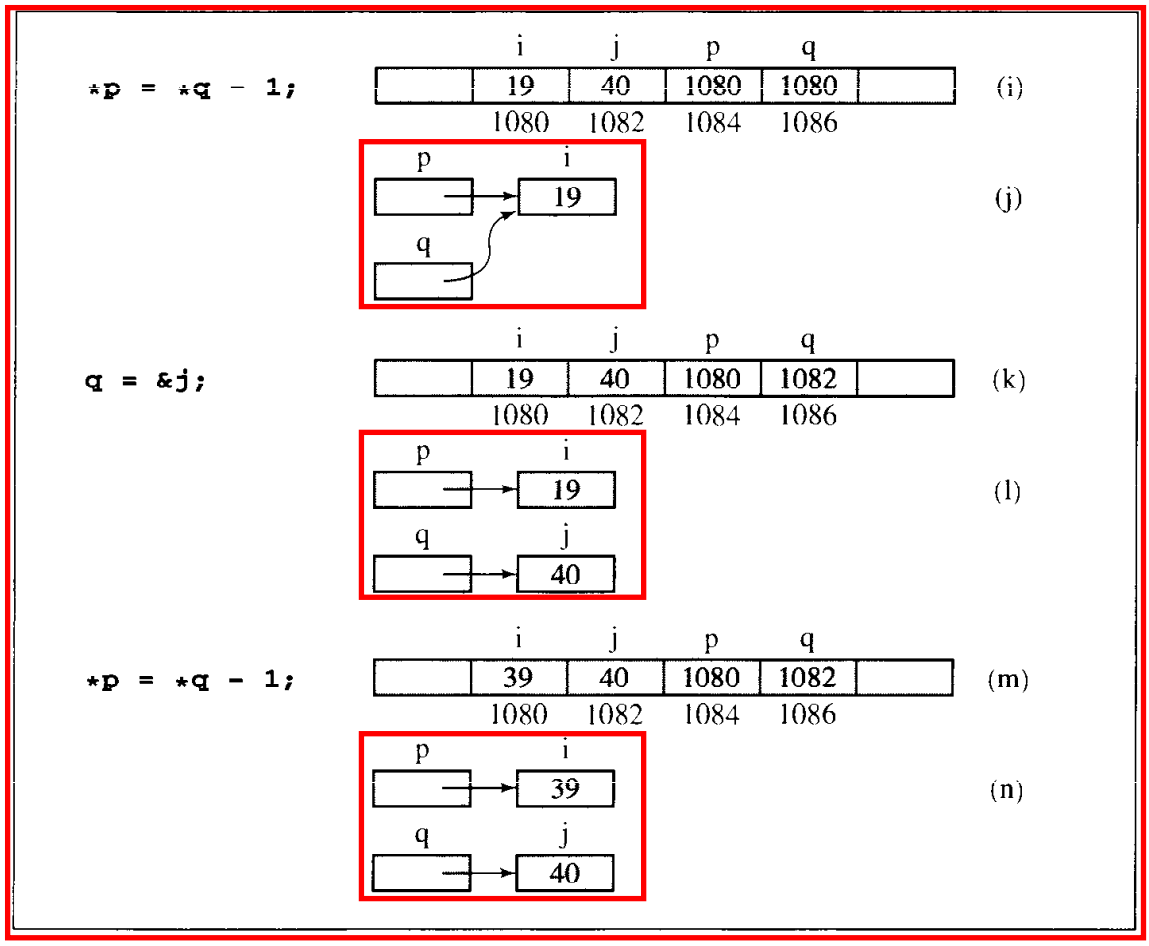
In Figure 1.1, addresses of variables were assigned to pointers. Pointers can, however, refer to anonymous locations that are accessible only through their addresses and not—like variables—by their names. These locations must be set apart by the memory manager, which is performed dynamically during the run of the program, unlike for variables, whose locations are allocated at compilation time.

To dynamically allocate and deallocate memory, two functions are used. One function, `new`, takes from memory as much space as needed to store an object whose type follows `new`. For example, with the instruction

```
p = new int;
```

the program requests from memory manager enough space to store one integer, and the address of this portion of memory is stored in `p`. Now the values can be assigned

FIGURE **1.1** *(continued)*



to the memory block pointed to by p only indirectly through a pointer, either pointer p or any other pointer q which was assigned the address stored in p with the assignment q = p.

If the space occupied by the integer accessible from p is no longer needed, it can be returned to the pool of free memory locations managed by the operating system by issuing the instruction

```
delete p;
```

However, after executing this statement, the addresses of the released memory block is still in p, although the block, as far as the program is concerned, does not exist anymore. It is like treating an address of a house that has been demolished as the address of the existing location. If we use this address to find someone, the result can be easily foreseen. Similarly, if after issuing the `delete` statement we do not erase the address from the pointer variable participating in deletion, the result is potentially dangerous, and we can crash the program when trying to access nonexisting locations, particularly for more complex objects than numerical values. This is the *dangling reference*

problem. To avoid this problem, an address has to be assigned to a pointer; if it cannot be the address of any location, it should be a null address which is simply 0. After execution of the assignment

```
p = 0;
```

we may not say that p refers null or points to null but that p becomes null or p is null.

1.4.1 Pointers and Arrays

In the last example, the pointer p refers to a block of memory that holds one integer. A more interesting situation is when a pointer refers to a data structure that is created and modified dynamically. This is a situation we would like to have to overcome the restrictions imposed by arrays. Arrays in C++, and in most programming languages, have to be declared in advance; therefore, their sizes have to be known before the program starts. This means that the programmer needs a fair knowledge of the problem being programmed to choose the right size for the array. If the size is too big, then the array unnecessarily occupies memory space which is basically wasted. If the size is too small, the array can overflow with data and the program will abort. Sometimes the size of the array simply cannot be predicted; therefore, the decision is delayed until run time, and then enough memory is allocated to hold the array.

The problem is solved with the uses of pointers. Consider Figure 1.1b. In this figure, pointer p points to location 1080. But it also allows accessing location 1082, 1084, and so forth because the locations are evenly spaced. For example, to access the value of variable j, which is a neighbor of i, it is enough to add the size of an integer variable to the address of i stored in p to access the value of j also from p. And this is basically the way C++ handles arrays.

Consider the following declarations:

```
int a[5], *p;
```

The declarations specify that a is a pointer to a block of memory that can hold five integers. The pointer is fixed; that is, a should be treated as a constant so that any attempt to assign a value to a, as in

```
a = p;
```

or in

```
a++;
```

is considered a compilation error. Because a is a pointer, pointer notation can be used to access cells of the array a. For example, an array notation used in the loop which adds all the numbers in a,

```
for (sum = a[0], i = 1; i < 5; i++)
        sum += a[i];
```

can be replaced by a pointer notation

```
for (sum = *a, i = 1; i < 5; i++)
        sum += *(a + i);
```

or by

```
for (sum = *a, p = a+1; p < a+5; p++)
        sum += *p;
```

Note that `a+1` is a location of the next cell of the array `a` so that `a+1` is equivalent to `&a[1]`. Thus, if `a` equals 1020, then `a+1` is not 1021 but 1022 because pointer arithmetic depends on the type of pointed entity. For example, after declarations

```
char b[5];
long c[5];
```

and assuming that `b` equals 1050 and `c` equals 1055, `b+1` equals 1051 because one character occupies one byte, and `c+1` equals 1059 because one long number occupies four bytes. The reason for these results of pointer arithmetic is that the expression `c+i` denotes the memory address `c+i*sizeof(long)`.

In this discussion, the array `a` is declared statically by specifying in its declaration that it contains five cells. The size of the array is fixed for the duration of the program run. But arrays can also be declared dynamically. To that end, pointer variables are used. For example, the assignment

```
p = new int[n];
```

allocates enough room to store `n` integers. Pointer `p` can be treated as an array variable so that array notation can be used. For example, the sum of numbers in the array `p` can be found with the code that uses array notation,

```
for (sum = p[0], i = 1; i < n; i++)
        sum += p[i];
```

a pointer notation that is a direct rendition of the previous loop,

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

or a pointer notation that uses two pointers,

```
for (sum = *p, q = p+1; q < p+n; q++)
        sum += *q;
```

Because `p` is a variable, it can be assigned a new array. But if the array currently pointed to by `p` is no longer needed, it should be disposed of by the instruction

```
delete [] p;
```

Note the use of empty brackets in the instruction. The brackets indicate that `p` points to an array.

A very important type of arrays is strings, or arrays of characters. There are many predefined functions that operate on strings. The names of these functions start with `str`, as in `strlen(s)` to find the length of the string `s` or `strcpy(s1,s2)` to copy string `s2` to `s1`. It is important to remember that all these functions assume that strings are ended with the null character `'\0'`. For example, `strcpy(s1,s2)` continues copying until it finds this character in `s2`. If a programmer does not include this character in `s2`, copying stops when the first occurrence of this character is found

somewhere in computer memory after location s2. This means that copying is performed to locations outside s1, which eventually may lead the program to crash.

1.4.2 Pointers and Copy Constructors

Some problems can arise when pointer data members are not handled properly when copying data from one object to another. Consider the following definition:

```
struct Node {
    char *name;
    int age;
    Node(char *n = "", int a = 0) {
       name = new char[strlen(n)+1];
       strcpy(name,n);
       age = a;
    }
};
```

The intention of the declarations

```
Node node1("Roger",20), node2(node1); //or node2 = node1;
```

is to create object node1, assign values to the two data members in node1, and then create object node2 and initialize its data members to the same values as in node1. These objects are to be independent entities so that assigning values to one of them should not affect values in the other. However, after the assignments

```
strcpy(node2.name,"Wendy");
node2.age = 30;
```

the printing statement

```
cout<<node1.name<<' '<<node1.age<<' '<<node2.name<<' '<<node2.age;
```

generates the output

```
Wendy 30 Wendy 20
```

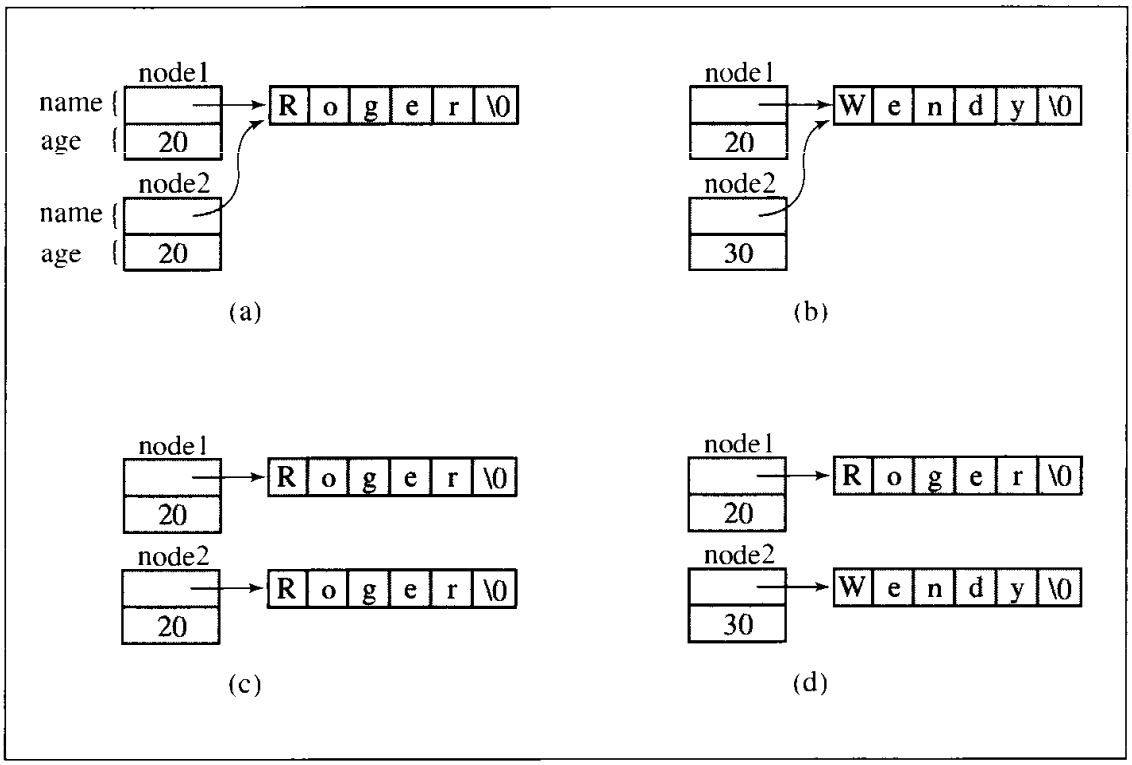
The ages are different, but the names in the two objects are the same. What happened? The problem is that the definition of Node does not provide a copy constructor

```
Node(const Node&);
```

which is necessary to execute the declaration node2(node1) to initialize node1. If a user copy constructor is missing, the constructor is generated automatically by the compiler. But the compiler-generated copy constructor performs member-by-member copying. Because name is a pointer, the copy constructor copies the string address node1.name to node2.name, not the string content, so that right after execution of the declaration, the situation is as in Figure 1.2a. Now if the assignments

```
strcpy(node2.name,"Wendy");
node2.age = 30;
```

FIGURE **1.2** Illustrating the necessity of using a copy constructor for objects with pointer members.



are executed, `node2.age` is properly updated, but the string `"Roger"` pointed to by `name` member of both objects is overwritten by `"Wendy"`, which is also pointed to by the two pointers (Figure 1.2b). To prevent this from happening, the user must define a proper copy constructor, as in

```
struct Node {
    char *name;
    int age;
    Node(char *n = 0, int a = 0) {
         name = new char[strlen(n)+1];
         strcpy(name,n);
         age = a;
    }
    Node(const Node& n) { // copy constructor;
         name = new char[strlen(n.name)+1];
         strcpy(name,n.name);
         age = n.age;
    }
};
```

With the new constructor, the declaration `node2(node1)` generates another copy of `"Roger"` pointed to by `node2.name` (Figure 1.2c), and the assignments to data members in one object have no effect on members in the other object so that after execution of the assignments

```
strcpy(node2.name,"Wendy");
node2.age = 30;
```

the object `node 1` remains unchanged, as illustrated in Figure 1.2d.

Note that a similar problem is raised by the assignment operator. If a definition of the assignment operator is not provided by the user, an assignment

```
node1 = node2;
```

performs member-by-member copying, which leads to the same problem as in Figure 1.2a–b. To avoid the problem, the assignment operator has to be overloaded by the user. For `Node`, the overloading is accomplished by

```
Node& operator=(const Node& n) {
    if (this != &n) { // no assignment to itself;
        if (name != 0)
            delete [] name;
        name = new char[strlen(n.name)+1];
        strcpy(name,n.name);
        age = n.age;
    }
    return *this;
}
```

In this code, a special pointer `this` is used. Each object can access its own address through the pointer `this` so that `*this` is the object itself.

1.4.3 Pointers and Destructors

What happens to locally defined objects of type `Node`? Like all local items, they are destroyed in the sense that they become unavailable outside the block in which they are defined, and memory occupied by them is also released. But although memory occupied by an object of type `Node` is released, not all the memory related to this object becomes available. One of the data members of this object is a pointer to a string; therefore, memory occupied by the pointer data member is released, but memory taken by the string is not. After the object is destroyed, the string previously available from its data member `name` becomes inaccessible (if not assigned to `name` of some other object or to a string variable) and memory occupied by this string can no longer be released. This is a problem with objects that have data members pointing to dynamically allocated locations. To avoid the problem, the class definition should include a definition of a destructor. A *destructor* is a function that is automatically invoked when an object is destroyed, which takes place upon exit from the block in which the object is defined or upon the call of `delete`. Destructors take no

arguments and return no values so that there can be only one destructor per class. For class `Node`, a destructor can be defined as

```
~Node() {
     if (name != 0)
          delete [] name;
}
```

1.4.4 Pointers and Reference Variables

Consider the following declarations:

```
int n = 5, *p = &n, &r = n;
```

Variable `p` is declared as being of type `int*`, a pointer to an integer, and `r` is of type `int&`, an integer reference variable. A reference variable must be initialized in its declaration as a reference to a particular variable, and this reference cannot be changed. This means that a reference variable cannot be null. A reference variable `r` can be considered a different name for a variable `n` so that if `n` changes then `r` changes as well. This is because a reference variable is implemented as a constant pointer to the variable.

After the three declarations, the printing statement

```
cout << n << ' ' << *p << ' ' << r << endl;
```

outputs 5 5 5. After the assignment

```
n = 7;
```

the same printing statement outputs 7 7 7. Also, an assignment

```
*p = 9;
```

gives the output 9 9 9, and the assignment

```
r = 10;
```

leads to the output 10 10 10. These statements indicate that in terms of notation, what we can accomplish with dereferencing of pointer variables is accomplished without dereferencing of reference variables. This is no accident because, as mentioned, reference variables are implemented as constant pointers. Instead of the declaration

```
int& r = n;
```

we can use a declaration

```
int *const r = &n;
```

where `r` is a constant pointer to an integer, which means that the assignment

```
r = q;
```

where `q` is another pointer, is an error because the value of `r` cannot change. However, the assignment

```
*r = 1;
```

is acceptable if n is not a constant integer.

It is important to note the difference between the type `int *const` and the type `const int *`. The latter is a type of pointer to a constant integer:

```
const int *s = &m;
```

after which the assignment

```
s = &m;
```

where m in an integer (whether constant or not) is admissible, but the assignment

```
*s = 2;
```

is erroneous, even if m is not a constant.

Reference variables are used in passing arguments by reference to function calls. Passing by reference is required if an actual parameter should be changed permanently during execution of a function. This can be accomplished with pointers (and in C, this is the only mechanism available for passing by reference) or with reference variables. For example, after declaring a function

```
void f1(int i, int* j, int& k) {
    i = 1;
    *j = 2;
    k = 3;
}
```

the values of the variables

```
int n1 = 4, n2 = 5, n3 = 6;
```

after executing the call

```
f1(n1,&n2,n3);
```

are n1 = 4, n2 = 2, n3 = 3.

Reference type is also used in indicating return type of functions. For example, having defined the function

```
int& f2(int a[], int i) {
    return a[i];
}
```

and declaring the array

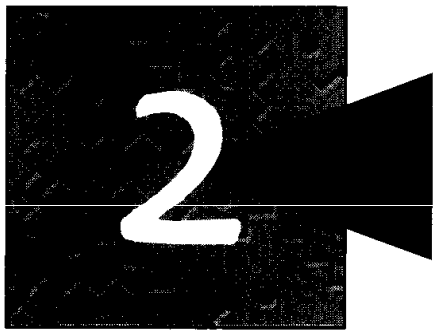
```
int a[] = {1,2,3,4,5};
```

we can use `f2()` on any side of the assignment operator. For instance, on the right-hand side,

```
n = f2(a,3);
```

or on the left-hand side,

Complexity Analysis



2.1 Computational and Asymptotic Complexity

The same problem can frequently be solved with algorithms that differ in efficiency. The differences between the algorithms may be immaterial for processing a small number of data items, but these differences grow proportionally with the amount of data. To compare the efficiency of algorithms, a measure of the degree of difficulty of an algorithm called *computational complexity* was developed by Juris Hartmanis and Richard E. Stearns.

Computational complexity indicates how much effort is needed to apply an algorithm or how costly it is. This cost can be measured in a variety of ways and the particular context determines its meaning. This book concerns itself with the two efficiency criteria: time and space. The factor of time is more important than that of space, so efficiency considerations usually focus on the amount of time elapsed when processing data. However, the most inefficient algorithm run on a Cray computer can execute much faster than the most efficient algorithm run on a PC, so run time is always system-dependent. For example, to compare a hundred algorithms, all of them would have to be run on the same machine. Furthermore, the results of run-time tests depend on the language in which a given algorithm is written even if the tests are performed on the same machine. If programs are compiled, they execute much faster than when they are interpreted. A program written in C or Pascal may be 20 times faster than the same program encoded in BASIC or LISP.

To evaluate an algorithm's efficiency, real-time units such as microseconds and nanoseconds should not be used. Rather, logical units that express a relationship between the size n of a file or an array and the amount of time t required to process the data should be used. If there is a linear relationship between the size n and time t, that is, $t_1 = cn_1$, then an increase of data by a factor of 5 results in the increase of the execution

time by the same factor, if $n_2 = 5n_1$, then $t_2 = 5t_1$. Similarly, if $t_1 = \log_2 n$, then doubling n increases t by only one unit of time. Therefore, if $t_2 = \log_2(2n)$, then $t_2 = t_1 + 1$.

A function expressing the relationship between n and t is usually much more complex, and calculating such a function is important only in regard to large bodies of data; any terms which do not substantially change the function's magnitude should be eliminated from the function. The resulting function gives only an approximate measure of efficiency of the original function. However, this approximation is sufficiently close to the original, especially for a function that processes large quantities of data. This measure of efficiency is called *asymptotic complexity* and is used when disregarding certain terms of a function to express the efficiency of an algorithm or when calculating a function is difficult or impossible and only approximations can be found. To illustrate the first case, consider the following example:

$$f(n) = n^2 + 100n + \log_{10} n + 1000 \tag{2.1}$$

For small values of n, the last term, 1000, is the largest. When n equals 10, the second ($100n$) and last (1000) terms are on equal footing with the other terms making a small contribution to the function value. When n reaches the value of 100, the first and the second terms make the same contribution to the result. But when n becomes larger than 100, the contribution of the second term becomes less significant. Hence, for large values of n, due to the quadratic growth of the first term (n^2), the value of the function f depends mainly on the value of this first term, as Figure 2.1 demonstrates. Other terms can be disregarded in the long run.

2.2 Big-O Notation

The most commonly used notation for specifying asymptotic complexity, that is, for estimating the rate of function growth, is the big-O notation introduced in 1894 by Paul Bachmann. Given two positive-valued functions f and g, consider the following definition:

Definition 1: $f(n)$ is $O(g(n))$ if there exist positive numbers c and N such that $f(n) \leq cg(n)$ for all $n \geq N$.

This definition reads: f is big-O of g if there is a positive number c such that f is not larger than cg for sufficiently large ns, that is, for all ns larger than some number N. The relationship between f and g can be expressed by stating either that $g(n)$ is an upper bound on the value of $f(n)$ or that, in the long run, f grows at most as fast as g.

The problem with this definition is that, first, it states only that there must exist certain c and N, but it does not give any hint how to calculate these constants. Second, it does not put any restrictions on these values and gives little guidance in situations when there are many candidates. In fact, there are usually infinitely many pairs of cs and Ns that can be given for the same pair of functions f and g. For example, for

$$f(n) = 2n^2 + 3n + 1 = O(n^2) \tag{2.2}$$

where $g(n) = n^2$, candidate values for c and N are shown in Figure 2.2.

FIGURE **2.1** The growth rate of all terms of function $f(n) = n^2 + 100n + \log_{10} n + 1000$.

n	f(n)	n^2		100n		$\log_{10}$n		1000	
	Value	Value	%	Value	%	Value	%	Value	%
1	1,101	1	0.1	100	9.1	0	0.0	1000	90.82
10	2,101	100	4.76	1,000	47.6	1	0.05	1000	47.62
100	21002	10,000	47.6	10,000	47.6	2	0.991	1000	4.76
1,000	1,101,003	1,000,000	90.8	100,000	9.1	3	0.0003	1000	0.09
10,000	101,001,004	100,000,000	99.0	1,000,000	0.99	4	0.0	1000	0.001
100,000	10,010,001,005	10,000,000,000	99.9	10,000,000	0.099	5	0.0	1000	0.00

FIGURE **2.2** Different values of c and N for function $f(n) = 2n^2 + 3n + 1 = O(n^2)$ calculated according to the definition of big-O.

c	≥ 6	$\geq 3\frac{3}{4}$	$\geq 3\frac{1}{9}$	$\geq 2\frac{13}{16}$	$\geq 2\frac{16}{25}$	...	$\rightarrow$	2
N	1	2	3	4	5	...	$\rightarrow$	∞

We obtain these values by solving the inequality:

$$2n^2 + 3n + 1 \leq cn^2$$

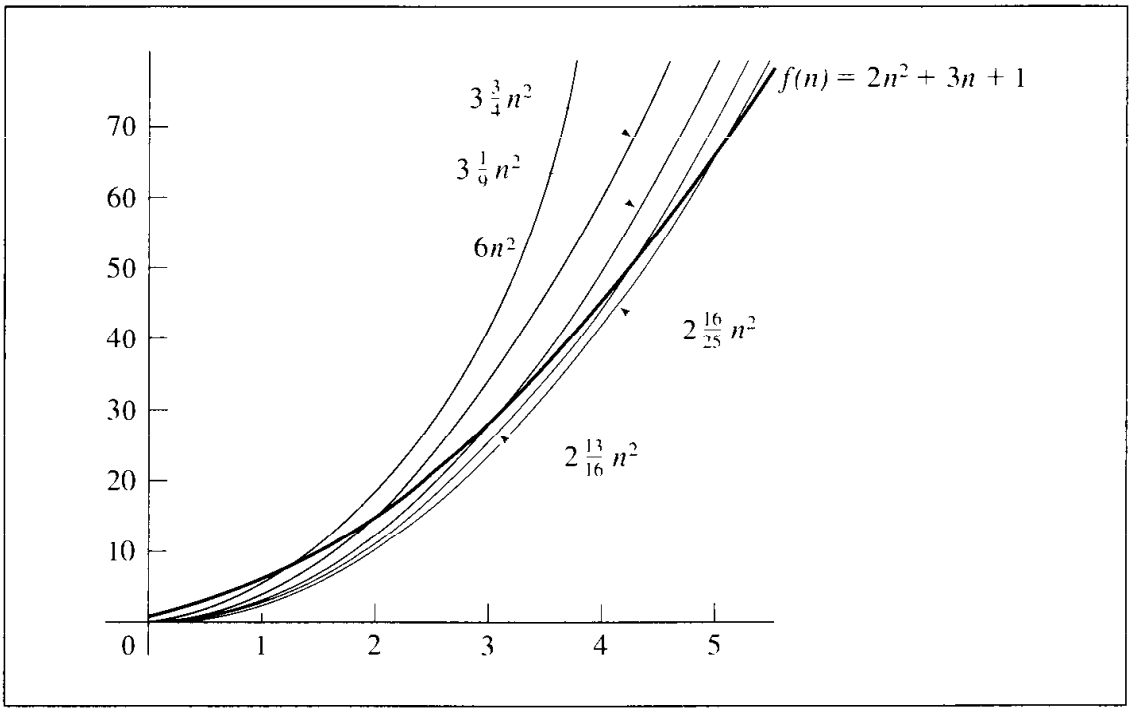
or equivalently

$$2 + \frac{3}{n} + \frac{1}{n^2} \leq c$$

for different ns. The first inequality results in substituting the quadratic function from Equation 2.2 for $f(n)$ in the definition of the big-O notation and n^2 for $g(n)$. Since it is one inequality with two unknowns, different pairs of constants c and N for the same function $g(= n^2)$ can be determined. To choose the best c and N, it should be determined for which N a certain term in f becomes the largest and stays the largest. In Equation 2.2, the only candidates for the largest term are $2n^2$ and $3n$; these terms can be compared using the inequality $2n^2 > 3n$ that holds for $n > 1$. Thus, $N = 2$ and $c \geq 3\frac{3}{4}$, as Figure 2.2 indicates.

What is the practical significance of the pairs of constants just listed? All of them are related to the same function $g(n) = n^2$ and to the same $f(n)$. For a fixed g, an infinite

FIGURE 2.3 Comparison of functions for different values of c and N from Figure 2.2.



number of pairs of cs and Ns can be identified. The point is that f and g grow at the same rate. The definition states, however, that g is almost always greater than or equal to f if it is multiplied by a constant c. "Almost always" means for all ns not less than a constant N. The crux of the matter is that the value of c depends on which N is chosen and vice versa. For example, if 1 is chosen as the value of N—that is, if g is multiplied by c so that $cg(n)$ will not be less than f right away—then c has to be equal to 6 or greater. If $cg(n)$ is greater than or equal to $f(n)$ starting from $n = 2$, then it is enough that c is equal to 3.75. The constant c has to be at least $3\frac{1}{9}$ if $cg(n)$ is not less than $f(n)$ starting from $n = 3$. Figure 2.3 shows the graphs of the functions f and g. The function g is plotted with different coefficients c. Also, N is always a point where the functions $cg(n)$ and f intersect each other.

The inherent imprecision of the big-O notation goes even further, since there can be infinitely many functions g for a given function f. For example, the f from Equation 2.2 is big-O not only of n^2, but also of n^3, n^4, . . . , n^k, . . . for any $k \geq 2$. To avoid this embarrassment of riches, the smallest function g is chosen, n^2 in this case.

The approximation of function f can be refined using big-O notation only for the part of the equation suppressing irrelevant information. For example, in Equation 2.1, the contribution of the third and last terms to the value of the function can be omitted (see Equation 2.3).

$$f(n) = n^2 + 100n + O(\log_{10} n) \tag{2.3}$$

Similarly, the function f in Equation 2.2 can be approximated as

$$f(n) = 2n^2 + O(n) \tag{2.4}$$

2.3 Properties of Big-O Notation

Big-O notation has some helpful properties that can be used when estimating the efficiency of algorithms.

Fact 1. (transitivity) If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ if $O(h(n))$. (This can be rephrased as $O(O(g(n)))$ is $O(g(n))$.)

Proof: According to the definition, $f(n)$ is $O(g(n))$ if there exist positive numbers c_1 and N_1 such that $f(n) \le c_1 g(n)$ for all $n \ge N_1$, and $g(n)$ is $O(h(n))$ if there exist positive numbers c_2 and N_2 such that $g(n) \le c_2 h(n)$ for all $n \ge N_2$.

Hence, $c_1 g(n) \le c_1 c_2 h(n)$ for $n \ge N$ where N is the larger of N_1 and N_2. If we take $c = c_1 c_2$, then $f(n) \le ch(n)$ for $n \ge N$, which means that f is big-O of h.

Fact 2. If $f(n)$ is $O(h(n))$ and $g(n)$ is $O(h(n))$, then $f(n) + g(n)$ is $O(h(n))$.

Proof: After setting c equal to $c_1 + c_2$, $f(n) + g(n) \le ch(n)$.

Fact 3. The function an^k is $O(n^k)$.

Proof: For the inequality $an^k \le cn^k$ to hold, $c \ge a$ is necessary.

Fact 4. The function n^k is $O(n^{k+j})$ for any positive j.

Proof: The statement holds if $c = N = 1$.

It follows from all these facts that every polynomial is big-O of n raised to the largest power, or

$$f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \text{ is } O(n^k)$$

It is also obvious that in the case of polynomials, $f(n)$ is $O(n^{k+j})$ for any positive j.

One of the most important functions in the evaluation of the efficiency of algorithms is the logarithmic function. In fact, if it can be stated that the complexity of an algorithm is on the order of the logarithmic function, the algorithm can be regarded as very good. There are an infinite number of functions that can be considered better than the logarithmic function, among which only a few, such as $O(\lg \lg n)$ or $O(1)$, have practical bearing. Before we show an important fact about logarithmic functions, let us state without proof:

Fact 5. If $f(n) = cg(n)$, then $f(n)$ is $O(g(n))$.

Fact 6. The function $\log_a n$ is $O(\log_b n)$ for any positive numbers a and $b \ne 1$.

This correspondence holds between logarithmic functions. Fact 6 states that regardless of their bases, logarithmic functions are big-O of each other; that is, all these functions have the same rate of growth.

Proof: Letting $\log_a n = x$ and $\log_b n = y$, we have, by the definition of logarithm, $a^x = n$ and $b^y = n$.

Taking ln of both sides results in

$$x \ln a = \ln n \text{ and } y \ln b = \ln n$$

Thus

$$x \ln a = y \ln b,$$

$$\ln a \log_a n = \ln b \log_b n,$$

$$\log_a n = \frac{\ln b}{\ln a} \log_b n = c \log_b n$$

which proves that $\log_a n$ and $\log_b n$ are multiples of each other. By Fact 5, $\log_a n$ is $O(\log_b n)$.

Because the base of the logarithm is irrelevant in the context of big-O notation, we can always use just one base and Fact 6 can be written as

Fact 7. $\log_a n$ is $O(\lg n)$ for any positive $a \neq 1$, where $\lg n = \log_2 n$.

2.4 Ω and Θ Notations

Big-O notation refers to the upper bounds of functions. There is a symmetrical definition for a lower bound in the definition of big-Ω:

Definition 2: The function $f(n)$ is $\Omega(g(n))$ if there exist positive numbers c and N such that $f(n) \geq cg(n)$ for all $n \geq N$.

This definition reads: f is Ω (big-omega) of g if there is a positive number c such that f is at least equal to cg for almost all ns. In other words, $cg(n)$ is a lower bound on the size of $f(n)$, or, in the long run, f grows at least at the rate of g.

The only difference between this definition and the definition of big-O notation is the direction of the inequality; one definition can be turned into the other by replacing "≥" by "≤." There is an interconnection between these two notations expressed by the equivalence

$$f(n) \text{ is } \Omega(g(n)) \text{ iff } g(n) \text{ is } O(f(n))$$

Ω notation suffers from the same profusion problem as does big-O notation: There is an unlimited number of choices for the constants c and N. For Equation 2.2, we are looking for such a c, for which $2n^2 + 3n + 1 \geq cn^2$, which is true for any $n \geq 0$, if $c \leq 2$, where 2 is the limit for c in Figure 2.2. Also, if f is an Ω of g and $h \leq g$, then f is an Ω of h; that is, if for f we can find one g such that f is an Ω of g, then we can find infinitely many. For example, the function 2.2 is an Ω of n^2 but also of n, $n^{1/2}$, $n^{1/3}$, $n^{1/4}$, . . ., and also of $\lg n$, $\lg \lg n$, . . . , and of many other functions. For practical purposes, only the closest Ωs are the most interesting, the largest lower bounds. This restriction is made implicitly each time we choose an Ω of a function f.

There are an infinite number of possible lower bounds for the function f; that is, there is an infinite set of gs such that $f(n)$ is $\Omega(g(n))$ as well as an unbounded number of possible upper bounds of f. This may be somewhat disquieting, so we restrict our attention to the smallest upper bounds and the largest lower bounds. Note that there is a common ground for big-O and Ω notations indicated by the equalities in

the definitions of these notations: Big-O is defined in terms of "≤" and Ω in terms of "≥"; "=" is included in both inequalities. This suggests a way of restricting the sets of possible lower and upper bounds. This restriction can be accomplished by the following definition of Θ (theta) notation:

Definition 3: $f(n)$ is $\Theta(g(n))$ if there exist positive numbers c_1, c_2, and N such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq N$.

This definition reads: f has an order of magnitude g, f is on the order of g, or both functions grow at the same rate in the long run. We see that $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

The only function just listed that is both big-O and Ω of the function 2.2 is n^2. However, it is not the only choice and there are still an infinite number of choices, since the functions $2n^2, 3n^2, 4n^2, \ldots$ are also Θ of function 2.2. But it is rather obvious that the simplest, n^2, will be chosen.

When applying any of these notations (big-O, Ω, and Θ), do not forget that they are approximations that hide some detail which in many cases may be considered important.

2.5 Possible Problems

All the notations serve the purpose of comparing the efficiency of various algorithms designed for solving the same problem. However, if only big-Os are used to represent the efficiency of algorithms, then some of them may be rejected prematurely. The problem is that in the definition of big-O notation, f is considered big-O of g if the inequality $f(n) \leq cg(n)$ holds in the long run for all natural numbers with a few exceptions. The number of ns violating this inequality is always finite. It is enough to meet the condition of the definition. As Figure 2.2 indicates, this number of exceptions can be reduced by choosing a sufficiently large c. However, this may be of little practical significance if the constant c in $f(n) \leq cg(n)$ is prohibitively large, say 10^8, although the function g taken by itself seems to be promising.

Consider that there are two algorithms to solve a certain problem and suppose that the number of operations required by these algorithms is $10^8 n$ and $10n^2$. The first function is big-O of n and the second is big-O of n^2. Using just the big-O information, the second algorithm is rejected because it grows too fast. It is true but, again, in the long run, since for $n \leq 10^7$, which is 10 million, the second algorithm performs fewer operations than the first. Although 10 million is not an unheard-of number of elements to be processed by an algorithm, in most cases the number is much lower, and in these cases the second algorithm is preferable.

For these reasons, it may be desirable to use one more notation that includes constants which are very large for practical reasons. Udi Manber proposes a double-O (OO) notation to indicate such functions: f is $OO(g(n))$ if it is big-O of $g(n)$ and the constant c is too large to have practical significance. Thus, $10^8 n$ is $OO(n)$. However, the definition of "too large" depends on the particular application.

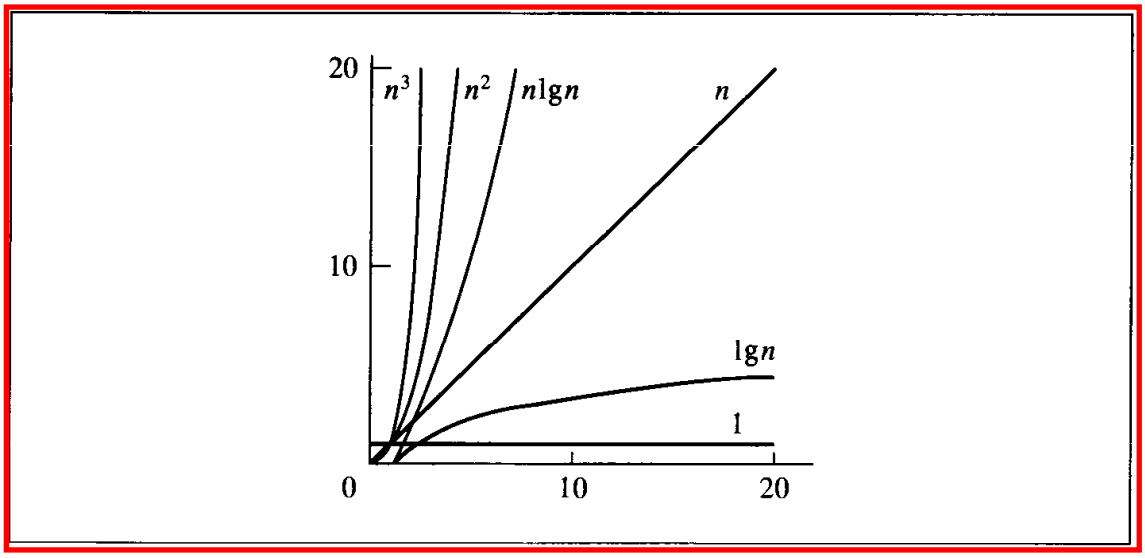
2.6 Examples of Complexities

Algorithms can be classified by their time or space complexities, and in this respect, several classes of such algorithms can be distinguished, as Figure 2.4 illustrates. Their growth is also displayed in Figure 2.5. For example, an algorithm is called *constant* if its execution time remains the same for any number of elements; it is called *quadratic* if its execution time is $O(n^2)$. For each of these classes, a number of operations is shown along with the real time needed for executing them on a machine able to perform 1 million operations per second, or one operation per microsecond (μsec). The table in Figure 2.4 indicates that some ill-designed algorithms, or algorithms whose complexity cannot be improved, have no practical application on available computers. To process 1 million items with a quadratic algorithm, over 11 days are needed,

FIGURE **2.4** **Classes of algorithms and their execution times on a computer executing 1 million operations per second (1 sec = 10^6 μsec = 10^3 msec).**

Class	Complexity	Number of Operations and Execution Time (1 instr/μsec)					
n		10		10^2		10^3	
constant	O(1)	1	1 μsec	1	1 μsec	1	1 μsec
logarithimic	O(lg n)	3.32	3 μsec	6.64	7 μsec	9.97	10 μsec
linear	O(n)	10	10 μsec	10^2	100 μsec	10^3	1 msec
O(n lg n)	O(n lg n)	33.2	33 μsec	664	664 μsec	9970	10 msec
quadratic	$O(n^2)$	10^2	100 μsec	10^4	10 msec	10^6	1 sec
cubic	$O(n^3)$	10^3	1 msec	10^6	1 sec	10^9	16.7 min
exponential	$O(2^n)$	1024	10 msec	10^{30}	$3.17 * 10^{17}$ yrs	10^{301}	
n		10^4		10^5		10^6	
constant	$O(1)$	1	1 μsec	1	1 μsec	1	1 μsec
logarithmic	$O(\lg n)$	13.3	13 μsec	16.6	7 μsec	19.93	20 μsec
linear	$O(n)$	10^4	10 msec	10^5	0.1 sec	10^6	1 sec
$O(n \lg n)$	$O(n \lg n)$	$133 * 10^3$	133 msec	$166 * 10^4$	1.6 sec	$199.3 * 10^5$	20 sec
quadratic	$O(n^2)$	10^8	1.7 min	10^{10}	16.7 min	10^{12}	11.6 days
cubic	$O(n^3)$	10^{12}	11.6 days	10^{15}	31.7 yr	10^{18}	31,709 yr
exponential	$O(2^n)$	10^{3010}		10^{30103}		10^{301030}	

FIGURE 2.5 Typical functions applied in big-O estimates.



and for a cubic algorithm, thousands of years. Even if a computer can perform one operation per nanosecond (1 billion operations per second), the quadratic algorithm finishes in only 16.7 seconds, but the cubic algorithm requires over 31 years. Even a 1000-fold improvement in execution speed has very little practical bearing for this algorithm. Analyzing the complexity of algorithms is of extreme importance and cannot be abandoned on account of the argument that we have entered an era when, at relatively little cost, a computer on our desktop can execute millions of operations per second. The importance of analyzing the complexity of algorithms, in any context but in the context of data structures in particular, cannot be overstressed. The impressive speed of computers is of limited use if the programs that run on them use inefficient algorithms.

2.7 Finding Asymptotic Complexity: Examples

Asymptotic bounds are used to estimate the efficiency of algorithms by assessing the amount of time and memory needed to accomplish the task for which the algorithms were designed. This section illustrates how this complexity can be determined.

In most cases, we are interested in time complexity, which usually measures the number of assignments and comparisons performed during the execution of a program. Chapter 9, which deals with sorting algorithms, considers both types of operations; this chapter considers only the number of assignment statements.

Begin with a simple loop to calculate the sum of numbers in an array:

```
for(i = sum = 0; i < n; i++)
    sum += a[i];
```

First, two variables are initialized, then the `for` loop iterates n times, and during each iteration, it executes two assignments, one of which updates `sum` and the other updates `i`. Thus, there are $2 + 2n$ assignments for the complete run of this `for` loop; its asymptotic complexity is $O(n)$.

Complexity usually grows if nested loops are used, as in the following code, which outputs the sums of all the subarrays that begin with position 0:

```
for(i = 0; i < n; i++) {
    for(j = 1, sum = a[0]; j <= i; j++)
        sum += a[j];
    cout<<"sum for subarray 0 through "<< i <<" is "<<sum<<endl;
}
```

Before the loops start, `i` is initialized. The outer loop is performed n times, executing in each iteration an inner `for` loop, print statement, and assignment statements for `i`, `j`, and `sum`. The inner loop is executed i times for each $i \in \{1, \ldots, n-1\}$ with two assignments in each iteration: one for `sum` and one for `j`. Therefore, there are $1 + 3n + \sum_{i=1}^{n-1} 2i = 1 + 3n + 2(1 + 2 + \cdots + n - 1) = 1 + 3n + n(n-1) = O(n) + O(n^2) = O(n^2)$ assignments executed before the program is completed.

For nested loops, complexity usually grows in comparison with one loop, but it does not have to grow at all. For example, we may request printing sums of numbers in the last five cells of the subarrays starting in position 0. We adopt the foregoing code and transform it to

```
for(i = 4; i < n; i++) {
    for(j = i-3, sum = a[i-4]; j <= i; j++)
        sum += a[j];
    cout<<"sum for subarray "<<i-4<<" through "<< i <<" is ""<<sum<<endl;
}
```

For each i, the inner loop is executed only four times; for each iteration of the outer loop, there are 11 assignments. This number does not depend on the size of the array. The program makes $1 + 11(n-4) = O(n)$ assignments, although the use of nested loops suggests something else.

Analysis of these two examples is relatively uncomplicated because the number of times the loops executed did not depend on the ordering of the arrays. Computation of asymptotic complexity is more involved if the number of iterations is not always the same. This point can be illustrated with a loop used to determine the length of the longest subarray with the numbers in increasing order. For example, in [1 8 1 2 5 0 11 12], it is three, the length of subarray [1 2 5]. The code is

```
for(i = 0, length = 1; i < n-1; i++) {
    for(i1 = i2 = k = i; k < n-1 && a[k] < a[k+1]; k++,i2++);
    if(length < i2 - i1 + 1)
      length = i2 - i1 + 1;
}
```

Notice that if all numbers in the array are in decreasing order, the outer loop is executed $n-1$ times, but in each iteration, the inner loop executes just one time. Thus, the

algorithm is $O(n)$. The algorithm is least efficient if the numbers are in increasing order. In this case, the outer `for` loop is executed $n-1$ times, and the inner loop is executed i times for each $i \in \{1, \ldots, n-1\}$. Thus, the algorithm is $O(n^2)$. In most cases, the arrangement of data is less orderly, and measuring the efficiency in these cases is of great importance. However, it is far from trivial to determine the efficiency in the average cases.

A fifth example used to determine the computational complexity is the *binary search algorithm*, which is used to locate an element in an ordered array. If it is an array of numbers and we try to locate number k, then the algorithm accesses the middle element of the array first. If that element is equal to k, then the algorithm returns its position; if not, the algorithm continues. In the second trial, only half of the original array is considered: the first half, if k is smaller than the middle element, and the second otherwise. Now, the middle element of the chosen subarray is accessed and compared to k. If it is the same, the algorithm completes successfully. Otherwise, the subarray is divided into two halves, and if k is larger than this middle element, the first half is discarded; otherwise, the first half is retained. This process of halving and comparing continues until k is found or the array can no longer be divided into two subarrays. This relatively simple algorithm can be coded as follows:

```
template<class genType>  // overloaded operator < is used;
int binarySearch(const genType arr[], int arrSize, const genType& key) {
    int lo = 0, mid, hi = arrSize-1;
    while (lo <= hi) {
          mid = (lo + hi)/2;
          if (key < arr[mid])
             hi = mid - 1;
          else if (arr[mid] < key)
             lo = mid + 1;
          else return mid;  // success: return the index of
    }                       //   the cell occupied by key;
    return -1;              // failure: key is not in the array;
}
```

If `key` is in the middle of the array, the loop executes only one time. How many times does the loop execute in the case where `key` is not in the array? First the algorithm looks at the entire array of size n, then at one of its halves of size $\frac{n}{2}$, then at one of the halves of this half, of size $\frac{n}{2^2}$, and so on, until the array is of size 1. Hence, we have the sequence $n, \frac{n}{2}, \frac{n}{2^2}, \ldots, \frac{n}{2^m}$, and we want to know the value of m. But the last term of this sequence $\frac{n}{2^m}$ equals 1, from which we have $m = \lg n$. So the fact that k is not in the array can be determined after $\lg n$ iterations of the loop.

2.8 The Best, Average, and Worst Cases

The last two examples in the preceding section indicate the need for distinguishing at least three cases for which the efficiency of algorithms has to be determined. The *worst case* is when an algorithm requires a maximum number of steps, and the *best case* is

when the number of steps is the smallest. The *average case* falls between these extremes. In simple cases, the average complexity is established by considering possible inputs to an algorithm, determining the number of steps performed by the algorithm for each input, adding the number of steps for all the inputs, and dividing by the number of inputs. This definition, however, assumes that the probability of occurrence of each input is the same, which is not always the case. To consider the probability explicitly, the average complexity is defined as the average over the number of steps executed when processing each input weighted by the probability of occurrence of this input, or,

$$C_{avg} = \sum_i p(input_i) steps(input_i)$$

This is the definition of expected value which assumes that all the possibilities can be determined and that the probability distribution is known, which simply determines a probability of occurrence of each input, $p(input_i)$. The probability function p satisfies two conditions: It is never negative, $p(input_i) \geq 0$, and all probabilities add up to 1, $\sum_i p(input_i) = 1$.

As an example, consider searching sequentially an unordered array to find a number. The best case is when the number is found in the first cell. The worst case is when the number is in the last cell or is not in the array at all. In this case, all the cells are checked to determine this fact. And the average case? We may make the assumption that there is an equal chance for the number to be found in any cell of the array; that is, the probability distribution is uniform. In this case, there is a probability equal to $\frac{1}{n}$ that the number is in the first cell, a probability equal to $\frac{1}{n}$ that it is in the second cell, . . . , and finally, a probability equal to $\frac{1}{n}$ that it is in the last, nth cell. This means that the probability of finding the number after one try equals $\frac{1}{n}$, the probability of finding it after two tries equals $\frac{1}{n}$, . . . , and the probability of finding it after n tries also equals $\frac{1}{n}$. Therefore, we can average all these possible numbers of tries over the number of possibilities and conclude that it takes on the average

$$\frac{1 + 2 + \ldots + n}{n} = \frac{n+1}{2}$$

steps to find a number. But if the probabilities differ, then the average case gives a different outcome. For example, if the probability of finding a number in the first cell equals $\frac{1}{2}$, the probability of finding it in the second cell equals $\frac{1}{4}$, and the probability of locating it in any of the remaining cells is the same and equal to

$$\frac{1 - \frac{1}{2} - \frac{1}{4}}{n-2} = \frac{1}{4(n-2)}$$

then, on the average, it takes

$$\frac{1}{2} + \frac{2}{4} + \frac{3 + \ldots n}{4(n-2)} = 1 + \frac{n(n+1)-6}{8(n-2)} = 1 + \frac{n+3}{8}$$

steps to find a number, which is approximately four times better than $\frac{n+1}{2}$ found previously for the uniform distribution. Note that the probabilities of accessing a particular cell have no impact on the best and last cases.

The complexity for the three cases was relatively easy to determine for sequential search, but usually it is not that straightforward. Particularly, the complexity of the average case can pose difficult computational problems. If the computation is very complex, approximations are used, and that is where we find the big-O, Ω, and Θ notations most useful.

As an example, consider the average case for binary search. Assume that the size of the array is a power of 2 and that a number to be searched has an equal chance to be in any of the cells of the array. Binary search can locate it either after one try in the middle of the array, or after two tries in the middle of the first half of the array, or after two tries in the middle of the second half, or after three tries in the middle of the first quarter of the array, or . . . or after three tries in the middle of the fourth quarter, or after four tries in the middle of the first eighth of the array, or . . . or after four tries in the middle of the eighth eighth of the array, or . . . or after try $\lg n$ in the first cell, or after try $\lg n$ in the third cell, or . . . or, finally, after try $\lg n$ in the last cell. That is, the number of all possible tries equals

$$1\cdot 1+2\cdot 2+4\cdot 3+8\cdot 4+\ldots+\frac{n}{2}\lg n=\sum_{i=0}^{\lg n-1}2^i(i+1)$$

which has to be divided by $\frac{1}{n}$ to determine the average case complexity. What is this sum equal to? We know that it is between 1 (the best case result) and $\lg n$ (the worst case) determined in the preceding section. But is it closer to the best case—say, $\lg\lg n$—or to the worst case—for instance, $\frac{\lg n}{2}$, or $\lg\frac{n}{2}$? The sum does not lend itself to a simple conversion into a closed form; therefore, its estimation should be used. Our conjecture is that the sum is not less than the sum of powers of 2 in the specified range multiplied by a half of $\lg n$, that is,

$$s_1=\sum_{i=0}^{\lg n-1}2^i(i+1)\geq\frac{\lg n}{2}\sum_{i=0}^{\lg n-1}2^i=s_2$$

The reason for this choice is that s_2 is a power series multiplied by a constant factor, and thus it can be presented in closed form very easily, namely,

$$s_2=\frac{\lg n}{2}\sum_{i=0}^{\lg n-1}2^i=\frac{\lg n}{2}\left(1+2\frac{2^{\lg n-1}-1}{2-1}\right)=\frac{\lg n}{2}(n-1)$$

which is $\Omega(n\lg n)$. Because s_2 is the lower bound for the sum s_1 under scrutiny—that is, s_1 is $\Omega(s_2)$—then so is $\frac{s_2}{n}$, the lower bound of the sought average case complexity $\frac{s_1}{n}$—that is, $\frac{s_1}{n}=\Omega(\frac{s_2}{n})$. Because $\frac{s_2}{n}$ is $\Omega(\lg n)$, so must be $\frac{s_1}{n}$. Because $\lg n$ is an assessment of the complexity of the worst case, the average case's complexity equals $\Theta(\lg n)$.

There is still one unresolved problem: Is $s_1\geq s_2$? To determine this, we conjecture that the sum of each pair of terms positioned symmetrically with respect to the center of the sum s_1 is not less than the sum of the corresponding terms of s_2. That is,

$$2^0 \cdot 1 + 2^{\lg n-1} \lg n \geq 2^0 \frac{\lg n}{2} + 2^{\lg n-1} \frac{\lg n}{2}$$

$$2^1 \cdot 2 + 2^{\lg n-2}(\lg n - 1) \geq 2^1 \frac{\lg n}{2} + 2^{\lg n-2} \frac{\lg n}{2}$$

...

$$2^j(j+1) + 2^{\lg n-1-j}(\lg n - j) \geq 2^j \frac{\lg n}{2} + 2^{\lg n-1-j} \frac{\lg n}{2}$$

..

where $j \leq \frac{\lg n}{2} - 1$. The last inequality, which represents every other inequality, is transformed into

$$2^{\lg n-1-j}\left(\frac{\lg n}{2} - j\right) \geq 2^j\left(\frac{\lg n}{2} - j - 1\right)$$

and then into

$$2^{\lg n-1-2j} \geq \frac{\frac{\lg n}{2} - j - 1}{\frac{\lg n}{2} - j} = 1 - \frac{1}{\frac{\lg n}{2} - j} \tag{2.5}$$

All of these transformations are allowed because all the terms that moved from one side of the conjectured inequality to another are nonnegative and thus do not change the direction of inequality. Is the inequality true? Because $j \leq \frac{\lg n}{2} - 1$, $2^{\lg n-1-2j} \geq 2$, and the right-hand side of the inequality (2.5) is always less than 1, the conjectured inequality is true.

This concludes our investigation of the average case for binary search. The algorithm is relatively straightforward, but the process of finding the complexity for the average case is rather grueling, even for uniform probability distributions. For more complex algorithms, such calculations are significantly more challenging.

2.9 Amortized Complexity

In many situations, data structures are subject to a sequence of operations rather than one operation. In this sequence, one operation possibly performs certain modifications that have an impact on the run time of the next operation in the sequence. One way of assessing the worst case run time of the entire sequence is to add worst case efficiencies for each operation. But this may result in an excessively large and unrealistic bound on the actual run-time. To be more realistic, amortized analysis can be used to find the average complexity of a worst case sequence of operations. By analyzing sequences of operations rather than isolated operations, amortized analysis takes into account interdependence between operations and their results. For example, if an array is sorted and only a very few new elements are added, then resorting this array should be much faster than sorting it for the first time because,

after the new additions, the array is nearly sorted. Thus, it should be quicker to put all elements in perfect order than in a completely disorganized array. Without taking this correlation into account, the run time of the two sorting operations can be considered twice the worst case efficiency. Amortized analysis, on the other hand, decides that the second sorting is hardly applied in the worst case situation so that the combined complexity of the two sorting operations is much less than double the worst case complexity. Consequently, the average for the worst case sequence of sorting, a few insertions, and sorting again is lower according to amortized analysis that according to worst case analysis, which disregards the fact that the second sorting is applied to an array operated on already by a previous sorting.

It is important to stress that amortized analysis is analyzing sequences of operations, or if single operations are analyzed, it is done in view of their being part of the sequence. The cost of operations in the sequence may vary considerably, but how frequently particular operations occur in the sequence is important. For example, for the sequence of operations $op_1, op_2, op_3, \ldots$, the worst case analysis renders the computational complexity for the entire sequence equal to

$$C(op_1, op_2, op_3, \ldots) = C_{worst}(op_1) + C_{worst}(op_2) + C_{worst}(op_3) + \ldots$$

whereas the average complexity determines it to be

$$C(op_1, op_2, op_3, \ldots) = C_{avg}(op_1) + C_{avg}(op_2) + C_{avg}(op_3) + \ldots$$

Although specifying complexities for a sequence of operations, neither worst case analysis nor average case analysis was looking at the position of a particular operation in the sequence. These two analyses considered the operations as executed in isolation and the sequence as a collection of isolated and independent operations. Amortized analysis changes the perspective by looking at what happened up until a particular point in the sequence of operations and then determines the complexity of a particular operation,

$$C(op_1, op_2, op_3, \ldots) = C(op_1) + C(op_2) + C(op_3) + \ldots$$

where C can be the worst, the average, the best case complexity, or very likely, a complexity other than the three depending on what happened before. To find amortized complexity in this way may be, however, too complicated. Therefore, another approach is used. The knowledge of the nature of particular processes and possible changes of a data structure is used to determine the function C which can be applied to each operation of the sequence. The function is chosen in such a manner that it considers quick operations as slower than they really are and time-consuming operations as quicker than they actually are. It is as though the cheap (quick) operations are charged more time units to generate credit to be used for covering the cost of expensive operations that are charged below their real cost. It is like letting the government charge us more for social security than necessary so that at the end of the fiscal year the overpayment can be received back and used to cover the expenses of something else. The art of amortized analysis lies in finding an appropriate function C so that it overcharges cheap operations sufficiently to cover expenses of undercharged operations. The overall balance must be nonnegative. If a debt occurs, there must be a prospect of paying it.

Consider the operation of adding a new element to the vector implemented as a flexible array. The best case is when the size of the vector is less than its capacity because adding a new element amounts to putting it in the first available cell. The cost of adding a new element is thus $O(1)$. The worst case is when size equals capacity, in which case there is no room for new elements. In this case, new space must be allocated, the existing elements are copied to the new space, and only then can the new element be added to the vector. The cost of adding a new element is $O(\text{size}(vector))$. It is clear that the latter situation is less frequent than the former, but this depends on another parameter, capacity increment, which refers to how much the vector is increased when overflow occurs. In the extreme case, it can be incremented by just one cell, so in the sequence of m consecutive insertions, each insertion causes overflow and requires $O(size(vector))$ time to finish. Clearly, this situation should be delayed. One solution is to allocate, say, 1 million cells for the vector, which in most cases does not cause an overflow, but the amount of space is excessively large and only a small percentage of space allocated for the vector may be expected to be in actual use. Another solution to the problem is to double the space allocated for the vector if overflow occurs. In this case, the pessimistic $O(size(vector))$ performance of the insertion operation may be expected to occur only infrequently. By using this estimate, it may be claimed that, in the best case, the cost of inserting m items is $O(m)$, but it is impossible to claim that, in the worst case, it is $O(m \cdot size(vector))$. Therefore, to see better what impact this performance has on the sequence of operations, the amortized analysis should be used.

In amortized analysis, the question is asked: What is the expected efficiency of a sequence of insertions? We know that the best case is $O(1)$ and the worst case is $O(size(vector))$, but also we know that the latter case occurs only occasionally and leads to doubling the size of the vector. In which case, what is the expected efficiency of one insertion in the series of insertions? Note that we are interested only in sequences of insertions, excluding deletions and modifications, to have the worst case scenario. The outcome of amortized analysis depends on the assumed amortized cost of one insertion. It is clear that if

$$amCost(push(x)) = 1$$

where 1 represents the cost of one insertion, then we are not gaining anything from this analysis because easy insertions are paying for themselves right away, and the insertions causing overflow and thus copying have no credit to use to make up for their high cost. Is

$$amCost(push(x)) = 2$$

a reasonable choice? Consider the table in Figure 2.6a. It shows the change in vector capacity and the cost of insertion when size grows from 0 to 18; that is, the table indicates the changes in the vector during the sequence of 18 insertions into an initially empty vector. For example, if there are four elements in the vector (size = 4), then before inserting the fifth element, the four elements are copied at the cost of four units and then the new fifth element is inserted in the newly allocated space for the vector. Hence, the cost of the fifth insertion is 4 + 1. But to execute this insertion, two units allocated for the fifth insertion are available plus one unit left from the previous

fourth insertion. This means that this operation is two units short to pay for itself. Thus, in the units left column, – 2 is entered to indicate the debt of two units. The table indicates that the debt decreases and becomes zero, one cheap insertion away from the next expensive insertion. This means that the operations are almost constantly executed in the red, and more important, if a sequence of operations finishes before the debt is paid off, then the balance indicated by amortized analysis is negative, which is inadmissible in the case of algorithm analysis. Therefore, the next best solution it to assume that

$$amCost(push(x)) = 3$$

The table in Figure 2.6b indicates that we are never in debt and that the choice of three units for amortized cost is not excessive because right after an expensive insertion, the accumulated units are almost depleted.

In this example, the choice of a constant function for amortized cost is adequate, but usually it is not. Define as *potential* a function that assigns a number to a particu-

FIGURE **2.6**

(a)

size	capacity	amortized cost	cost	units left
0	0			
1	1	2	0 + 1	1
2	2	2	1 + 1	1
3	4	2	2 + 1	0
4	4	2	1	1
5	8	2	4 + 1	–2
6	8	2	1	–1
7	8	2	1	0
8	8	2	1	1
9	16	2	8 + 1	–6
10	16	2	1	–5
:	:	:	:	:
16	16	2	1	1
17	32	2	16 + 1	–14
18	32	2	1	–13
:	:	:	:	:

(b)

size	capacity	amortized cost	cost	units left
0	0			
1	1	3	0 + 1	2
2	2	3	1 + 1	3
3	4	3	2 + 1	3
4	4	3	1	5
5	8	3	4 + 1	3
6	8	3	1	5
7	8	3	1	7
8	8	3	1	9
9	16	3	8 + 1	3
10	16	3	1	5
:	:	:	:	:
16	16	3	1	17
17	32	3	16 + 1	3
18	32	3	1	5
:	:	:	:	:

lar state of a data structure *ds* that is a subject of a sequence of operations. The amortized cost is defined as a function

$$amCost(op_i) = cost(op_i) + potential(ds_i) - potential(ds_{i-1})$$

which is the real cost of executing the operation op_i plus the change in potential in the data structure *ds* as a result of execution of op_i. This definition holds for one single operation of a sequence of *m* operations. If amortized costs for all the operations are added, then the amortized cost for the sequence

$$amCost(op_1, \ldots, op_m) = \sum_{i=1}^{m} (amCost(op_i) + potential(ds_i) - potential(ds_{i-1}))$$

$$= \sum_{i=1}^{m} amCost(op_i) + potential(ds_m) - potential(ds_0)$$

In most cases, the potential function is initially zero and is always nonnegative so that amortized time is an upper bound of real time. This form of amortized cost is used later in the book.

Amortized cost of including new elements in a vector can now be phrased in terms of the new form for the function *amCost*:

$$amCost(push_i) = \begin{cases} 0 & \text{if } size_i = capacity_i \text{ (vector is full)} \\ 2\text{size}_i - \text{capacity}_i & \text{otherwise} \end{cases}$$

To see that the function works as intended, consider three cases. The first case is when a cheap pushing follows cheap pushing (vector is not extended right before the current push and is not extended as a consequence of the current push) and

$$amCost(push_i()) = 1 + 2size_{i-1} + 2 - capacity_{i-1} - 2size_{i-1} + capacity_i = 3$$

because the capacity does not change, $size_i = size_{i-1} + 1$, and the actual cost equals 1. For expensive pushing following cheap pushing,

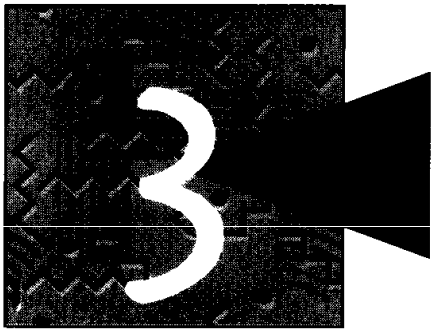
$$amCost(push_i()) = size_{i-1} + 2 + 0 - 2size_{i-1} + capacity_{i-1} = 3$$

because $size_{i-1} + 1 = capacity_{i-1}$ and the actual cost equals $size_i + 1 = size_{i-1} + 2$, which is the cost of copying the vector elements plus adding the new element. For cheap pushing following expensive pushing,

$$amCost(push_i()) = 1 + 2size_i - capacity_i - 0 = 3$$

because $2(size_i - 1) = capacity_i$ and actual cost equals 1. Note that the fourth case, expensive pushing following expensive pushing, occurs only twice, when capacity changes from zero to one and from one to zero. In both cases, amortized cost equals 3.

Linked Lists



An array is a very useful data structure provided in programming languages. However, it has at least two limitations: (1) its size has to be known at compilation time and (2) the data in the array are separated in computer memory by the same distance, which means that inserting an item inside the array requires shifting other data in this array. This limitation can be overcome by using *linked structures*. A linked structure is a collection of nodes storing data and links to other nodes. In this way, nodes can be located anywhere in memory, and passing from one node of the linked structure to another is accomplished by storing the addresses of other nodes in the linked structure. Although linked structures can be implemented in a variety of ways, the most flexible implementation is by using pointers.

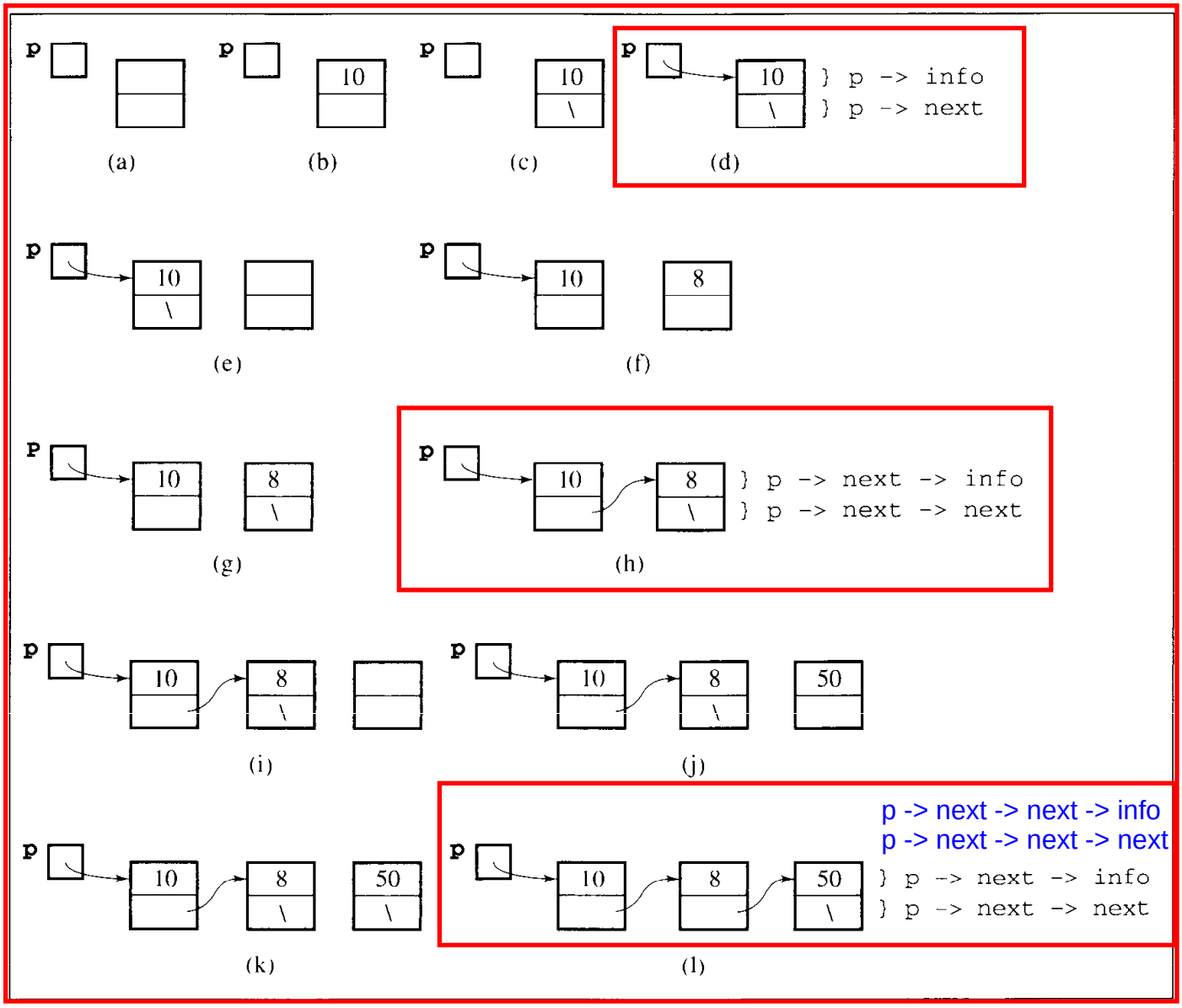
3.1 Singly Linked Lists

If a node contains a data member that is a pointer to another node, then many nodes can be strung together using only one variable to access the entire sequence of nodes. Such a sequence of nodes is the most frequently used implementation of a *linked list*, which is a data structure composed of nodes, each node holding some information and a pointer to another node in the list. If a node has a link only to its successor in this sequence, the list is called a *singly linked list*. An example of such a list is shown in Figure 3.1. Note that only one variable `p` is used to access any node in the list. The last node on the list can be recognized by the null pointer.

Each node in the list in Figure 3.1 is an instance of the following class definition:

```
class IntNode {
public:
    IntNode() {
```

FIGURE 3.1 A singly linked list.




```
        next = 0;
    }
    IntNode(int i, IntNode *in = 0) {
        info = i; next = in;
    }
    int info;
    IntNode *next;
};
```

A node includes two data members: `info` and `next`. The `info` member is used to store information, and this member is important to the user. The `next` member is

used to link together nodes to form a linked list. It is an auxiliary data member used to maintain the linked list. It is indispensable for implementation of the linked list but less important (if at all) from the user's perspective. Note that `IntNode` is defined in terms of itself because one data member, `next`, is a pointer to a node of the same type that is just being defined. This circularity, however, is permitted in C++.

The definition of a node also includes two constructors. The first constructor initializes the `next` pointer to null and leaves the value of `info` unspecified. The second constructor takes two arguments, one to initialize the `info` member and another to initialize the `next` member. The second constructor also covers the case when only one numerical argument is supplied by the user. In this case, `info` is initialized to the argument and `next` to null.

Now, let us create the linked list in Figure 3.11. One way to create this three-node linked list is to first generate the node containing number 10, then the node containing 8, and finally the node containing 50. Each node has to be initialized properly and incorporated into the list. To see this, each step is illustrated in Figure 3.1 separately. First, we execute the declaration and assignment

```
IntNode *p = new IntNode(10);
```

which creates the first node on the list and makes the variable `p` a pointer to this node. This is done in four steps. In the first step, a new `IntNode` is created (Figure 3.1a), in the second step, the `info` member of this node is set to 10 (Figure 3.1b), and in the third step, the node's `next` member is set to `null` (Figure 3.1c). The null pointer is marked with a slash in the pointer data member. Note that the slash in the `next` member is not a slash character. That is, the second and third steps—initialization of data members of the new `IntNode`—are performed by invoking the constructor `IntNode(10)`, which turns into the constructor `IntNode(10,0)`. The fourth step is making `p` a pointer to the newly created node (Figure 3.1d). This pointer is the address of the node, and it is shown as an arrow from the variable `p` to the new node.

The second node is created with the assignment

```
p->next = new IntNode(8);
```

where `p->next` is the `next` member of the node pointed by `p` (Figure 3.1d). As before, four steps are executed:

1. creating a new node (Figure 3.1e),
2. assigning by the constructor number 8 to the `info` member of this node (Figure 3.1f) and
3. `null` to its `next` member (Figure 3.1g), and finally
4. including the new node in the list by making the `next` member of the first node a pointer to the new node (Figure 3.1h).

Note that the data members of nodes pointed to by `p` are accessed using the arrow notation, which is clearer than using a dot notation, as in `(*p).next`.

The linked list is now extended by adding a third node with the assignment

```
p->next->next = new IntNode(50);
```

where `p->next->next` is the `next` member of the second node. This cumbersome notation has to be used because the list is accessible only through the variable `p`.

In processing the third node, four steps are also executed: creating the node (Figure 3.1i), initializing its two data members (Figure 3.1j–k), and then incorporating the node in the list (Figure 3.1l).

Our linked list example illustrates a certain inconvenience in using pointers: The longer the linked list, the longer the chain of `next`s to access the nodes at the end of the list. In this example, `p->next->next->next` allows us to access the `next` member of the 3rd node on the list. But what if it were the 103rd or, worse, the 1003rd node on the list? Typing 1003 `next`s, as in `p->next->...->next`, would be daunting. If we missed one `next` in this chain, then a wrong assignment is made. Also, the flexibility of using linked lists is diminished. Therefore, other ways of accessing nodes in linked lists are needed. One way is always to keep two pointers to the linked list: one to the first node and one to the last, as shown in Figure 3.2.

FIGURE 3.2 An implementation of a singly linked list of integers.

```
//************************  intSLLst.h  ***************************
//           singly-linked list class to store integers

#ifndef INT_LINKED_LIST
#define INT_LINKED_LIST

class IntNode {
public:
    int info;
    IntNode *next;
    IntNode(int el, IntNode *ptr = 0) {
        info = el; next = ptr;
    }
};

class IntSLList {
public:
    IntSLList() {
        head = tail = 0;
    }
    ~IntSLList();
    int isEmpty() {
        return head == 0;
    }
    void addToHead(int);
```

FIGURE 3.2 *(continued)*

```
    void addToTail(int);
    int  deleteFromHead(); // delete the head and return its info;
    int  deleteFromTail(); // delete the tail and return its info;
    void deleteNode(int);
    bool isInList(int) const;
private:
    IntNode *head, *tail;
};

#endif

//************************  intSLLst.cpp ***************************

#include <iostream.h>
#include "intSLLst.h"

IntSLList::~IntSLList() {
    for (IntNode *p; !isEmpty(); ) {
        p = head->next;
        delete head;
        head = p;
    }
}
void IntSLList::addToHead(int el) {
    head = new IntNode(el,head);
    if (tail == 0)
       tail = head;
}
void IntSLList::addToTail(int el) {
    if (tail != 0) {    // if list not empty;
         tail->next = new IntNode(el);
         tail = tail->next;
    }
    else head = tail = new IntNode(el);
}
int IntSLList::deleteFromHead() {
    int el = head->info;
    IntNode *tmp = head;
    if (head == tail)   // if only one node in the list;
         head = tail = 0;
```

Continues

FIGURE 3.2 *(continued)*

```
    else head = head->next;
    delete tmp;
    return el;
}
int IntSLList::deleteFromTail() {
    int el = tail->info;
    if (head == tail) { // if only one node in the list;
        delete head;
        head = tail = 0;
    }
    else {              // if more than one node in the list,
        IntNode *tmp;  // find the predecessor of tail;
        for (tmp = head; tmp->next != tail; tmp = tmp->next);
        delete tail;
        tail = tmp;    // the predecessor of tail becomes tail;
        tail->next = 0;
    }
    return el;
}
void IntSLList::deleteNode(int el) {
    if (head != 0)                    // if non-empty list;
        if (head == tail && el == head->info) { // if only one
            delete head;                        // node in the list;
            head = tail = 0;
        }
        else if (el == head->info) {// if more than one node in the list
            IntNode *tmp = head->next;
            head = head->next;
            delete tmp;            // and old head is deleted;
        }
        else {                     // if more than one node in the list
            IntNode *pred, *tmp;
            for (pred = head, tmp = head->next; // and a non-head node
                 tmp != 0 && !(tmp->info == el);// is deleted;
                 pred = pred->next, tmp = tmp->next);
            if (tmp != 0) {
                pred->next = tmp->next;
                if (tmp == tail)
                   tail = pred;
                delete tmp;
            }
```

FIGURE 3.2 *(continued)*

```
        }
}
bool IntSLList::isInList(int el) const {
    IntNode *tmp;
    for (tmp = head; tmp != 0 && !(tmp->info == el); tmp = tmp->next);
    return tmp != 0;
}
```

A singly linked list implementation in Figure 3.2 uses two classes: one class, `IntNode`, for nodes of the list, and another, `IntSLList`, for access to the list. The class `IntSLList` defines two data members, `head` and `tail`, which are pointers to the first and the last nodes of a list. This explains why all members of `IntNode` are declared public. Because particular nodes of the list are accessible through pointers, nodes are made inaccessible to outside objects by declaring `head` and `tail` private so that the information-hiding principle is not really compromised. If some of the members of `IntNode` were declared nonpublic, then classes derived from `IntSLList` could not access them.

An example of a list is shown in Figure 3.3. The list is declared with the statement

```
IntSLList list;
```

The first object in Figure 3.3a is not part of the list; it allows for having access to the list. For simplicity, in subsequent figures, only nodes belonging to the list are shown, the access node is omitted, and the `head` and `tail` members are marked as in Figure 3.3b.

Besides the `head` and `tail` members, the class `IntSLList` also defines member functions that allow us to manipulate the lists. We now look more closely at some basic operations on linked lists presented in Figure 3.2.

3.1.1 Insertion

Adding a node at the beginning of a linked list is performed in four steps.

1. An empty node is created. It is empty in the sense that the program performing insertion does not assign any values to the data members of the node (Figure 3.4a).
2. The node's `info` member is initialized to a particular integer (Figure 3.4b).
3. Because the node is being included at the front of the list, the `next` member becomes a pointer to the first node on the list, that is, the current value of `head` (Figure 3.4c).
4. The new node precedes all the nodes on the list, but this fact has to be reflected in the value of `head`; otherwise, the new node is not accessible. Therefore, `head` is updated to become the pointer to the new node (Figure 3.4d).

FIGURE 3.3 A singly linked list of integers.

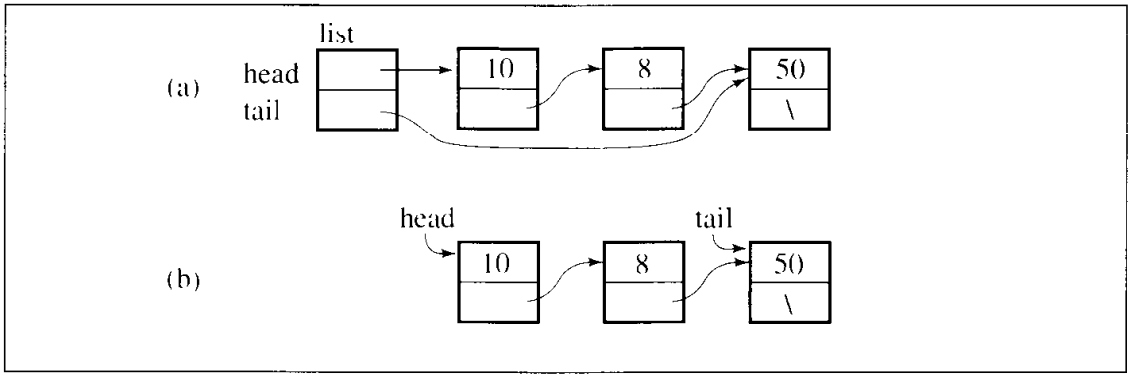
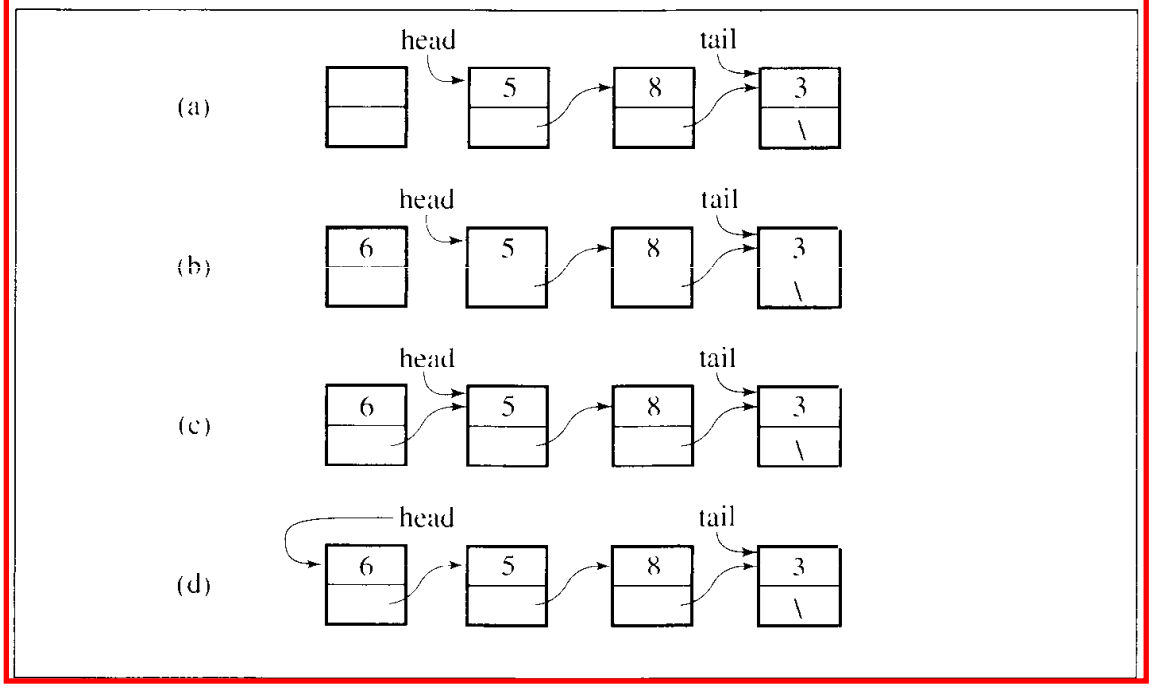


FIGURE 3.4 Inserting a new node at the beginning of a singly linked list.

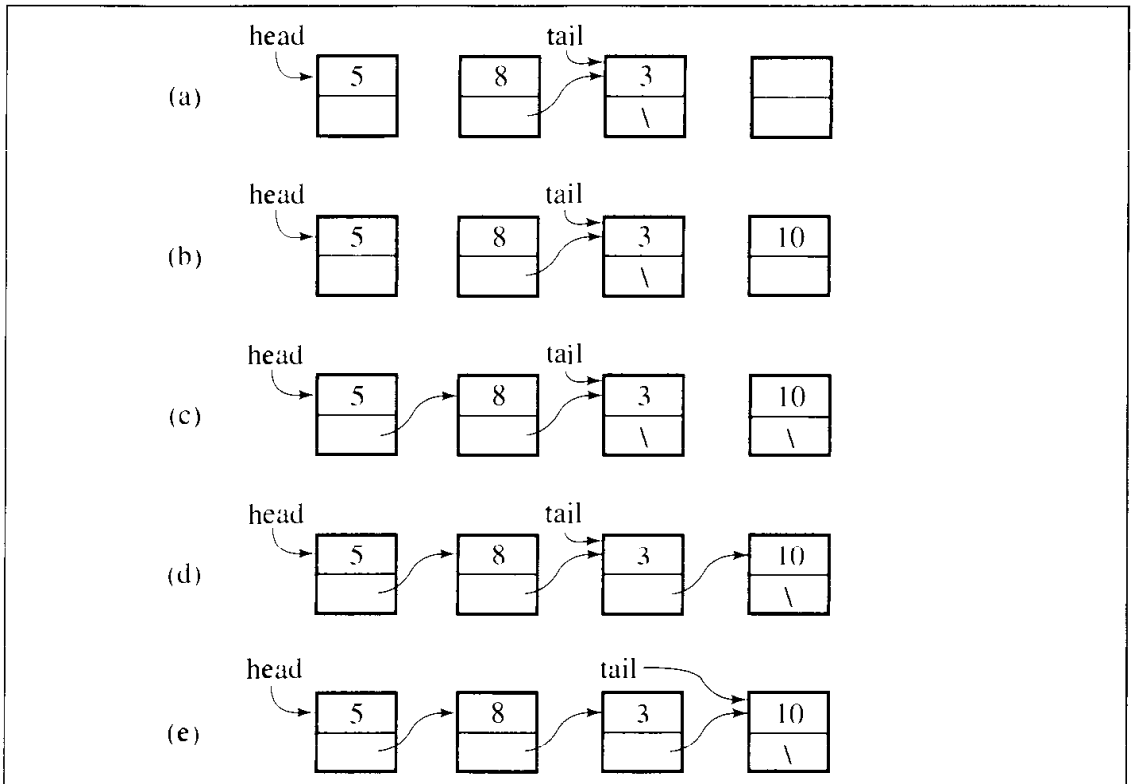


The four steps are executed by the member function `addToHead()` (Figure 3.2). The function executes the first three steps indirectly by calling the constructor `IntNode(el,head)`. The last step is executed directly in the function by assigning the address of the newly created node to `head`.

The member function `addToHead()` singles out one special case, namely, inserting a new node in an empty linked list. In an empty linked list, both `head` and `tail` are null; therefore, both become pointers to the only node of the new list. When inserting in a nonempty list, only `head` needs to be updated.

The process of adding a new node to the end of the list has five steps.

FIGURE 3.5 Inserting a new node at the end of a singly linked list.

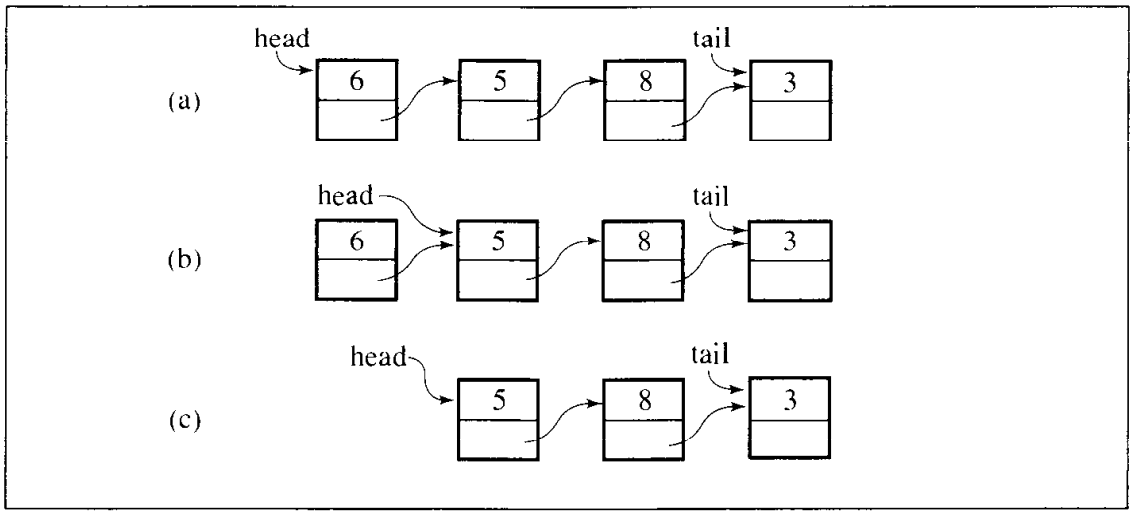


1. An empty node is created (Figure 3.5a).
2. The node's `info` member is initialized to an integer `el` (Figure 3.5b).
3. Because the node is being included at the end of the list, the `next` member is set to null (Figure 3.5c).
4. The node is now included in the list by making the `next` member of the last node of the list a pointer to the newly created node (Figure 3.5d).
5. The new node follows all the nodes of the list, but this fact has to be reflected in the value of `tail` which now becomes the pointer to the new node (Figure 3.5e).

All these steps are executed in the `if` clause of `addToTail()` (Figure 3.2). The `else` clause of this function is executed only if the linked list is empty. If this case were not included, the program may crash because in the `if` clause we make an assignment to the `next` member of the node referred by `tail`. In the case of an empty linked list, it is a pointer to a nonexisting data member of a nonexisting node.

The process of inserting a new node at the beginning of the list is very similar to the process of inserting a node at the end of the list. This is so because the implementation of `IntSLList` uses two pointer members: `head` and `tail`. For this reason, both `addToHead()` and `addToTail()` can be executed in constant time $O(1)$; that is, regardless of the number of nodes in the list, the number of operations performed

FIGURE 3.6
Deleting a node at the beginning of a singly linked list.



by these two member functions does not exceed some constant number c. Note that because the `head` pointer allows us to have access to a linked list, the `tail` pointer is not indispensable; its only role is to have immediate access to the last node of the list. With this access, a new node can be easily added at the end of the list. If the `tail` pointer were not used, then adding a node at the end of the list would be more complicated because we would first have to reach the last node in order to attach a new node to it. This requires scanning the list and requires $O(n)$ steps to finish; that is, it is linearly proportional to the length of the list. The process of scanning lists is illustrated when discussing deletion of the last node.

3.1.2 Deletion

One deletion operation consists in deleting a node at the beginning of the list and returning the value stored in it. This operation is implemented by the member function `deleteFromHead()`. In this operation, the information from the first node is temporarily stored in a local variable `el`, and then `head` is reset so that what was the second node becomes the first node. In this way, the former first node can be deleted in constant time $O(1)$ (Figure 3.6).

Unlike before, there are now two special cases to consider. One case is when we attempt to remove a node from an empty linked list. If such an attempt is made, the program is very likely to crash, which we don't want to happen. The caller should also know that such an attempt is made to perform certain action. After all, if the caller expects a number to be returned from the call to `deleteFromHead()` and no number can be returned, then the caller may be unable to accomplish some other operations.

There are several ways to approach this problem. One way is to use an `assert` statement:

```
int IntSLList::deleteFromHead() {
    assert(!isEmpty()); // terminate the program if false;
    int el = head->info;
    . . . . . .
    return el;
}
```

The `assert` statement checks the condition `!isEmpty()`, and if the condition is false, the program is aborted. This is a crude solution because from the perspective of the caller the program can be continued even if no number is returned from `deleteFromHead()`.

Another solution is to throw an exception and catch it by the user as in:

```
int IntSLList::deleteFromHead() {
    if (isEmpty())
        throw("Empty");
    int el = head->info;
    . . . . . .
    return el;
}
```

The `throw` clause with the string argument is expected to have a matching `try-catch` clause in the caller (or caller's caller, etc.) also with the string argument which catches the exception as in

```
void f() {
    . . . . . .
    try {
        n = list.deleteFromHead();
        // do something with n;
    } catch(char *s) {
        cerr << "Error: " << s << endl;
    }
    . . . . . .
}
```

This solution gives the caller some control over the abnormal situation without making it lethal to the program as with the use of the `assert` statement. The user is responsible for providing an exception handler in the form of the `try-catch` clause, with the solution appropriate to the particular case. If the clause is not provided, then the program crashes when the exception is thrown. The function `f()` may only print a message that a list is empty when an attempt is made to delete a number from an empty list, another function `g()` may assign a certain value to n in such a case, and yet another function `h()` may find such a situation detrimental to the program and abort the program altogether.

The idea that the user is responsible for providing an action in the case of an exception is also presumed in the implementation given in Figure 3.2. The member function assumes that the list is not empty. To prevent the program from crashing, the

member function `isEmpty()` is added to the `IntSLList` class, and the user should use it as in:

```
if (!list.isEmpty())
     n = list.deleteFromHead();
else do not remove;
```

Note that including a similar `if` statement in `deleteFromHead()` does not solve the problem. Consider this code:

```
int IntSLList::deleteFromHead() {
    if (!isEmpty()) {          // if non-empty list;
         int el = head->info;
         . . . . . .
         return el;
    }
    else return 0;
}
```

If an `if` statement is added, then the `else` clause must also be added; otherwise, the program does not compile because "not all control paths return a value." But now, if 0 is returned, the caller does not know whether the returned 0 is the sign of failure or if it is a literal 0 retrieved from the list. To avoid any confusion, the caller must use an `if` statement to test whether the list is empty before calling `deleteFromHead()`. In this way, one `if` statement would be redundant.

To maintain uniformity in the interpretation of the return value, the last solution can be modified so that instead of returning an integer, the function returns the pointer to an integer:

```
int* IntSLList::deleteFromHead() {
    if (!isEmpty()) {          // if non-empty list;
         int *el = new int(head->info);
         . . . . . .
         return el;
    }
    else return 0;
}
```

where 0 in the `else` clause is the null pointer, not the number 0. In this case, the function call

```
n = *list.deleteFromHead();
```

results in a program crash if `deleteFromHead()` returns the null pointer.

Therefore, a test must be performed by the caller before calling `deleteFromHead()` to check whether `list` is empty or a pointer variable has to be used,

```
int *p = list.deleteFromHead();
```

and then a test is performed after the call to check whether `p` is null or not. In either case, this means that the `if` statement in `deleteFromHead()` is redundant.

The second special case is when the list has only one node to be removed. In this case, the list becomes empty, which requires setting `tail` and `head` to null.

The second deletion operation consists in deleting a node from the end of the list, and it is implemented as the member function `deleteFromTail()`. The problem is that after removing a node, `tail` should refer to the new tail of the list; that is, `tail` has to be moved backward by one node. But moving backward is impossible because there is no direct link from the last node to its predecessor. Hence, this predecessor has to be found by searching from the beginning of the list and stopping right before `tail`. This is accomplished with a temporary variable `tmp` used to scan the list within the `for` loop. The variable `tmp` is initialized to the head of the list, and then in each iteration of the loop, it is advanced to the next node. If the list is as in Figure 3.7a, then `tmp` first refers the head node holding number 6; after executing the assignment `tmp = tmp->next`, `tmp` refers to the second node (Figure 3.7b). After the second iteration and executing the same assignment, `tmp` refers to the third node (Figure 3.7c). Because this node is also the next to last node, the loop is exited, after which the last node is deleted (Figure 3.7d). Because `tail` is now pointing to a nonexisting node, it is immediately set to point to the next to last node currently pointed to by `tmp` (Figure 3.7e). To mark the fact that it is the last node of the list, the `next` member of this node is set to null (Figure 3.7f).

Note that in the `for` loop, a temporary variable is used to scan the list. If the loop were simplified to

```
for ( ; head->next != tail; head = head->next);
```

then the list is scanned only once, and the access to the beginning of the list is lost because `head` was permanently updated to the next to last node, which is about to become the last node. It is absolutely critical that, in cases such as this, a temporary variable is used so that the access to the beginning of the list is kept intact.

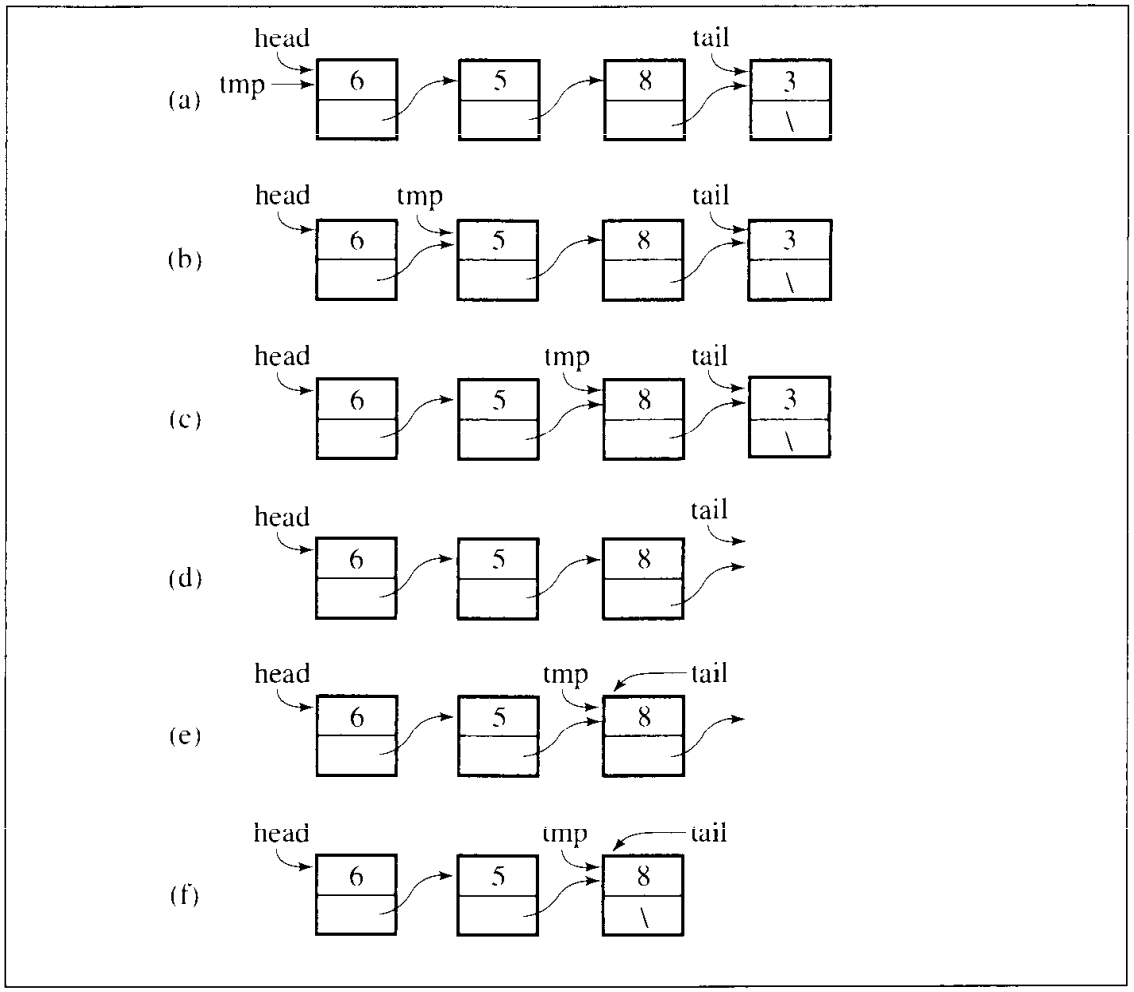
In removing the last node, the two special cases are the same as in `deleteFromHead()`. If the list is empty, then nothing can be removed, but what should be done in this case is decided in the user program just as in the case of `deleteFromHead()`. The second case is when a single-node list becomes empty after removing its only node, which also requires setting `head` and `tail` to null.

The most time-consuming part in `deleteFromTail()` is finding the next to last node performed by the `for` loop. It is clear that the loop performs $n-1$ iterations in a list of n nodes, which is the main reason this member function takes $O(n)$ time to delete the last node.

The two discussed deletion operations remove a node from the head or from the tail (that is, always from the same position) and return an integer that happens to be in the node being removed. A different approach is when we want to delete a node that holds a particular integer regardless of the position of this node in the list. It may be right at the beginning, at the end, or anywhere inside the list. Briefly, a node has to be located first and then detached from the list by linking the predecessor of this node directly to its successor. Because we do not know where the node may be, the process of finding and deleting a node with a certain integer is much more complex than the deletion operations discussed so far. The member function `deleteNode()` (Figure 3.2) is an implementation of this process.

FIGURE 3.7

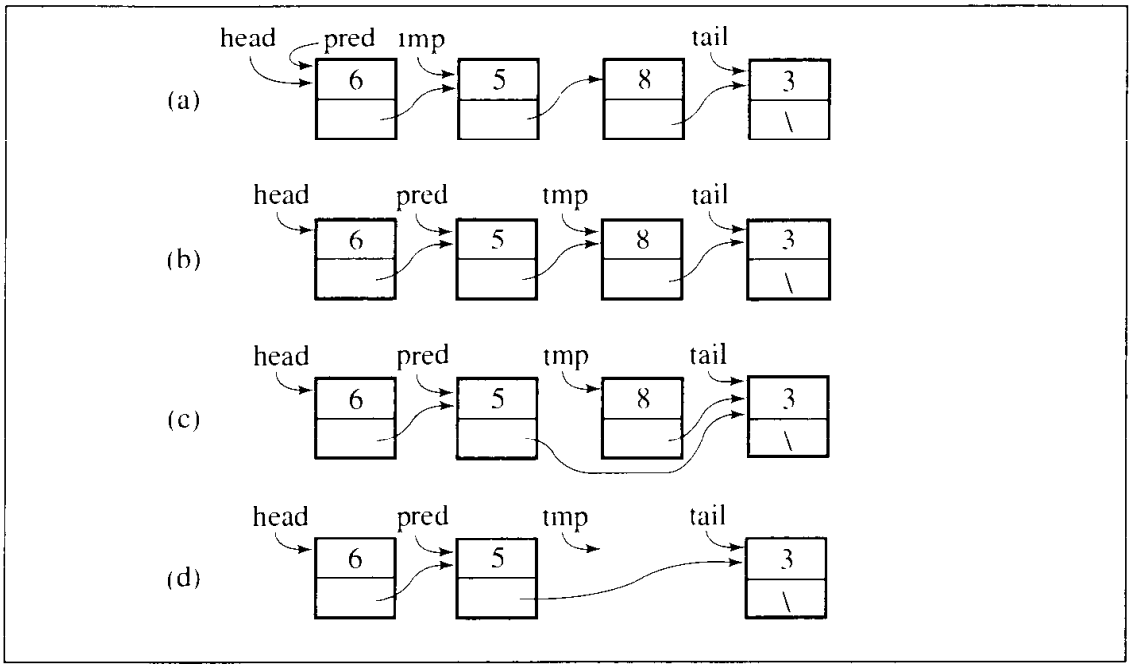
Deleting a node from the end of a singly linked list.



A node is removed from inside a list by linking its predecessor to its successor. But because the list has only forward links, the predecessor of a node is not reachable from the node. One way to accomplish the task is to find the node to be removed first by scanning the list and then scanning it again to find its predecessor. Another way is presented in `deleteNode()`, as shown in Figure 3.8. Assume that we want to delete a node that holds number 8. The function uses two pointer variables, `pred` and `tmp`, which are initialized in the `for` loop so that they point to the first and second nodes of the list, respectively (Figure 3.8a). Because the node `tmp` has number 5, the first iteration is executed in which both `pred` and `tmp` are advanced to the next nodes (Figure 3.8b). Because the condition of the `for` loop is now true (`tmp` points to the node with 8), the loop is exited and an assignment `pred->next = tmp->next` is executed (Figure 3.8c). This assignment effectively excludes the node with 8 from the list. The node is still accessible from variable `tmp`, and this access is used to return space occupied by this node to the pool of free memory cells by executing `delete` (Figure 3.8d).

The preceding paragraph discusses only one case. Here are the remaining cases:

FIGURE 3.8 Deleting a node from a singly linked list.



1. An attempt to remove a node from an empty list, in which case the function is immediately exited.
2. Deleting the only node from a one-node linked list: Both `head` and `tail` are set to null.
3. Removing the first node of the list with at least two nodes, which requires updating `head`.
4. Removing the last node of the list with at least two nodes, leading to the update of `tail`.
5. An attempt to delete a node with a number that is not in the list: Do nothing.

It is clear that the best case for `deleteNode()` is when the head node is to be deleted, which takes $O(1)$ time to accomplish. The worst case is when the last node needs to be deleted, which reduces `deleteNode()` to `deleteFromTail()` and to its $O(n)$ performance. What is the average case? It depends on how many iterations the `for` loop executes. Assuming that any node on the list has an equal chance to be deleted, the loop performs no iteration if it is the first node, one iteration if it is the second node, . . . , and finally $n - 1$ iterations if it is the last node. For a long sequence of deletions, one deletion requires on the average

$$\frac{0+1+\ldots+(n-1)}{n} = \frac{\frac{(n-1)n}{2}}{n} = \frac{n-1}{2}$$

That is, on the average, `deleteNode()` executes $O(n)$ steps to finish, just like in the worst case.

3.1.3 Search

The insertion and deletion operations modify linked lists. The searching operation scans an existing list to learn whether or not a number is in it. We implement this operation with the Boolean member function `isInList()`. The function uses a temporary variable `tmp` to go through the list starting from the head node. The number stored in each node is compared to the number being sought, and if the two numbers are equal, the loop is exited; otherwise, `tmp` is updated to `tmp->next` so that the next node can be investigated. After reaching the last node and executing the assignment `tmp = tmp->next`, `tmp` becomes null, which is used as an indication that the number `el` is not in the list. That is, if `tmp` is not null, the search was discontinued somewhere inside the list because `el` was found. That is why `isInList()` returns the result of comparison `tmp != null`: If `tmp` is not null, `el` was found and `true` is returned. If `tmp` is null, the search was unsuccessful and `false` is returned.

With reasoning similar to that used to determine the efficiency of `deleteNode()`, `isInList()` takes $O(1)$ time in the best case and $O(n)$ in the worst and average cases.

In the foregoing discussion, the operations on nodes have been stressed. However, a linked list is built for the sake of storing and processing information, not for the sake of itself. Therefore, the approach used in this section is limited in that the list can only store integers. If we wanted a linked list for float numbers or arrays of numbers, then a new class would have to be declared with a new set of member functions, all of them resembling the ones discussed here. However, it is more advantageous to declare such a class only once without deciding in advance what type of data will be stored in it. This can be done very conveniently in C++ with templates. To illustrate the use of templates for list processing, the next section uses them to define lists, although examples of list operations are still limited to lists that store integers.

3.2 Doubly Linked Lists

The member function `deleteFromTail()` indicates a problem inherent to singly linked lists. The nodes in such lists contain only pointers to the successors; therefore, there is no immediate access to the predecessors. For this reason, `deleteFromTail()` was implemented with a loop which allowed us to find the predecessor of `tail`. Although this predecessor is, so to speak, within sight, it is out of reach. We have to scan the entire list to stop right in front of `tail` to delete it. For long lists and for frequent executions of `deleteFromTail()`, this may be an impediment to swift list processing. To avoid this problem, the linked list is redefined so that each node in the list has two pointers, one to the successor and one to the predecessor. A list of this type is called a *doubly linked list* and is illustrated in Figure 3.9. Figure 3.10 contains a fragment of implementation for a generic `DoublyLinkedList` class.

Member functions for processing doubly linked lists are slightly more complicated than their singly linked lists counterparts because there is one more pointer

FIGURE 3.9 A doubly linked list.

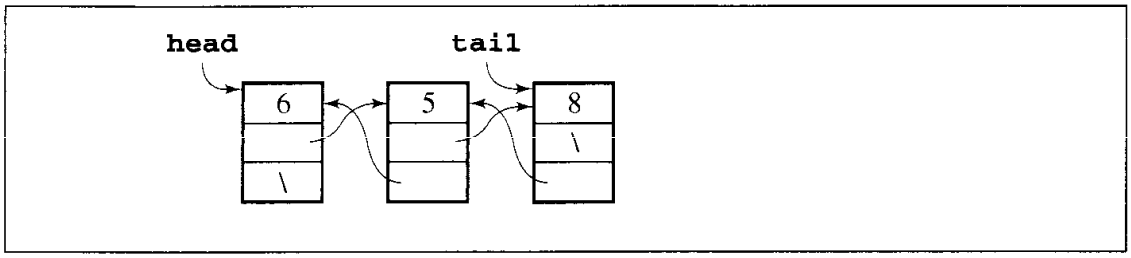


FIGURE 3.10 An implementation of a doubly linked list.

```
#ifndef DOUBLY_LINKED_LIST
#define DOUBLY_LINKED_LIST

#include <iostream.h>

template<class T>
class Node {
public:
    Node() {
        next = prev = 0;
    }
    Node(const T& el, Node *n = 0, Node *p = 0) {
        info = el; next = n; prev = p;
    }
    T info;
    Node *next, *prev;
};

template<class T>
class DoublyLinkedList {
public:
    DoublyLinkedList() {
        head = tail = 0;
    }
    void addToDLLTail(const T&);
    T deleteFromDLLTail();
    . . . . . .
protected:
    Node<T> *head, *tail;
};
```

Continues

FIGURE 3.10 *(continued)*

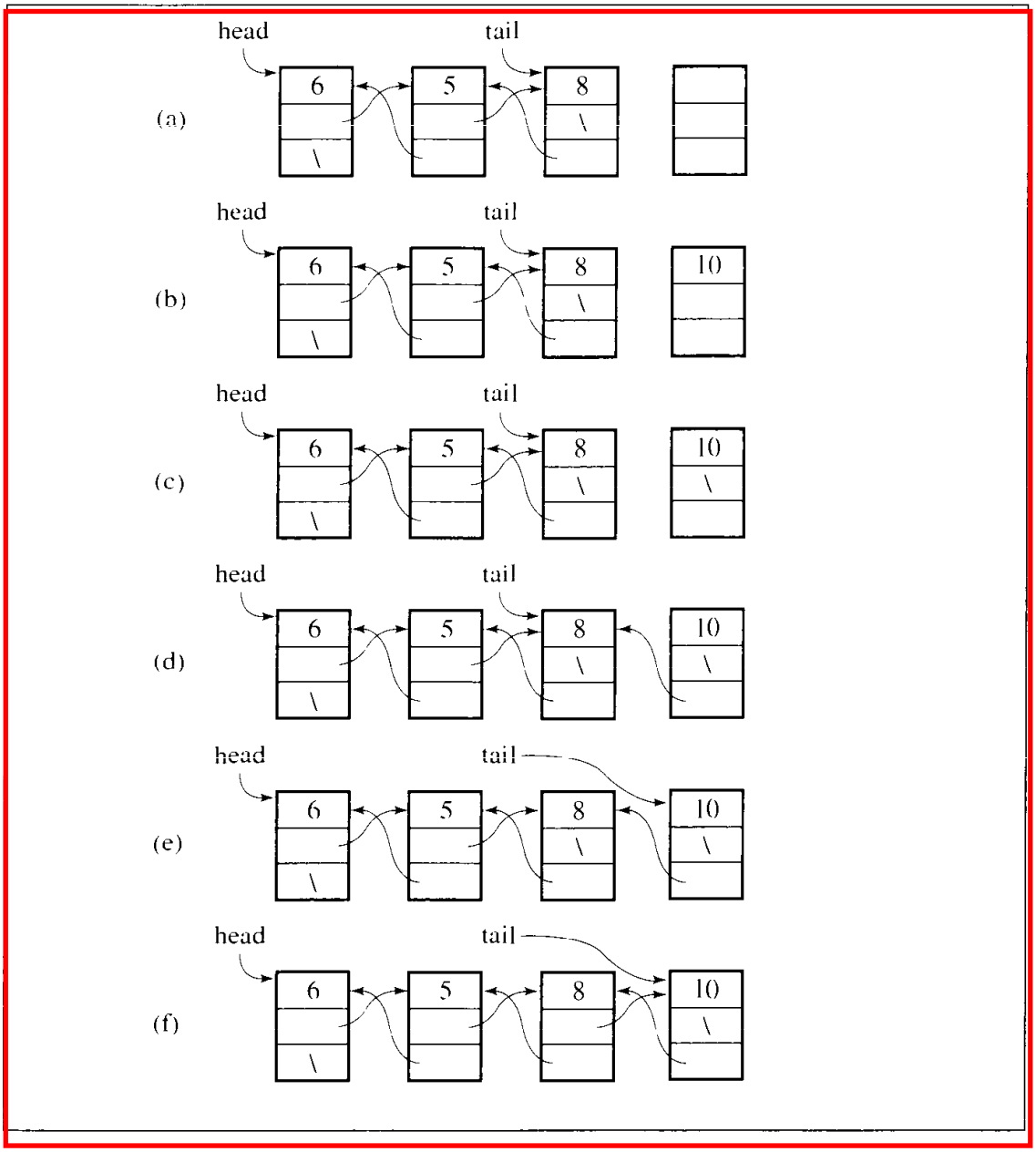
```
template<class T>
void DoublyLinkedList<T>::addToDLLTail(const T& el) {
    if (tail != 0) {
         tail = new Node<T>(el,0,tail);
         tail->prev->next = tail;
    }
    else head = tail = new Node<T>(el);
}
template<class T>
T DoublyLinkedList<T>::deleteFromDLLTail() {
    T el = tail->info;
    if (head == tail) { // if only one node in the list;
         delete head;
         head = tail = 0;
    }
    else {              // if more than one node in the list;
         tail = tail->prev;
         delete tail->next;
         tail->next = 0;
    }
    return el;
}
. . . . . .
#endif
```

member to be maintained. Only two functions are discussed: a function to insert a node at the end of the doubly linked list and a function to remove a node from the end (Figure 3.10).

To add a node to a list, the node has to be created, its data members properly initialized, and then the node needs to be incorporated into the list. Inserting a node at the end of a doubly linked list performed by `addToDLLTail()` is illustrated in Figure 3.11. The process is performed in six steps:

1. A new node is created (Figure 3.11a) and then its three data members are initialized:
2. the `info` member to the number `el` being inserted (Figure 3.11b),
3. the `next` member to null (Figure 3.11c),
4. and the `prev` member to the value of `tail` so that this member points to the last node in the list (Figure 3.11d). But now, the new node should become the last node; therefore,

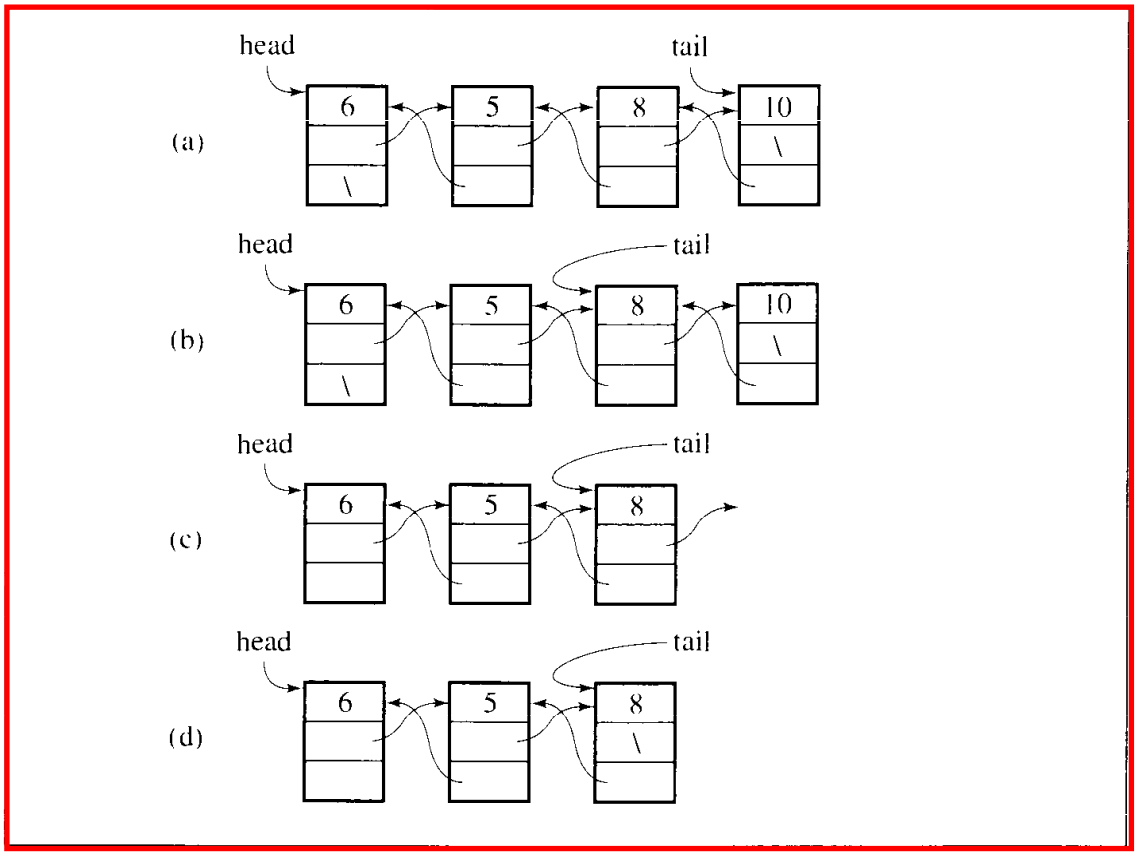
FIGURE 3.11 Adding a new node at the end of a doubly linked list.



5. `tail` is set to point to the new node (Figure 3.11e). But the new node is not yet accessible from its predecessor; to rectify this,
6. the `next` member of the predecessor is set to point to the new node (Figure 3.11f).

A special case concerns the last step. It is assumed in this step that the newly created node has a predecessor, so it accesses its `prev` member. It should be obvious that for an empty linked list, the new node is the only node in the list and that it has no predecessor. In this case, both `head` and `tail` refer to this node, and the sixth step is now setting `head` to point to this node. Note that step four—setting `prev` member to

FIGURE 3.12 Deleting a node from the end of a doubly linked list.



the value of `tail`—is executed properly because for an initially empty list, `tail` is null. Thus, null becomes the value of the `prev` member of the new node.

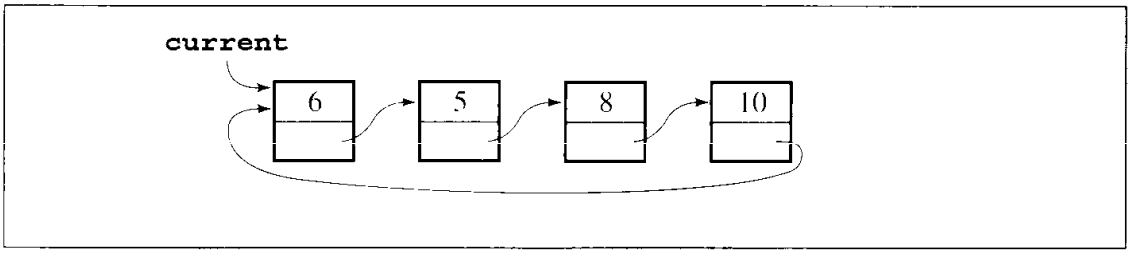
Deleting the last node from the doubly linked list is straightforward because there is direct access from the last node to its predecessor and no loop is needed to remove the last node. When deleting the last node from the list in Figure 3.12a, temporary variable `el` is set to the value in the node, then `tail` is set to its predecessor (Figure 3.12b), and the last and now redundant node is deleted (Figure 3.12c). In this way, the next to last node becomes the last node. The `next` member of the tail node is a dangling reference; therefore, `next` is set to null (Figure 3.12d). The last step is returning the copy of an object stored in the removed node.

An attempt to delete a node from an empty list may result in a program crash. Therefore, the user has to check whether the list is not empty before making an attempt of deleting the last node from it to extract information from the node. As for singly linked list's `deleteFromHead()`, the caller should have an `if` statement

```
if (!list.isEmpty())
     n = list.deleteFromDLLTail();
else do not remove;
```

Another special case is the deletion of the node from a single-node linked list. In this case, both `head` and `tail` are set to null.

FIGURE 3.13 A circular singly linked list.



Because of the immediate accessibility of the last node, both `addToDLLTail()` and `deleteFromDLLTail()` execute in constant time $O(1)$.

Functions for operating at the beginning of the doubly linked list are easily obtained from the two functions just discussed by changing `head` to `tail` and vice versa, changing `next` to `prev` and vice versa, and exchanging the order of parameters when executing `new`.

3.3 Circular Lists

In some situations, a *circular list* is needed in which nodes form a ring: The list is finite and each node has a successor. An example of such a situation is when several processes are using the same resource for the same amount of time, and we have to assure that no process accesses the resource before all other processes did. Therefore, all processes—let their numbers be 6, 5, 8, and 10 as in Figure 3.13—are put on a circular list accessible through the pointer `current`. After one node of the list is accessed and the process number is retrieved from the node to activate this process, `current` moves to the next node so that the next process can be activated the next time.

In an implementation of a circular singly linked list, we can use only one permanent pointer, `tail`, to the list even though operations on the list require access to the tail and its predecessor, the head. To that end, a linear singly linked list discussed in Section 3.1 uses two permanent pointers, `head` and `tail`.

Figure 3.14a shows a sequence of insertions at the front of the circular list, and Figure 3.14b illustrates insertions at the end of the list. As an example of a member function operating on such a list, we present a function to insert a node at the tail of a circular singly linked list in $O(1)$:

```
void addToTail(int el) {
   if (isEmpty()) {
        tail = new IntNode(el);
        tail->next = tail;
   }
   else {
        tail->next = new IntNode(el,tail->next);
        tail = tail->next;
}
```

FIGURE 3.14 Inserting nodes at the front of circular singly linked list (a) and at its end (b).

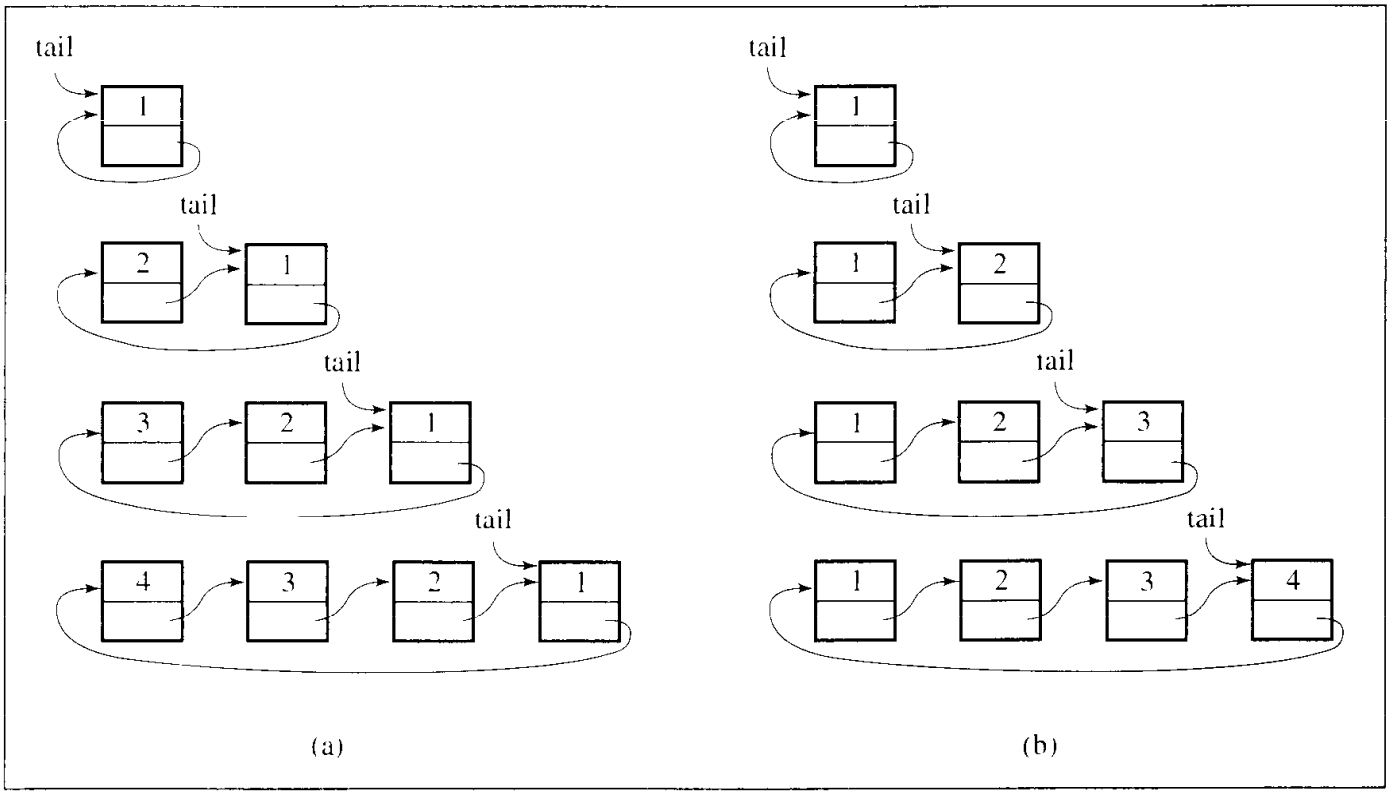
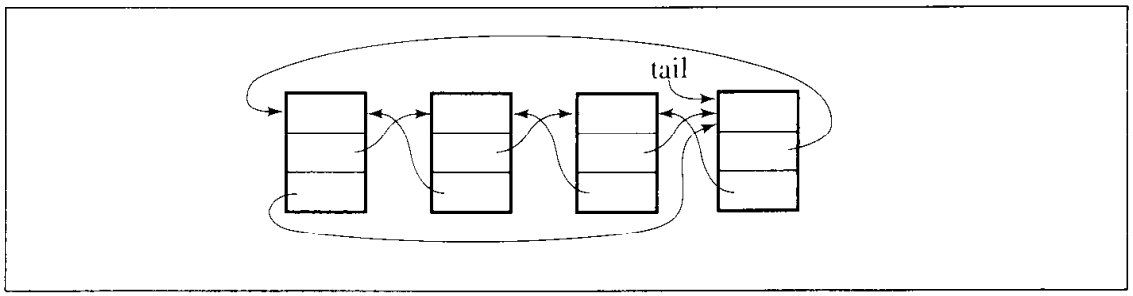
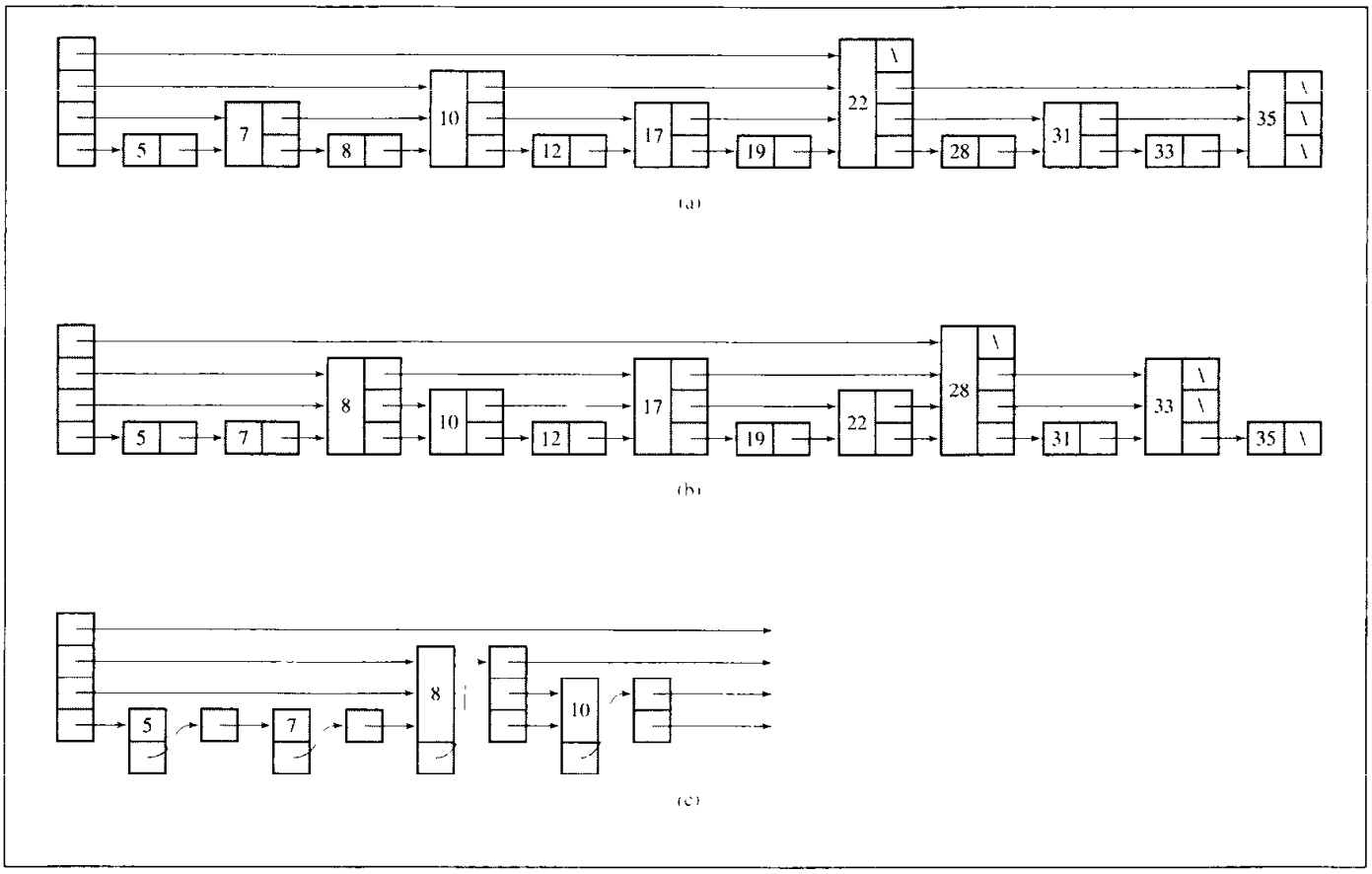


FIGURE 3.15 A circular doubly linked list.



The implementation just presented is not without its problems. A member function for deletion of the tail node requires a loop so that `tail` can be set after deletion to its predecessor. This makes this function delete the tail node in $O(n)$ time. Moreover, processing data in the reverse order (printing, searching, etc.) is not very efficient. To avoid the problem and still be able to insert and delete nodes at the front and at the end of the list without using a loop, a doubly linked circular list can be used. The list forms two rings: one going forward through `next` members and one going backward through `prev` members. Figure 3.15 illustrates such a list accessible through the last node. Deleting the node from the end of the list can be done easily

FIGURE 3.16 A skip list with (a) evenly and (b) unevenly spaced nodes of different levels; (c) the skip list with pointer nodes clearly shown.



10 and then again to 17. The last try is by starting the first-level sublist which begins in node 10; this sublist's first node has 12, the next number is 17, and since there is no lower level, the search is pronounced unsuccessful. Code for the searching procedure is given in Figure 3.17.

Searching appears to be efficient. However, the design of skip lists can lead to very inefficient insertion and deletion procedures. To insert a new element, all nodes following the node just inserted have to be restructured; the number of pointers and the values of pointers have to be changed. In order to retain some of the advantages which skip lists offer with respect to searching and avoid problems with restructuring the lists when inserting and deleting nodes, the requirement on the positions of nodes of different levels is now abandoned and only the requirement on the number of nodes of different levels is kept. For example, the list in Figure 3.16a becomes the list in Figure 3.16b: Both lists have six nodes with only one pointer, three nodes with two pointers, two nodes with three pointers, and one node with four pointers. The new list is searched exactly the same way as the original list. Inserting does not require list restructuring, and nodes are generated so that the distribution of the nodes on different levels is kept adequate. How can this be accomplished?

FIGURE 3.17 An implementation of a skip list.

```
//************************  genSkipL.h  ************************
//                      generic skip list class

const int maxLevel = 4;

template<class T>
class SkipListNode {
public:
    SkipListNode() {
    }
    T key;
    SkipListNode **next;
};

template<class T>
class SkipList {
public:
    SkipList();
    void choosePowers();
    int  chooseLevel();
    T* skipListSearch(const T&);
    void skipListInsert(const T&);
private:
    typedef SkipListNode<T> *nodePtr;
    nodePtr root[maxLevel];
    int powers[maxLevel];
};

template<class T>
SkipList<T>::SkipList() {
    for (int i = 0; i < maxLevel; i++)
        root[i] = 0;
}
template<class T>
void SkipList<T>::choosePowers() {
    powers[maxLevel-1] = (2 << (maxLevel-1)) - 1;  // 2^maxLevel - 1
    for (int i = maxLevel - 2, j = 0; i > = 0; i--, j++)
        powers[i] = powers[i+1] - (2 << j);        // 2^(j+1)
}
template<class T>
int SkipList<T>::chooseLevel() {
```

Continues

FIGURE 3.17 *(continued)*

```
    int i, r = rand() % powers[maxLevel-1] + 1;
    for (i = 1; i < maxLevel; i++)
        if (r < powers[i])
            return i-1; // return a level < the highest level;
    return i-1;         // return the highest level;
}
template<class T>
T* SkipList<T>::skipListSearch(const T& key) {
    nodePtr prev, curr;
    int lvl;                              // find the highest non-null
    for (lvl = maxLevel-1; lvl >= 0 && !root[lvl]; lvl--);  // level;
    prev = curr = root[lvl];
    while (1) {
        if (key == curr->key)                    // success if equal;
            return &curr->key;
        else if (key < curr->key) {              // if smaller, go down
            if (lvl == 0)                        // if possible,
                return 0;
            else if (curr == root[lvl])          // by one level
                curr = root[--lvl];              // starting from the
            else curr = *(prev->next + --lvl); // predecessor which
        }                                        // can be the root;
        else {                                   // if greater,
            prev = curr;                         // go to the next
            if (*(curr->next + lvl) != 0)        // non-null node
                curr = *(curr->next + lvl);      // on the same level
            else {                               // or to a list on a
                                                 // lower level;
                for (lvl--; lvl> = 0 && *(curr->next + lvl)==0; lvl--);
                if (lvl >= 0)
                    curr = *(curr->next + lvl);
                else return 0;
            }
        }
    }
}
template<class T>
void SkipList<T>::skipListInsert(const T& key) {
    nodePtr curr[maxLevel], prev[maxLevel], newNode;
    int lvl, i;
```

FIGURE 3.17 *(continued)*

```
    curr[maxLevel-1] = root[maxLevel-1];
    prev[maxLevel-1] = 0;
    for (lvl = maxLevel - 1; lvl >= 0; lvl--) {
        while (curr[lvl] && curr[lvl]->key < key) { // go to the next
            prev[lvl] = curr[lvl];                  // if smaller;
            curr[lvl] = *(curr[lvl]->next + lvl);
        }
        if (curr[lvl] && curr[lvl]->key == key)       // don't include
            return;                                   // duplicates;
        if (lvl > 0)                                  // go one level down
            if (prev[lvl] == 0) {                     // if not the lowest
                curr[lvl-1] = root[lvl-1];            // level, using a link
                prev[lvl-1] = 0;                      // either from the root
            }
            else {                                    // or from the predecessor;
                curr[lvl-1] = *(prev[lvl]->next + lvl-1);
                prev[lvl-1] = prev[lvl];
            }
    }
    lvl = chooseLevel();        // generate randomly level for newNode;
    newNode = new SkipListNode<T>;
    newNode->next = new nodePtr[sizeof(nodePtr) * (lvl+1)];
    newNode->key  = key;
    for (i = 0; i <= lvl; i++) {          // initialize next fields of
        *(newNode->next + i) = curr[i];   // newNode and reset to newNode
        if (prev[i] == 0)                 // either fields of the root
            root[i] = newNode;            // or next fields of newNode's
        else *(prev[i]->next + i) = newNode;    // predecessors;
    }
}
```

Assume that *maxLevel* = 4. For 15 elements, the required number of one-pointer nodes is eight, two-pointer nodes is four, three-pointer nodes is two, and four-pointer nodes is one. Each time a node is inserted, a random number r between 1 and 15 is generated, and if $r < 9$, then a node of level one is inserted. If $r < 13$, a second-level node is inserted, if $r < 15$, it is a third-level node, and if $r = 15$, the node of level four is generated and inserted. If *maxLevel* = 5, then for 31 elements the correspondence between the value of r and the level of node is as follows:

r	Level of Node to Be Inserted
31	5
29–30	4
25–28	3
17–24	2
1–16	1

To determine such a correspondence between *r* and the level of node for any *maxLevel*, the function `choosePowers()` initializes the array `powers[]` by putting lower bounds for each range. For example, for *maxLevel* = 4, the array is [1 9 13 15], and for *maxLevel* = 5, it is [1 17 25 29 31]. `chooseLevel()` uses `powers[]` to determine the level of the node about to be inserted. Figure 3.17 contains the code for `choosePowers()` and `chooseLevel()`. Note that the levels range between 0 and *maxLevel*–1 (and not between 1 and *maxLevel*) so that the array indexes can be used as levels. For example, the first level is level zero.

But we also have to address the question of implementing a node. The easiest way is to make each node have *maxLevel* pointers, but this is wasteful. We need only as many pointers per one node as the level of the node requires. To accomplish this, the `next` member of each node is not a pointer to the next node, but to an array of pointer(s) to the next node(s). The size of this array is determined by the level of the node. The `SkipListNode` and a `SkipList` classes are declared, as in Figure 3.17. In this way, the list in Figure 3.16b is really a list whose first four nodes are shown in Figure 3.16c. Only now can an inserting procedure be implemented, as in Figure 3.17.

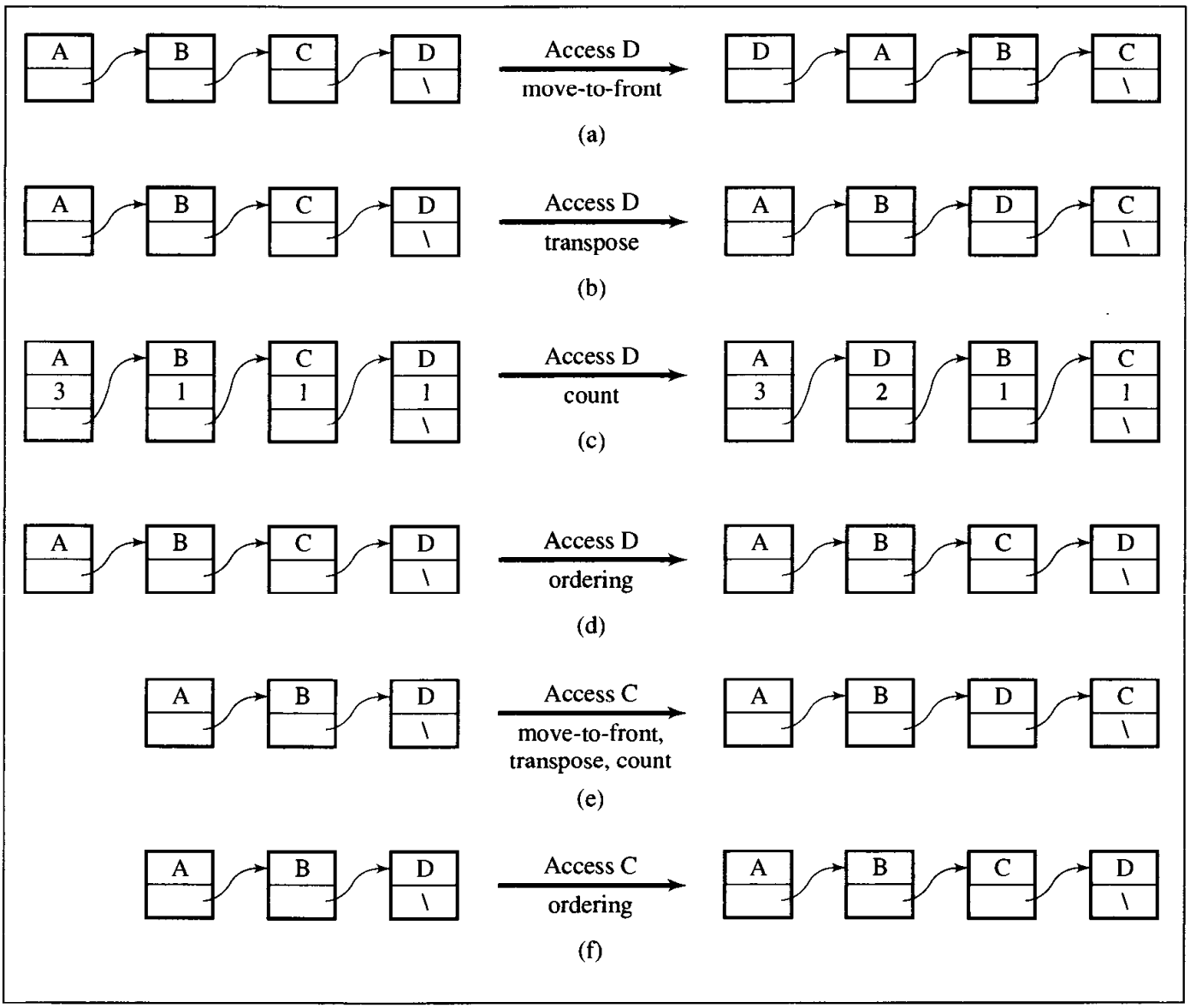
How efficient are skip lists? In the ideal situation, which is exemplified by the list in Figure 3.16a, the search time is $O(\lg n)$. In the worst situation, when all lists are on the same level, the skip list turns into a regular singly linked list, and the search time is $O(n)$. However, the latter situation is unlikely to occur; in the random skip list, the search time is of the same order as the best case, that is, $O(\lg n)$. This is an improvement over the efficiency of search in regular linked lists. It also turns out that skip lists fare extremely well in comparison with more sophisticated data structures, such as self-adjusting trees or AVL trees (cf. Sections 6.7.2, 6.8), and therefore they are a viable alternative to these data structures (see also the table in Figure 3.20).

3.5 Self-Organizing Lists

The introduction of skip lists was motivated by the need to speed up the searching process. Although singly and doubly linked lists require sequential search to locate an element or to see that it is not in the list, we can improve the efficiency of the search by dynamically organizing the list in a certain manner. This organization depends on the configuration of data; thus, the stream of data requires reorganizing the nodes already on the list. There are many different ways to organize the lists, and this section describes four of them.

1. *Move-to-front method.* After the desired element is located, put it at the beginning of the list.

FIGURE 3.18 Accessing an element on a linked list and changes on the list depending on the self-organization technique applied: (a) move-to-front method, (b) transpose method, (c) count method, and (d) ordering method, in particular, alphabetical ordering which leads to no change. In the case when the desired element is not in the list, (e) the first three methods add a new node with this element at the end of the list and (f) the ordering method maintains an order on the list.



2. *Transpose method.* After the desired element is located, swap it with its predecessor unless it is at the head of the list.
3. *Count method.* Order the list by the number of times elements are being accessed.
4. *Ordering method.* Order the list using certain criteria natural for information under scrutiny.

In the first three methods, new information is stored in a node added to the end of the list (Figure 3.18e); in the fourth method, new information is stored in a node inserted somewhere in the list to maintain the order of the list (Figure 3.18f). An

FIGURE 3.19 Processing the stream of data, A C B C D A D A C A C C E E, by different methods of organizing linked lists. Linked lists are presented in an abbreviated form; for example, the transformation shown in Figure 3.18a is abbreviated as transforming list A B C D into list D A B C.

element searched for	plain	move-to-front	transpose	count	ordering
A:	A	A	A	A	A
C:	AC	AC	AC	AC	AC
B:	ACB	ACB	ACB	ACB	ABC
C:	ACB	CAB	CAB	CAB	ABC
D:	ACBD	CABD	CABD	CABD	ABCD
A:	ACBD	ACBD	ACBD	CABD	ABCD
D:	ACBD	DACB	ACDB	DCAB	ABCD
A:	ACBD	ADCB	ACDB	ADCB	ABCD
C:	ACBD	CADB	CADB	CADB	ABCD
A:	ACBD	ACDB	ACDB	ACDB	ABCD
C:	ACBD	CADB	CADB	ACDB	ABCD
C:	ACBD	CADB	CADB	CADB	ABCD
E:	ACBDE	CADBE	CADBE	CADBE	ABCDE
E:	ACBDE	ECADB	CADEB	CAEDB	ABCDE

example of searching for elements in a list organized by these different methods is shown in Figure 3.19.

With the first three methods, we try to locate the elements most likely to be looked for near the beginning of the list, most explicitly with the move-to-front method and most cautiously with the transpose method. The ordering method already uses some properties inherent to the information stored in the list. For example, if we are storing nodes pertaining to people, then the list can be organized alphabetically by the name of the person or the city or in ascending or descending order using, say, birthday or salary. This is particularly advantageous when searching for information which is not in the list, since the search can terminate without scanning the entire list. Searching all the nodes of the list, however, is necessary in such cases using the other three methods. The count method can be subsumed in the category of the ordering methods if frequency is part of the information. In many cases, however, the count itself is an additional piece of information required solely to maintain the list; hence, it may to be considered "natural" to the information at hand.

Analyses of the efficiency of these methods customarily compare their efficiency to that of *optimal static ordering*. With this ordering, all the data are already ordered by the frequency of their occurrence in the body of data so that the list is used only for searching, not for inserting new items. Therefore, this approach requires two passes through the body of data, one to build the list and another to use the list for search alone.

To experimentally measure the efficiency of these methods, the number of all actual comparisons was compared to the maximum number of possible comparisons. The latter number is calculated by adding the lengths of the list at the moment of processing each element. For example, in the table in Figure 3.19, the body of data contains 14 letters, 5 of them being different, which means that 14 letters were processed. The length of the list before processing each letter is recorded, and the result, 0 + 1 + 2 + 3 + 3 + 4 + 4 + 4 + 4 + 4 + 4 + 4 + 4 + 5 = 46, is used to compare the number of all made comparisons to this combined length. In this way, we know what percentage of the list was scanned during the entire process. For all the list organizing methods except optimal ordering, this combined length is the same; only the number of comparisons can change. For example, when using the move-to-front technique for the data in the table in Figure 3.19, 33 comparisons were made, which is 71.7% when compared to 46. The latter number gives the worst possible case, the combined length of intermediate lists every time all the nodes in the list are looked at. Plain search, with no reorganization, required only 30 comparisons, which is 65.2%.

These samples are in agreement with theoretical analyses which indicate that count and move-to-front methods are, in the long run, at most twice as costly as the optimal static ordering; the transpose method approaches, in the long run, the cost of the move-to-front method. In particular, with amortized analysis, it can be established that the cost of accessing a list element with the move-to-front method is at most twice the cost of accessing this element on the list that uses optimal static ordering.

In a proof of this statement, the concept of inversion is used. For two lists containing the same elements, an inversion is defined to be a pair of elements (x, y) such that on one list x precedes y and on the other list y precedes x. For example, the list (C, B, D, A) has four inversions with respect to list (A, B, C, D): (C, A), (B, A), (D, A), and (C, B). Define the amortized cost to be the sum of actual cost and the difference between the number of inversions before accessing an element and after accessing it,

$$amCost(x) = cost(x) + (inversionsBeforeAccess(x) - inversionsAfterAccess(x))$$

To assess this number, consider an optimal list OL = (A, B, C, D) and a move-to-front list MTF = (C, B, D, A). The access of elements usually changes the balance of inversions. Let $displaced(x)$ be the number of elements preceding x in MTF but following x in OL. For example, $displaced(A) = 3$, $displaced(B) = 1$, $displaced(C) = 0$, and $displaced(D) = 0$. If $pos_{MTF}(x)$ is the current position of x in MTF, then $pos_{MTF}(x) - 1 - displaced(x)$ is the number of elements preceding x in both lists. It is easy to see that for D this number equals 2, and for the remaining elements it is 0. Now, accessing an element x and moving it to the front of MTF creates $pos_{MTF}(x) - 1 - displaced(x)$ new inversions and removes $displaced(x)$ other inversions so that the amortized time to access x is

$$amCost(x) = pos_{MTF}(x) + pos_{MTF}(x) - 1 - displaced(x) - displaced(x) =$$
$$2(pos_{MTF}(x) - displaced(x)) - 1$$

where $cost(x) = pos_{MTF}(x)$. Accessing A transforms MTF = (C, B, D, A) into (A, C, B, D) and $amCost(A) = 2(4 - 3) - 1 = 1$. For B, the new list is (B, C, D, A) and $amCost(B) = 2(2 - 1) - 1 = 1$. For C, the list does not change and $amCost(C) = 2(1 - 0) - 1 = 1$. Finally, for D, the new list is (D, C, B, A) and $amCost(D) = 2(3 - 0) - 1 = 5$. However, the number of common elements preceding x on the two lists cannot exceed the number of all the elements preceding x on OL; therefore, $pos_{MTF}(x) - 1 - displaced(x) \leq pos_{OL}(x) - 1$, so that

$$amCost(x) \leq 2pos_{OL}(x) - 1$$

The amortized cost of accessing an element x in MTF is in excess of $pos_{OL}(x) - 1$ units to its actual cost of access on OL. This excess is used to cover an additional cost of accessing elements in MTF for which $pos_{MTF}(x) > pos_{OL}(x)$, that is, elements that require more accesses on MTF than on OL.

It is important to stress that the amortized costs of single operations are meaningful in the context of sequences of operations. A cost of an isolated operation may seldom equal its amortized cost; however, in a sufficiently long sequence of accesses, each access on the average takes at most $2pos_{OL}(x) - 1$ time.

Figure 3.20 contains sample runs of the self-organizing lists. The first two columns of numbers refer to files containing programs, and the remaining columns refer to files containing English text. Except for alphabetical ordering, all methods improve their efficiency with the size of the file. The move-to-front and count methods are almost the same in their efficiency, and both outperform the transpose, plain, and ordering methods. The poor performance for smaller files is due to the fact that all of the methods are busy including new words to the lists, which requires an exhaustive search of the lists. Later, the methods concentrate on organizing the lists to reduce the number of searches. The table in Figure 3.20 also includes data for a skip list. There is an overwhelming difference between the skip list's efficiency compared to the other methods. However, keep in mind that in the table in Figure 3.20, only comparisons of data are included with no indication of the other operations needed for execution of the analyzed methods. In particular, there is no indication of how many pointers are used and relinked, which, when included, may make the difference between various methods less dramatic.

These sample runs show that for lists of modest size, the linked list suffices. With the increase in the amount of data and in the frequency with which they have to be accessed, more sophisticated methods and data structures need to be used.

3.6 Sparse Tables

In many applications, the choice of a table seems to be the most natural one, but space considerations may preclude this choice. This is particularly true if only a small fraction of the table is actually used. A table of this type is called a *sparse table* since the

FIGURE 3.20 Measuring the efficiency of different methods using formula (number of data comparison)/(combined length) expressed in percentages.

Different Words/ All Words	149/423	550/2847	156/347	609/1510	1163/5866	2013/23065
Optimal	26.4	17.6	28.5	24.5	16.2	10.0
Plain	71.2	56.3	70.3	67.1	51.7	35.4
Move-to-Front	49.5	31.3	61.3	54.5	30.5	18.4
Transpose	69.5	53.3	68.8	66.1	49.4	32.9
Count	51.6	34.0	61.2	54.7	32.0	19.8
Alphabetical Order	45.6	55.7	50.9	48.0	50.4	50.0
Skip List	12.3	5.5	15.1	6.6	4.8	3.8

table is populated sparsely by data and most of its cells are empty. In this case, the table can be replaced by a system of linked lists.

As an example, consider the problem of storing grades for all students in a university for a certain semester. Assume that there are 8000 students and 300 classes. A natural implementation is a two-dimensional array *grades* where student numbers are indexes of the columns and class numbers the indexes of the rows (see Figure 3.21). An association of student names and numbers is represented by the one-dimensional array *students* and an association of class names and numbers by the array *classes*. The names do not have to be ordered. If order is required, then another array can be used where each array element is occupied by a record with two fields, name and number,[1] or the original array can be sorted each time an order is required. This, however, leads to the constant reorganization of *grades* and is not recommended.

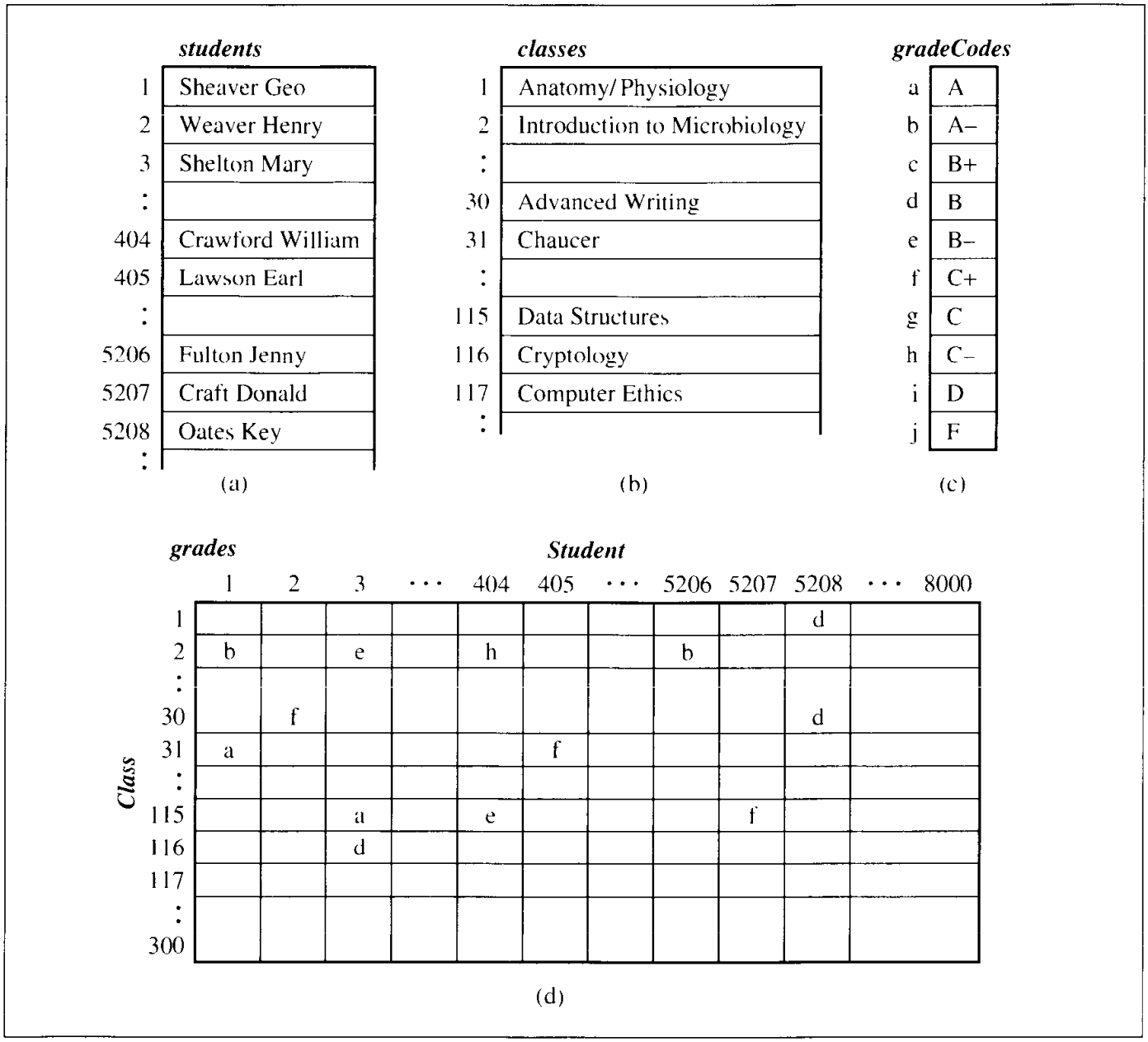
Each cell of *grades* stores a grade obtained by each student after finishing a class. If signed grades such as A–, B+, or C+ are used, then two bytes are require to store each grade. To reduce the table size by one-half, the array *gradeCodes* in Figure 3.21c associates each grade with a letter which requires only one byte of storage.

The entire table (Figure 3.21d) occupies 8000 students · 300 classes · 1 byte = 2.4 million bytes. This table is very large but is sparsely populated by grades. Assuming that, on the average, students take four classes a semester, each column of the table has only four cells occupied by grades, and the rest of the cells, 296 cells or 98.7%, are unoccupied and wasted.

A better solution is to use two 2-dimensional arrays. *classesTaken* represents all the classes taken by every student and *studentsInClasses* represents all students participating in each class (see Figure 3.22). A cell of each table is an object with two data

[1]This is called an *index-inverted table.*

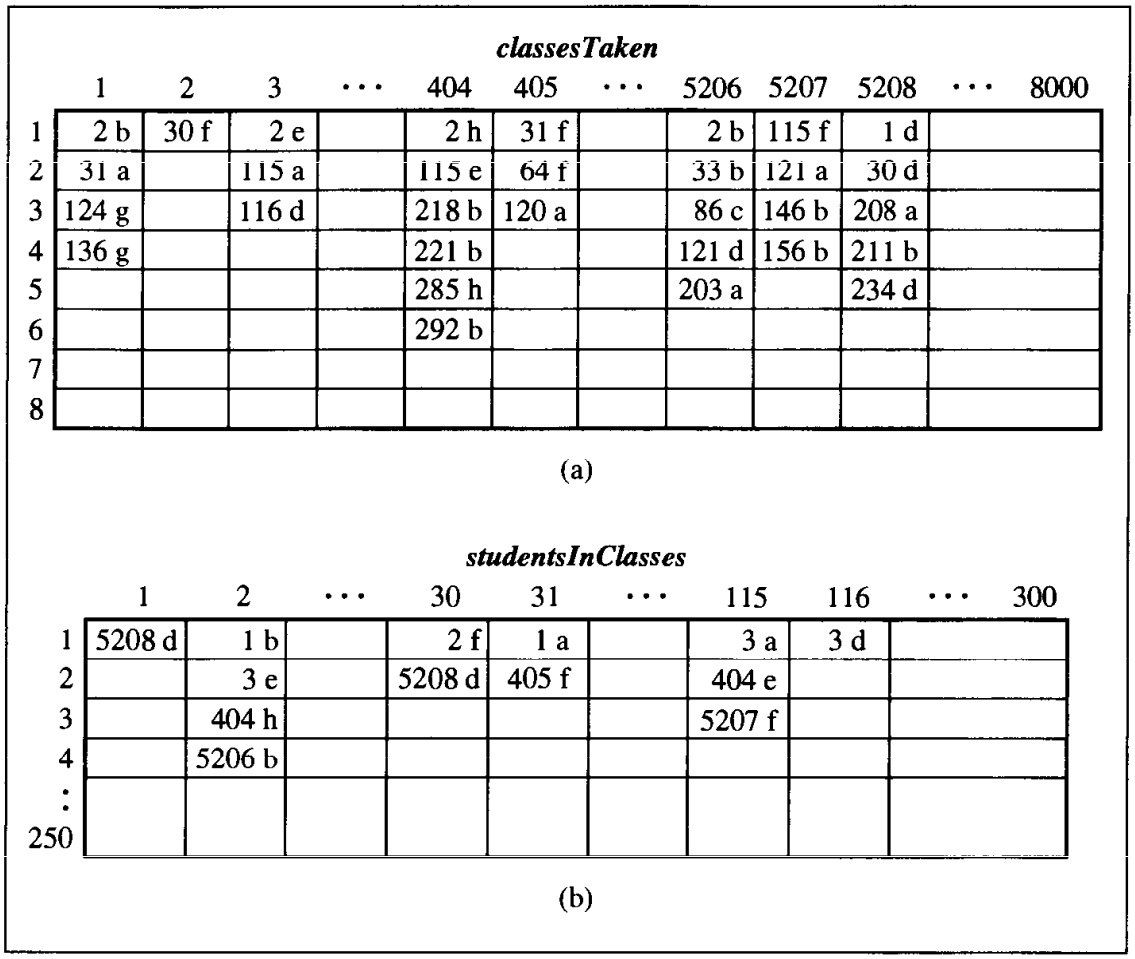
FIGURE 3.21 Arrays and sparse table used for storing student grades.



members: a student or class number and a grade. We assume that a student can take at most eight classes and that there can be at most 250 students signed up for a class. We need two arrays, since with only one array it is very time-consuming to produce lists. For example, if only *classesTaken* is used, then printing a list of all students taking a particular class requires an exhaustive search of *classesTaken*.

Assume that the computer on which this program is being implemented requires two bytes to store an integer. With this new structure, three bytes are needed for each cell. Therefore, *classesTaken* occupies 8000 students · 8 classes · 3 bytes = 192,000 bytes, *studentsInClasses* occupies 300 classes · 250 students · 3 bytes = 225,000 bytes, and both tables require a total of 417,000 bytes, less than one-fifth the number of bytes required for the sparse table in Figure 3.21.

FIGURE 3.22 Two-dimensional arrays for storing student grades.



classesTaken

	1	2	3	...	404	405	...	5206	5207	5208	...	8000
1	2 b	30 f	2 e		2 h	31 f		2 b	115 f	1 d		
2	31 a		115 a		115 e	64 f		33 b	121 a	30 d		
3	124 g		116 d		218 b	120 a		86 c	146 b	208 a		
4	136 g				221 b			121 d	156 b	211 b		
5					285 h			203 a		234 d		
6					292 b							
7												
8												

(a)

studentsInClasses

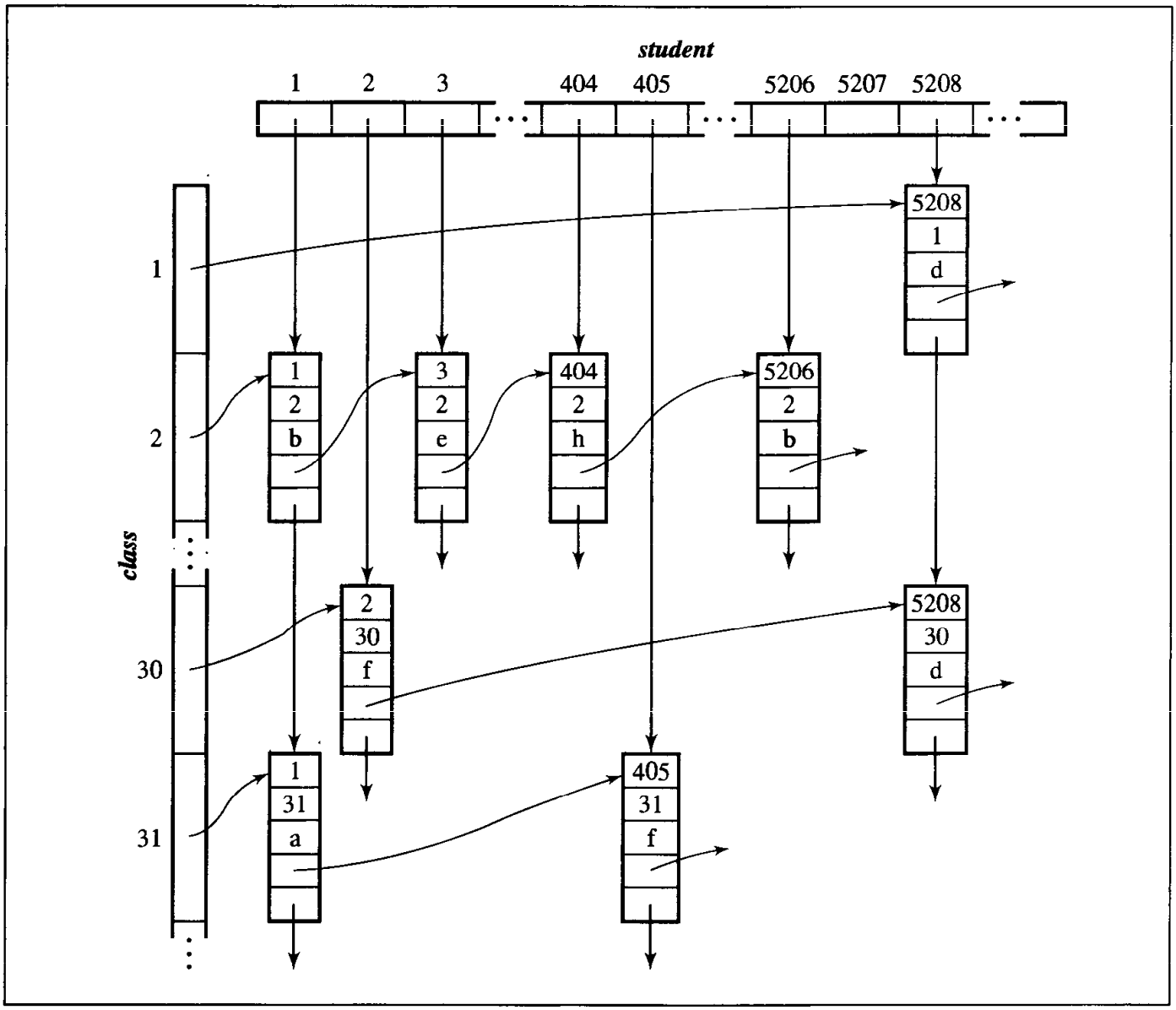
	1	2	...	30	31	...	115	116	...	300
1	5208 d	1 b		2 f	1 a		3 a	3 d		
2		3 e		5208 d	405 f		404 e			
3		404 h					5207 f			
4		5206 b								
:										
250										

(b)

Although this is a much better implementation than before, it still suffers from a wasteful use of space; seldom if ever will both arrays be full since most classes have fewer than 250 students and most students take fewer than eight classes. This structure is also inflexible: If a class can be taken by more than 250 students, a problem occurs which has to be circumvented in an artificial way. One way is to create a nonexistent class which holds students for the overflowing class. Another way is to recompile the program with a new table size, which may not be practical at a future time. Another more flexible solution is needed that uses space frugally.

Two one-dimensional arrays of linked lists can be used as in Figure 3.23. Each cell of the array *class* is a pointer to a linked list of students taking a class, and each cell of the array *student* indicates a linked list of classes taken by a student. The linked lists contain nodes of five data members: student number, class number, grade, a pointer to the next student, and a pointer to the next class. Assuming that each pointer requires only two bytes, one node occupies nine bytes, and the entire structure can be stored in 8000 students · 4 classes (on the average) · 9 bytes = 288,000 bytes, which is approximately 10% of the space required for the first implementation and about 70% of the space of the second. No space is used unnecessarily, there is no restriction imposed on the number of students per class, and the lists of students taking a class can be printed immediately.

FIGURE 3.23 Student grades implemented using linked lists.



3.7 LISTS IN THE STANDARD TEMPLATE LIBRARY

The list sequence container is an implementation of various operations on the nodes of a linked list. The STL implements a list as a generic doubly linked list with pointers to the head and to the tail. An instance of such a list that stores integers is presented in Figure 3.9.

The class `list` can be used in a program only if it is included with the instruction

```
#include <list>
```

The member functions included in the list container are presented in Figure 3.24.

FIGURE 3.24 An alphabetical list of member functions in the class `list`.

Member Function	Action and Return Value
`void assign(first, last)`	remove all the nodes in the list and insert in it the elements from the range indicated by iterators `first` and `last`
`void assign(n, el = T())`	remove all the nodes in the list and insert in it `n` copies of `el` (if `el` is not provided, a default constructor `T()` is used)
`T& back()`	return the element in the last node of the list
`const T& back() const`	return the element in the first node of the list
`iterator begin()`	return an iterator that references the first node of the list
`const_iterator begin() const`	return a `const` iterator that references the first node of the list
`void clear()`	remove all the nodes in the list
`bool empty() const`	return `true` if the list includes no nodes and false otherwise
`iterator end()`	return an iterator that is past the last node of the list
`const_iterator end() const`	return a `const` iterator that is past the last node of the list
`iterator/void erase(i)`	remove the node referenced by iterator `i`
`iterator/void erase(first, last)`	remove the nodes in the range indicated by iterators `first` and `last`
`T& front()`	return the element in the first node of the list
`const T& front() const`	return the element in the first node of the list
`iterator insert(i, el = T())`	insert `el` before the node referenced by iterator `i` and return iterator referencing the new node
`void insert(i, n, el)`	insert `n` copies of `el` before the node referenced by iterator `i`
`void insert(i, first, last)`	insert elements from location referenced by `first` to location referenced by `last` before the node referenced by iterator `i`
`list()`	construct an empty list
`list(n, el = T())`	construct a list with `n` copies of `el` of type `T`
`list(first, last)`	construct a list with the elements from the range indicated by iterators `first` and `last`
`list(lst)`	copy constructor
`size_type max_size() const`	return the maximum number of nodes for the list
`void merge(lst)`	for the sorted current list and `lst`, remove all nodes from `lst` and insert them in sorted order in the current list
`void merge(lst, f)`	for the sorted current list and `lst`, remove all nodes from `lst` and insert them in the current list in the sorted order specified by a two-argument Boolean function `f()`

Continues

FIGURE 3.24 *(continued)*

`void pop_back()`	remove the last node of the list
`void pop_front()`	remove the first node of the list
`void push_front(el)`	insert `el` at the head of the list
`void push_back(el)`	insert `el` at the end of the list
`reverse_iterator rbegin()`	return an iterator that references the last node of the list
`const_reverse_iterator rbegin() const`	return a `const` iterator that references the last node of the list
`void remove(el)`	remove from the list all the nodes that include `el`
`void remove_if(f)`	remove the nodes for which a one-argument Boolean function `f()` returns `true`
`reverse_iterator rend()`	return an iterator that is before the first node of the list
`const_reverse_iterator rend() const`	return a `const` iterator that is before the first node of the list
`void resize(n, el = T())`	make the list have `n` nodes by adding `n - size()` more nodes with element `el` or by discarding overflowing `size() - n` nodes from the end of the list
`size_type size() const`	return the number of nodes in the list
`void sort()`	sort elements of the list in ascending order
`void sort(f)`	sort elements of the list in the order specified by a one-argument Boolean function `f()`
`void splice(i, lst)`	remove the nodes of list `lst` and insert them into the list before the position referenced by iterator `i`
`void splice(i, lst, j)`	remove from list `lst` the node referenced by iterator `j` and insert it into the list before the position referenced by iterator `i`
`void splice(i, lst, first, last)`	remove from list `lst` the nodes in the range indicated by iterators `first` and `last` and insert them into the list before the position referenced by iterator `i`
`void swap(lst)`	swap the content of the list with the content of another list `lst`
`void unique()`	remove duplicate elements from the sorted list
`void unique(f)`	remove duplicate elements from the sorted list where being a duplicate is specified by a two-argument Boolean function `f()`

A new list is generated with the instruction

```
list<T> lst;
```

where `T` can be any data type. If it is a user-defined type, the type must also include a default constructor which is required for initialization of new nodes. Otherwise, the compiler is unable to compile the member functions with arguments initialized by the default constructor. These include one constructor and functions `resize()`, `assign()`, and one version of `insert()`. Note that this problem does not arise when creating a list of pointers to user-defined types, as in

```
list<T*> ptrLst;
```

The working of most of the member functions has already been illustrated in the case of the vector container (see Figure 1.4 and the discussion of these functions in Section 1.8). Vector container has only three member functions not found in the list container (`at()`, `capacity()`, and `reserve()`), but there are a number of list member functions that are not found in the vector container. Examples of their operation are presented in Figure 3.25.

FIGURE **3.25** A program demonstrating the operation of `list` member functions.

```
#include <iostream>
#include <list>
#include <algorithm>

using namespace std;

void main() {
    list<int> lst1;              // lst1 is empty
    list<int> lst2(3,7);         // lst2 = (7 7 7)
    for (int j = 1; j <= 5; j++) // lst1 = (1 2 3 4 5)
        lst1.push_back(j);
    list<int>::iterator i1 = lst1.begin(), i2 = i1, i3;
    i2++; i2++; i2++;
    list<int> lst3(++i1,i2);     // lst3 = (2 3)
    list<int> lst4(lst1);        // lst4 = (1 2 3 4 5)
    i1 = lst4.begin();
    lst4.splice(++i1,lst2);      // lst2 is empty,
                                 // lst4 = (1 7 7 7 2 3 4 5)
    lst2 = lst1;                 // lst2 = (1 2 3 4 5)
    i2 = lst2.begin();
    lst4.splice(i1,lst2,++i2);   // lst2 = (1 3 4 5),
                                 // lst4 = (1 7 7 7 2 2 3 4 5)
    i2 = lst2.begin();
```

Continues

FIGURE 3.25 *(continued)*

```
    i3 = i2;
    lst4.splice(i1,lst2,i2,++i3); // lst2 = (3 4 5),
                                  // lst4 = (1 7 7 7 2 1 2 3 4 5)
    lst4.remove(1);               // lst4 = (7 7 7 2 2 3 4 5)
    lst4.sort();                  // lst4 = (2 2 3 4 5 7 7 7)
    lst4.unique();                // lst4 = (2 3 4 5 7)
    lst1.merge(lst2);             // lst1 = (1 2 3 3 4 4 5 5),
                                  // lst2 is empty
    lst3.reverse();               // lst3 = (3 2)
    lst4.reverse();               // lst4 = (7 5 4 3 2)
    lst3.merge(lst4,greater<int>());     // lst3 = (7 5 4 3 3 2 2),
                                         // lst4 is empty
    lst3.remove_if(bind2nd(not_equal_to<int>(),3));// lst3 = (3 3)
    lst3.unique(not_equal_to<int>());    // lst3 = (3 3)
}
```

3.8 DEQUES IN THE STANDARD TEMPLATE LIBRARY

A *deque* (double-ended queue) is a list that allows for direct access to both ends of the list particularly to insert and delete elements. Hence, a deque can be implemented as a doubly linked list with pointer data members `head` and `tail` as discussed in Section 3.2. Moreover, as pointed out in the previous section, the container `list` uses a doubly linked list already. The STL, however, adds another functionality to the deque, namely, random access to any position of the deque, just as in arrays and vectors. Vectors, as discussed in Section 1.8, have poor performance for insertion and deletion of elements at the front, but these operations are quickly performed for doubly linked lists. This means that the STL deque should combine the behavior of a vector and a list.

The member functions of the STL container `deque` are listed in Figure 3.26. The functions are basically the same as those available for lists, with few exceptions. Deque does not include function `splice()`, which is specific to `list`, and functions `merge()`, `remove()`, `sort()`, and `unique()`, which are also available as algorithms, and `list` only reimplements them as member functions. The most significant difference is the function `at()` (and its equivalent, `operator[]`) that is unavailable in `list`. The latter function is available in vector, and if we compare the set of member functions in `vector` (Figure 1.3) and in `deque`, we see only a few differences. `vector` does not have `pop_front()` and `push_front()`, as does `deque`, but `deque` does not include functions `capacity()` and `reserve()`, which are available in `vector`. A few operations are illustrated in Figure 3.27. Note that for lists

FIGURE 3.26 A list of member functions in the class deque.

Member Function	Operation
`void assign(first, last)`	remove all the elements in the deque and insert in it the elements from the range indicated by iterators `first` and `last`
`void assign(n, el = T())`	remove all the elements in the deque and insert in it n copies of `el`
`T& at(n)`	return the element in position n of the deque
`const T& at(n) const`	return the element in position n of the deque
`T& back()`	return the last element in the deque
`const T& back() const`	return the last element in the deque
`iterator begin()`	return an iterator that references the first element of the deque
`const_iterator begin() const`	return a `const` iterator that references the first element of the deque
`void clear()`	remove all the elements in the deque
`deque()`	construct an empty deque
`deque(n, el = T())`	construct a deque with n copies of `el` of type `T` (if `el` is not provided, a default constructor `T()` is used)
`deque(dq)`	copy constructor
`deque(first, last)`	construct a deque and initialize it with values from the range indicated by iterators `first` and `last`
`bool empty() const`	return `true` if the deque includes no elements and `false` otherwise
`iterator end()`	return an iterator that is past the last element of the deque
`const_iterator end() const`	return a `const` iterator that is past the last element of the deque
`iterator/void erase(i)`	remove the element referenced by iterator `i`
`iterator/void erase(first, last)`	remove the elements in the range indicated by iterators `first` and `last`
`T& front()`	return the first element in the deque
`const T& front() const`	return the first element in the deque
`iterator insert(i, el = T())`	insert `el` before the element indicated by iterator `i` and return iterator referencing the newly inserted element
`void insert(i, n, el)`	insert n copies of `el` before the element referenced by iterator `i`

Continues

FIGURE 3.26 *(continued)*

`void insert(i, first, last)`	insert elements from location referenced by `first` to location referenced by `last` before the element referenced by iterator `i`; `first` and `last` are either `const_iterators` or `const` pointers
`size_type max_size() const`	return the maximum number of elements for the deque
`T& operator[]`	subscript operator
`void pop_back()`	remove the last element of the deque
`void pop_front()`	remove the first element of the deque
`void push_back(el)`	insert `el` at the end of the deque
`void push_front(el)`	insert `el` at the beginning of the deque
`reverse_iterator rbegin()`	return an iterator that references the last element of the deque
`const_reverse_iterator rbegin() const`	return a `const` iterator that references the last element of the deque
`reverse_iterator rend()`	return an iterator that is before the first element of the deque
`void resize(n, el = T())`	make the deque have n positions by adding `n - size()` more positions with element `el` or by discarding overflowing `size() - n` positions from the end of the deque
`reverse_iterator rend()`	return an iterator that is before the first element of the deque
`const_reverse_iterator rend() const`	return a `const` iterator that is before the first element of the deque
`size_type size() const`	return the number of elements in the deque
`void swap(dq)`	swap the content of the deque with the content of another deque `dq`

only autoincrement and autodecrement were possible for iterators, but for deques we can add any number to iterators. For example, `dq1.begin()+1` is legal for deques, but not for lists.

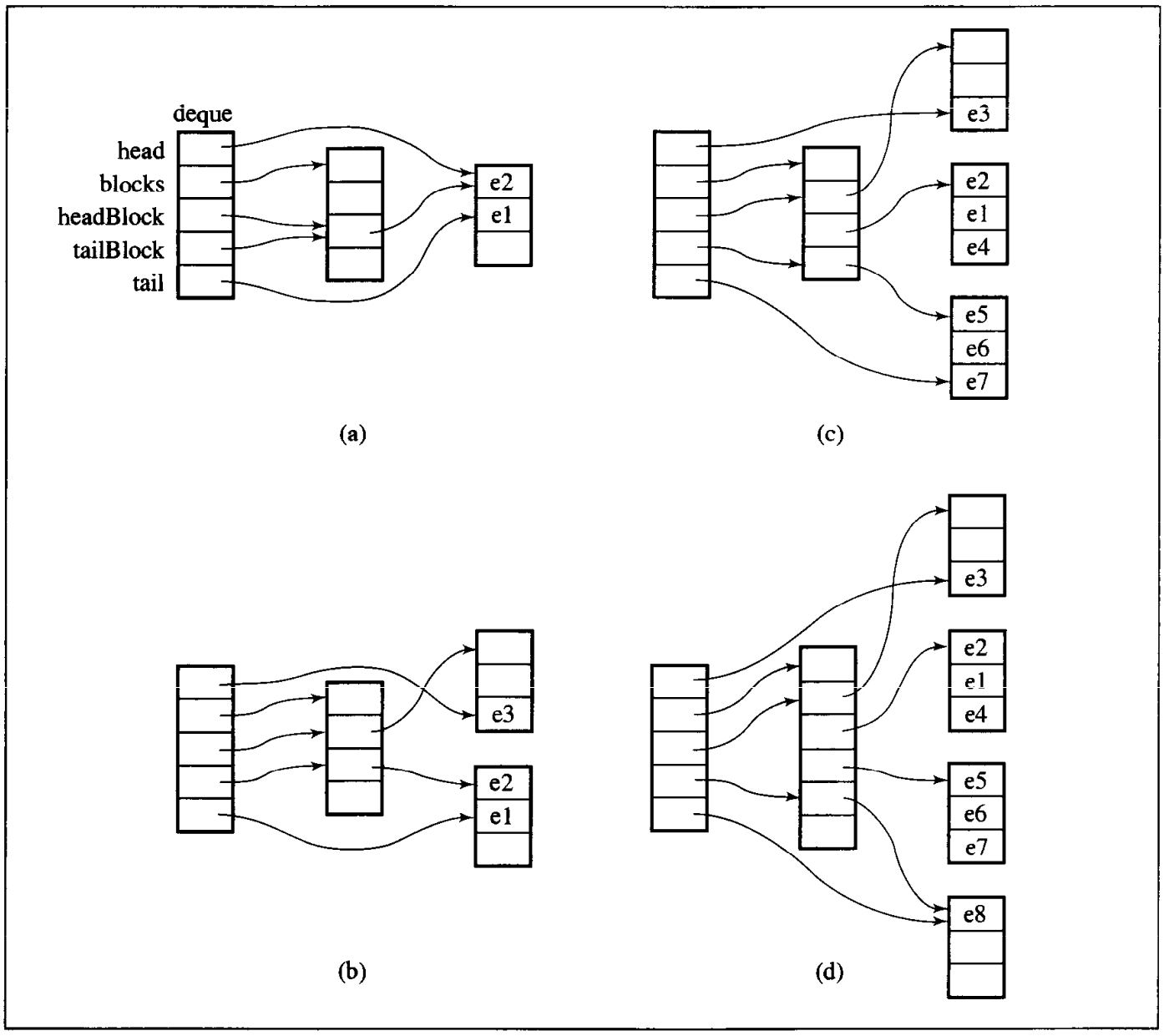
A very interesting aspect of the STL deque is its implementation. Random access can be simulated in doubly linked lists having in the definition of `operator[](int n)` a loop that scans sequentially the list and stops at the *n*th node. The STL implementation solves this problem differently. An STL deque is not implemented as a linked list but as an array of pointers to blocks or arrays of data. The number of blocks changes dynamically depending on storage needs, and the size of the array of pointers changes accordingly. (We encounter a similar approach applied in extendible hashing in Section 10.5.1.)

FIGURE 3.27 A program demonstrating the operation of deque member functions.

```
#include <iostream.h>
#include <algorithm>
#include <deque>

using namespace std;

void main() {
    deque<int> dq1;
    dq1.push_front(1);                              // dq1 = (1)
    dq1.push_front(2);                              // dq1 = (2 1)
    dq1.push_back(3);                               // dq1 = (2 1 3)
    dq1.push_back(4);                               // dq1 = (2 1 3 4)
    deque<int> dq2(dq1.begin()+1,dq1.end()-1); // dq2 = (1 3)
    dq1[1] = 5;                                     // dq1 = (2 5 3 4)
    dq1.erase(dq1.begin());                         // dq1 = (5 3 4)
    dq1.insert(dq1.end()-1,2,6);                    // dq1 = (5 3 6 6 4)
    sort(dq1.begin(),dq1.end());                    // dq1 = (3 4 5 6 6)
    deque<int> dq3;
    dq3.resize(dq1.size()+dq2.size());              // dq3 = (0 0 0 0 0 0 0)
    merge(dq1.begin(),dq1.end(),dq2.begin(),dq2.end(),dq3.begin());
    // dq1 = (3 4 5 6 6) and dq2 = (1 3) ==> dq3 = (1 3 3 4 5 6 6)
}
```

To discuss one possible implementation, assume that the array of pointers has four cells and an array of data has three cells, that is, `blockSize = 3`. An object deque includes fields `head`, `tail`, `headBlock`, `tailBlock`, and `blocks`. After execution of `push_front(e1)` and `push_front(e2)` with an initially empty deque, the situation is as in Figure 3.28a. First, the array `blocks` is created and then one data block accessible from a middle cell of `blocks`. Next, e1 is inserted in the middle of the data block. The subsequent calls place elements consecutively in the first half of the data array. The third call to `push_front()` cannot successfully place e3 in the current data array; therefore, a new data array is created and e3 is located in the last cell (Figure 3.28b). Now we execute `push_back()` four times. Element e4 is placed in an existing data array accessible from deque through `tailBlock`. Elements e5, e6, and e7 are placed in a new data block, which also becomes accessible through `tailBlock` (Figure 3.28c). The next call to `push_back()` affects the pointer array `blocks` because the last data block is full and the block is accessible for the last cell of `blocks`. In this case, a new pointer array is created which contains (in this implementation) twice as many cells as the number of data blocks. Next, the pointers from old array `blocks` are copied to the new array, and then a new data block can be created to accommodate

FIGURE 3.28 Changed on the deque in the process of pushing new elements.



element e8 being inserted (Figure 3.28d). This is an example of the worst case for which between n/`blockSize` and n/`blockSize` + 2 cells have to be copied from the old array to the new one; therefore, in the worst case, the pushing operation takes $O(n)$ time to perform. But assuming that `blockSize` is a large number, the worst case can be expected to occur very infrequently. Most of the time, the pushing operation requires constant time.

Inserting an element into a deque is very simple conceptually. To insert an element in the first half of the deque, the front element is pushed onto the deque, and all elements that should precede the new element are copied to the preceding cell. Then

the new element can be placed in the desired position. To insert an element into the second half of the deque, the last element is pushed onto the deque, and elements that should follow the new element in the deque are copied to the next cell.

With the discussed implementation, a random access can be performed in constant time. For the situation illustrated in Figure 3.28, that is, with declarations

```
T **blocks;
T **headBlock;
T *head;
```

the subscript operator can be overloaded as follows:

```
T& operator[] (int n) {
    if (n < blockSize - (head - *headBlock))    // if n is
        return * (head + n);                    // in the first
    else {                                      // block;
        n = n - (blockSize - (head - *headBlock));
        int q = n / blockSize + 1;
        int r = n % blockSize;
        return *(*(headBlock + q) + r);
    }
}
```

Although access to a particular position requires several arithmetic, dereferencing, and assignment operations, the number of operations is constant for any size of the deque.

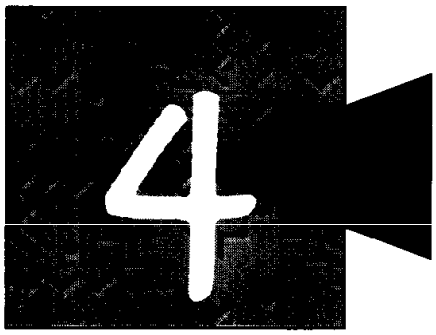
3.9 Concluding Remarks

Linked lists have been introduced to overcome limitations of arrays by allowing dynamic allocation of necessary amounts of memory. Also, linked lists allow easy insertion and deletion of information, since such operations have a local impact on the list. To insert a new element at the beginning of an array, all elements in the array have to be shifted to make room for the new item; hence, insertion has a global impact on the array. Deletion is the same. So should we always use linked lists instead of arrays?

Arrays have some advantages over linked lists, namely that they allow random accessing. To access the tenth node in a linked list, all nine preceding nodes have to be passed. In the array, we can go to the tenth cell immediately. Therefore, if an immediate access of any element is necessary, then an array is a better choice. This was the case with binary search and it will be the case with most sorting algorithms (see Chapter 9). But if we are constantly accessing only some elements—the first, the second, the last, and the like—and if changing the structure is the core of an algorithm, then using a linked list is a better option. A good example is a queue, which is discussed in the next chapter.

Another advantage in the use of arrays is space. To hold items in arrays, the cells have to be of the size of the items. In linked lists, we store one item per node and the

Stacks and Queues



As the first chapter explained, abstract data types allow us to delay the specific implementation of a data type until it is well understood what operations are required to operate on the data. In fact, these operations determine which implementation of the data type is most efficient in a particular situation. This situation is illustrated by two data types, stacks and queues, which are described by a list of operations. Only after the list of the required operations is determined do we present some possible implementations and compare them.

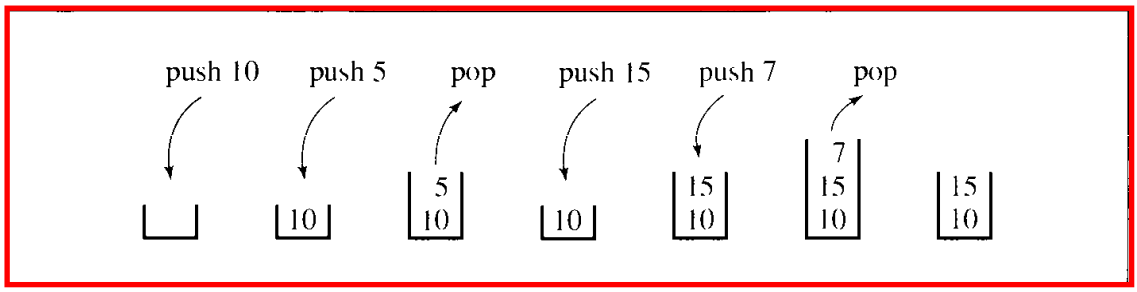
4.1 Stacks

A *stack* is a linear data structure which can be accessed only at one of its ends for storing and retrieving data. Such a stack resembles a stack of trays in a cafeteria: New trays are put on the top of the stack and taken off the top. The last tray put on the stack is the first tray removed from the stack. For this reason, a stack is called an *LIFO* structure: last in/first out.

A tray can be taken only if there are trays on the stack, and a tray can be added to the stack only if there is enough room, that is, if the stack is not too high. Therefore, a stack is defined in terms of operations which change its status and operations which check this status. The operations are as follows:

- *clear()*—Clear the stack.
- *isEmpty()*—Check to see if the stack is empty.
- *push(el)*—Put the element *el* on the top of the stack.
- *pop()*—Take the topmost element from the stack.
- *topEl()*—Return the topmost element in the stack without removing it.

FIGURE 4.1 A series of operations executed on a stack.



A series of push and pop operations is shown in Figure 4.1. After pushing number 10 onto an empty stack, the stack contains only this number. After pushing 5 on the stack, the number is placed on top of 10 so that, when the popping operation is executed, 5 is removed from the stack, because it arrived after 10, and 10 is left on the stack. After pushing 15 and then 7, the topmost element is 7, and this number is removed when executing the popping operation, after which the stack contains 10 at the bottom and 15 above it.

Generally, the stack is very useful in situations when data have to be stored and then retrieved in reverse order. One application of the stack is in matching delimiters in a program. This is an important example because delimiter matching is part of any compiler: No program is considered correct if the delimiters are mismatched.

In C++ programs, we have the following delimiters: parentheses '(' and ')', square brackets '[' and ']', curly brackets '{' and '}', and comment delimiters '/*' and '*/'. Here are examples of C++ statements that use delimiters properly:

```
a = b + (c - d) * (e - f);
g[10] = h[i[9]] + (j + k) * l;
while (m < (n[8] + o)) { p = 7; /* initialize p */ r = 6; }
```

These examples are statements in which mismatching occurs:

```
a = b + (c - d) * (e - f));
g[10] = h[i[9]] + j + k) * l;
while (m < (n[8) + o]) { p = 7; /* initialize p */ r = 6; }
```

A particular delimiter can be separated from its match by other delimiters; that is, delimiters can be nested. Therefore, a particular delimiter is matched up only after all the delimiters following it and preceding its match have been matched. For example, in the condition of the loop

```
while (m < (n[8] + o))
```

the first opening parenthesis must be matched with the last closing parenthesis, but this is done only after the second opening parenthesis is matched with the next to last closing parenthesis; this, in turn, is done after the opening square bracket is matched with the closing bracket.

The delimiter matching algorithm reads a character from a C++ program and stores it on a stack if it is an opening delimiter. If a closing delimiter is found, the delimiter is compared to a delimiter popped off the stack. If they match, processing continues. If not, processing discontinues by signaling an error. The processing of the C++ program ends successfully after the end of the program is reached and the stack is empty. Here is the algorithm:

```
delimiterMatching(file)
  read character ch from file;
   while not end of file
      if ch is '(', '[', or '{'
            push(ch);
      else if ch is '/'
            read the next character;
            if this character is '*'
                  push(ch);
            else ch = the character read in;
                  continue; // go to the beginning of the loop;
      else if ch is ')', ']', or '}'
            if ch and popped off delimiter do not match
               failure;
      else if ch is '*'
            read the next character;
            if this character is '/' and popped off delimiter is not '/'
                  failure;
            else ch = the character read in;
                  push back the popped off delimiter;
                  continue;
     // else ignore other characters;
         read next character ch from file;
   if stack is empty
        success;
   else failure;
```

Figure 4.2 shows the processing that occurs when applying this algorithm to the statement

```
s=t[5]+u/(v*(w+y));
```

The first column in Figure 4.2 shows the contents of the stack at the end of the loop before the next character is input from the program file. The first line shows the initial situation in the file and on the stack. Variable ch is initialized to the first character of the file, letter s, and in the first iteration of the loop, the character is simply ignored. This situation is shown in the second row in Figure 4.2. Then the next character, equal sign, is read. It is also ignored and so is the letter t. After reading the left bracket, the bracket is pushed onto the stack so that the stack now has one element, the left bracket. Reading digit 5 does not change the stack, but after the right bracket becomes the value of ch, the topmost element is popped off the stack and compared with ch.

FIGURE 4.2 Processing the statement s= t[5]+u / (v*(w+y)); with the algorithm `delimiterMatching()`.

Stack	Nonblank Character Read	Input Left
empty		s = t[5] + u / (v * (w + y));
empty	s	= t[5] + u / (v * (w + y));
empty	=	t[5] + u / (v * (w + y));
empty	t	[5] + u / (v * (w + y));
[	[	5] + u / (v * (w + y));
[	5	] + u / (v * (w + y));
empty	]	+ u / (v * (w + y));
empty	+	u / (v * (w + y));
empty	u	/ (v * (w + y));
empty	/	(v * (w + y));
(	(	v * (w + y));
(	v	* (w + y));
(	*	(w + y));
(		
(	(	w + y));
(		
(	w	+y));
(		
(	+	y));
(		
(	y	));
(	)	);
empty	)	;
empty	;	

Because the popped off element (left bracket) matches `ch` (right bracket), the processing of input continues. After reading and discarding the letter `u`, a slash is read and the algorithm checks whether it is part of the comment delimiter by reading the next character, a left parenthesis. Because the character read in is not an asterisk, the slash is not a beginning of a comment, so `ch` is set to left parenthesis. In the next iteration, this parenthesis is pushed onto the stack and processing continues, as shown in Figure 4.2. After reading the last character, a semicolon, the loop is exited and the stack is checked. Because it is empty (no unmatched delimiters are left), success is pronounced.

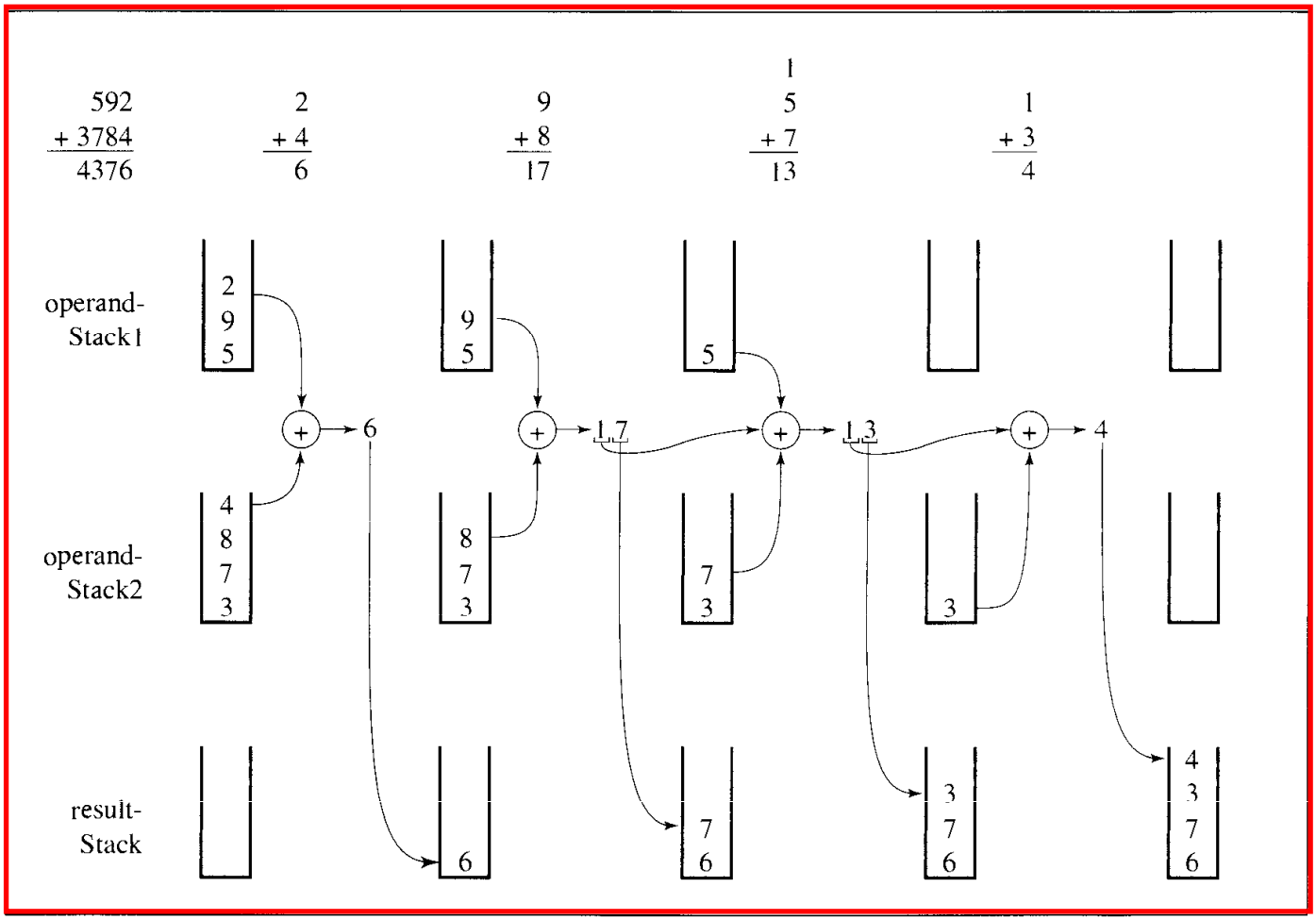
As another example of stack application, consider adding very large numbers. The largest magnitude of integers is limited, so we are not able to add 18,274,364,583,929,273,748,459,595,684,373 and 8,129,498,165,026,350,236, since integer variables cannot hold such large values, let alone their sum. The problem can be solved if we treat these numbers as strings of numerals, store the numbers corresponding to these numerals on two stacks, and then perform addition by popping numbers from the stacks. The pseudocode for this algorithm is as follows:

```
addingLargeNumbers()
    read the numerals of the first number and store the numbers corresponding to
    them on one stack;
    read the numerals of the second number and store the numbers corresponding
    to them on another stack;
    result = 0;
    while at least one stack is not empty
        pop a number from each nonempty stack and add them to result;
        push the unit part on the result stack;
        store carry in result;
    push carry on the result stack if it is not zero;
    pop numbers from the result stack and display them;
```

Figure 4.3 shows an example of application of this algorithm. In this example, numbers 592 and 3784 are added.

1. Numbers corresponding to digits composing the first number are pushed onto `operandStack1`, and numbers corresponding to digits of 3784 are pushed onto `operandStack2`. Note the order of digits on the stacks.
2. Numbers 2 and 4 are popped from the stacks, and the result, 6, is pushed onto `resultStack`.
3. Numbers 9 and 8 are popped from the stacks, and the unit part of their sum, 7, is pushed onto `resultStack`; the tens part of the result, number 1, is retained as a carry in the variable `result` for subsequent addition.
4. Numbers 5 and 7 are popped from the stacks, added to the carry, and the unit part of the result, 3, is pushed onto `resultStack`, and the carry, 1, becomes a value of the variable `result`.
5. One stack is empty, so a number is popped from the nonempty stack, added to carry, and the result is stored on `resultStack`.
6. Both operand stacks are empty, so the numbers from `resultStack` are popped and printed as the final result.

FIGURE 4.3 An example of adding numbers 592 and 3784 using stacks.



Consider now implementation of our abstract stack data structure. We used push and pop operations as though they were readily available, but they also have to be implemented as functions operating on the stack.

A natural implementation for a stack is a flexible array, that is, a vector. Figure 4.4 contains a generic stack class definition that can be used to store any type of objects. Also, a linked list can be used for implementation of a stack (Figure 4.5).

Figure 4.6 shows the same sequence of push and pop operations as Figure 4.1 with the changes that take place in the stack implemented as a vector (Figure 4.6b) and as a linked list (Figure 4.6c).

The linked list implementation matches the abstract stack more closely in that it includes only the elements that are on the stack because the number of nodes in the list is the same as the number of stack elements. In the vector implementation, the capacity of the stack can often surpass its size.

The vector implementation, like the linked list implementation, does not force the programmer to make a commitment at the beginning of the program concerning the size of the stack. If the size can be reasonably assessed in advance, then the predicted size can be used as a parameter for the stack constructor to create in advance a

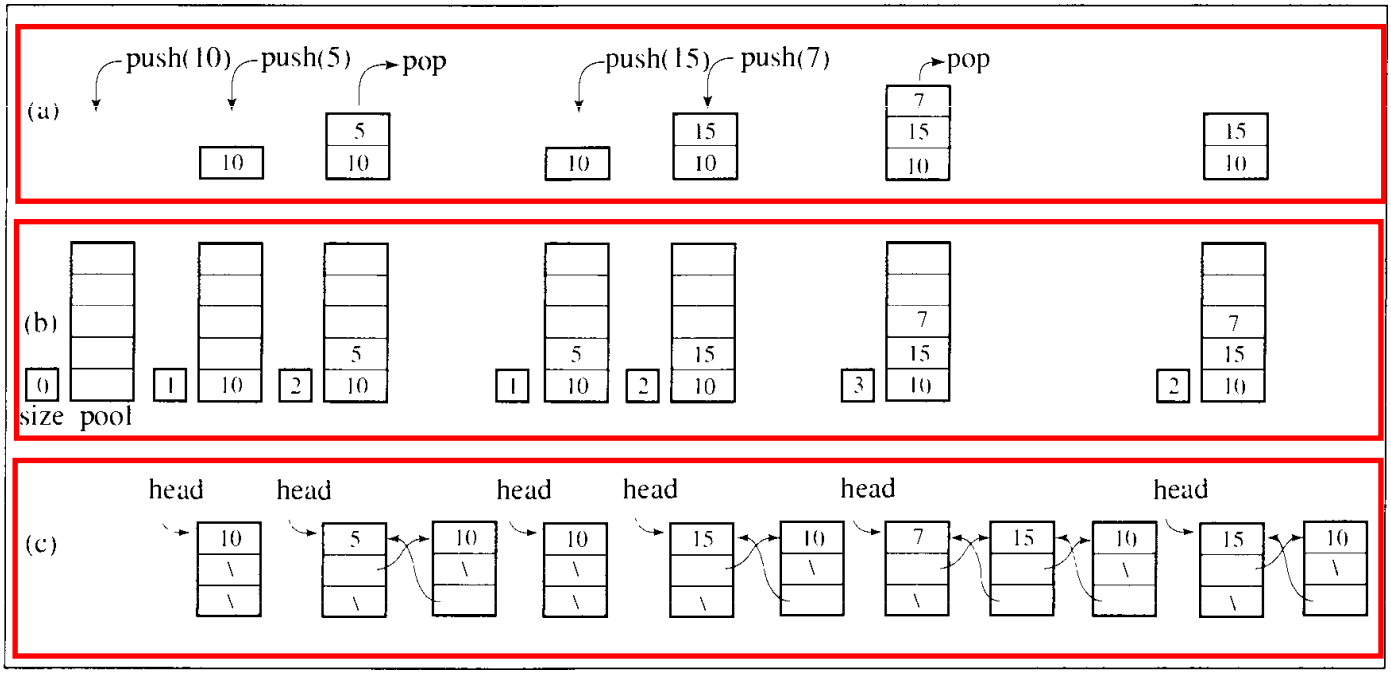
FIGURE 4.4 A vector implementation of a stack.

```
//*********************  genStack.h  *************************
//      generic class for vector implementation of stack

#ifndef STACK
#define STACK

#include <vector>

using namespace std;

template<class T, int capacity = 30>
class Stack {
public:
    Stack() {
        pool.reserve(capacity);
    }
    void clear() {
        pool.clear();
    }
    bool isEmpty() const {
        return pool.empty();
    }
    T& topEl() {
        return pool.back();
    }
    T pop() {
        T el = pool.back();
        pool.pop_back();
        return el;
    }
    void push(const T& el) {
        pool.push_back(el);
    }
private:
    vector<T> pool;
};

#endif
```

FIGURE 4.5 Implementing a stack as a linked list.

```
//*********************  genListStack.h  *************************
//     generic stack defined as a doubly linked list

#ifndef LL_STACK
#define LL_STACK

#include <list>

using namespace std;

template<class T>
class LLStack {
public:
    LLStack() {
    }
    void clear() {
        lst.clear();
    }
    bool isEmpty() const {
        return lst.empty();
    }
    T& topEl() {
        return lst.back();
    }
    T pop() {
        T el = lst.back();
        lst.pop_back();
        return el;
    }
    void push(const T& el) {
        lst.push_back(el);
    }
private:
    list<T> lst;
};

#endif
```

FIGURE 4.6 A series of operations executed on an abstract stack (a) and the stack implemented with an array (b) and with a linked list (c).



vector of the specified capacity. In this way, an overhead is avoided to copy the vector elements to a new larger location when pushing a new element to the stack for which size equals capacity.

It is easy to see that in the array and linked list implementations, popping and pushing are executed in constant time $O(1)$. However, in the vector implementation, pushing an element onto a full stack requires allocating more memory and copies the elements from the existing vector to a new vector. Therefore, in the worst case, pushing takes $O(n)$ time to finish.

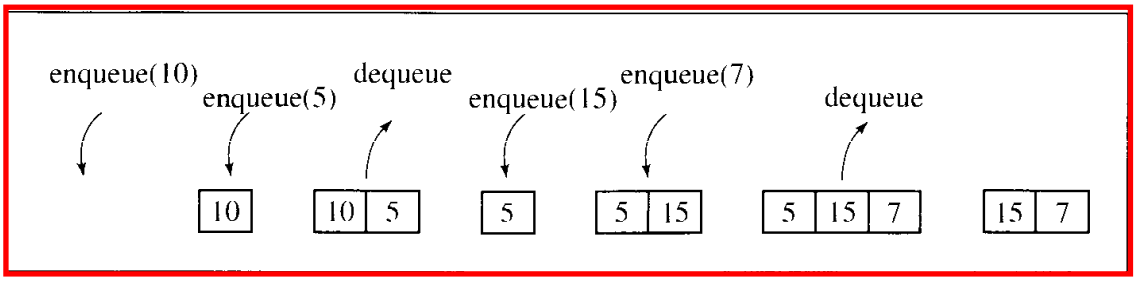
4.2 QUEUES

A *queue* is simply a waiting line that grows by adding elements to its end and shrinks by taking elements from its front. Unlike a stack, a queue is a structure in which both ends are used: one for adding new elements and one for removing them. Therefore, the last element has to wait until all elements preceding it on the queue are removed. A queue is an FIFO structure: first in/first out.

Queue operations are similar to stack operations. The following operations are needed to properly manage a queue:

- *clear()*—Clear the queue.
- *isEmpty()*—Check to see if the queue is empty.

FIGURE 4.7 A series of operations executed on a queue.



- *enqueue(el)*—Put the element *el* at the end of the queue.
- *dequeue()*—Take the first element from the queue.
- *firstEl()*—Return the first element in the queue without removing it.

A series of enqueue and dequeue operations is shown in Figure 4.7. This time—unlike for stacks—the changes have to be monitored both at the beginning of the queue and at the end. The elements are enqueued on one end and dequeued from the other. For example, after enqueuing 10 and then 5, the dequeue operation removes 10 from the queue (Figure 4.7).

For an application of a queue, consider the following poem written by Lewis Carroll:

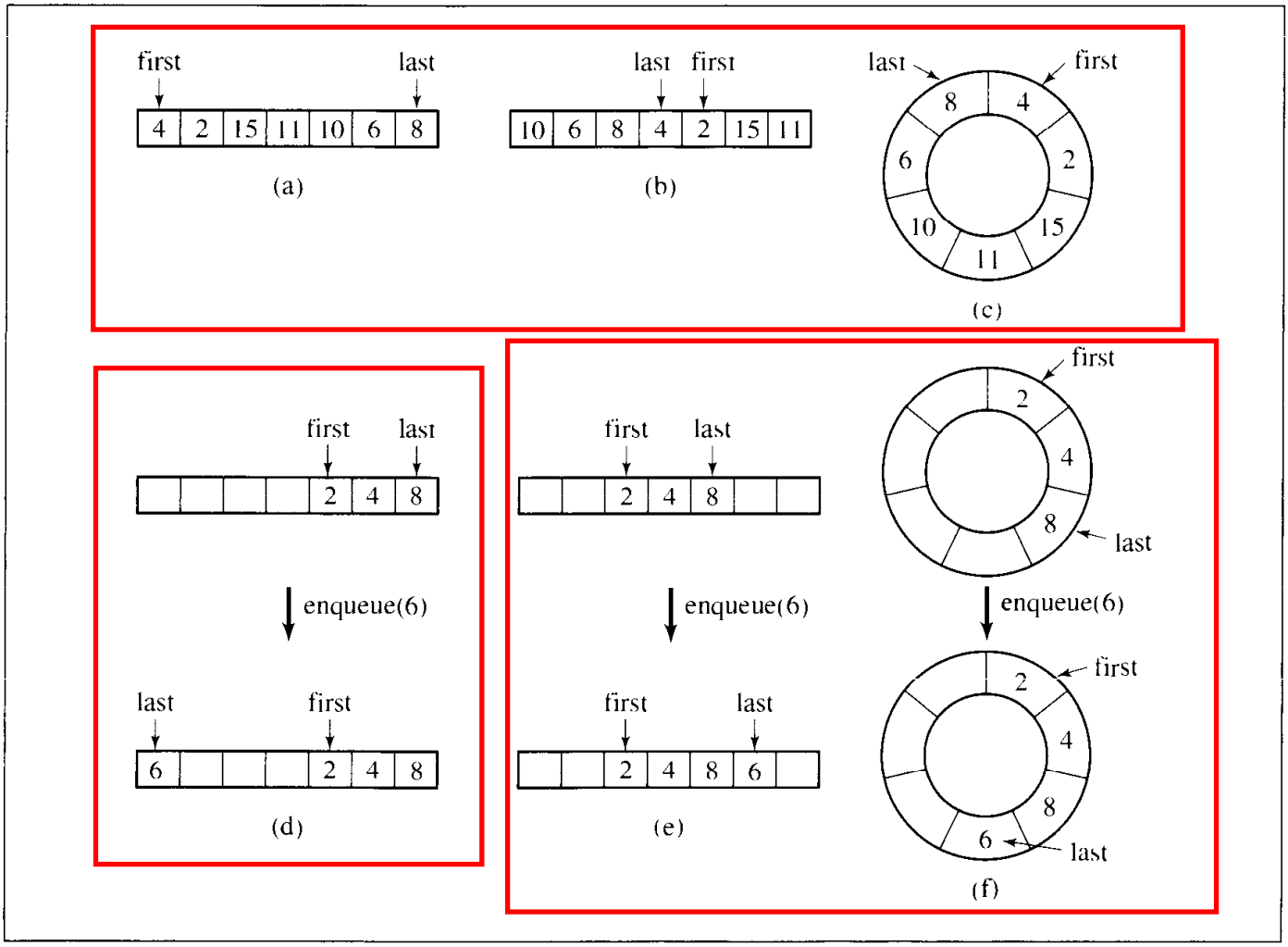
Round the wondrous globe I wander wild,
Up and down-hill—Age succeeds to youth—
Toiling all in vain to find a child
Half so loving, half so dear as Ruth.

The poem is dedicated to Ruth Dymes, which is indicated not only by the last word of the poem, but also by reading in sequence the first letters of each line, which also spells Ruth. This type of poem is called an acrostic and it is characterized by initial letters that form a word or phrase when taken in order. To see whether a poem is an acrostic, we devise a simple algorithm that reads a poem, echoprints it, retrieves and stores the first letter from each line on a queue, and after the poem is processed, all the stored first letters are printed in order. Here is an algorithm:

```
acrosticIndicator()
    while not finished
        read a line of poem;
        enqueue the first letter of the line;
        output the line;
    while queue is not empty
        dequeue and print a letter;
```

There is a more significant example to follow, but first consider the problem of implementation.

FIGURE 4.8 (a–b) Two possible configurations in an array implementation of a queue when the queue is full. (c) The same queue viewed as a circular array. (d) Enqueuing number 6 to a queue storing 2, 4, and 8. (e–f) The same queue seen as a one-dimensional array with the last element (e) at the end of the array and (f) in the middle.



One possible queue implementation is an array, although this may not be the best choice. Elements are added to the end of the queue, but they may be removed from its beginning, thereby releasing array cells. These cells should not be wasted. Therefore, they are utilized to enqueue new elements, whereby the end of the queue may occur at the beginning of the array. This situation is better pictured as a circular array as Figure 4.8c illustrates. The queue is full if the first element immediately precedes in the counterclockwise direction the last element. However, because a circular array is implemented with a "normal" array, the queue is full if either the first element is in the first cell and the last element is in the last cell (Figure 4.8a) or if the first element is right after the last (Figure 4.8b). Similarly, *enqueue()* and *dequeue()* have to consider the possibility of wrapping around the array when adding or removing elements. For example, *enqueue()* can be viewed as operating on a circular array (Figure 4.8c), but in reality, it is operating on a one-dimensional array. Therefore, if the last element is in

the last cell and if any cells are available at the beginning of the array, a new element is placed there (Figure 4.8e). If the last element is in any other position, then the new element is put after the last, space permitting (Figure 4.8f). These two situations must be distinguished when implementing a queue viewed as a circular array (Figure 4.8d).

Figure 4.9 contains possible implementations of member functions which operate on queues.

A more natural queue implementation is a doubly linked list as offered in the previous chapter and also in STL's `list` (Figure 4.10).

In both suggested implementations enqueuing and dequeuing can be executed in constant time $O(1)$ provided a doubly-linked list is used in the list implementation. In the singly linked list implementation, dequeuing requires $O(n)$ operations primarily to scan the list and stop at the next to last node (cf. discussion of `deleteFromTail()` in Section 3.1.2).

Figure 4.11 shows the same sequence of enqueue and dequeue operations as Figure 4.7 and indicates the changes in the queue implemented as an array (Figure 4.11b) and as a linked list (Figure 4.11c). The linked list keeps only the numbers that the logic of the queue operations indicated by Figure 4.11a requires. The array includes all the numbers until it fills up, after which new numbers are included starting from the beginning of the array.

Queues are frequently used in simulations to the extent that a well-developed and mathematically sophisticated theory of queues exists, called *queuing theory,* in which various scenarios are analyzed and models are built which use queues. In queuing processes there are a number of customers coming to servers to receive service. The throughput of the server may be limited. Therefore, customers have to wait in queues before they are served, and they spend some amount of time while they are being served. By customers, we mean not only people but also objects. For example, parts on an assembly line in the process of being assembled into a machine, trucks waiting for service at a weighing station on an interstate, or barges waiting for a sluice to be opened so they can pass through a channel also wait in queues. The most familiar examples are lines in stores, post offices, or banks. The type of problems posed in simulations are: How many servers are needed to avoid long queues? How large must the waiting space be to put the entire queue in it? Is it cheaper to increase this space or to open one more server?

As an example, consider Bank One which, over a period of 3 months, recorded the number of customers coming to the bank and the amount of time needed to serve them. The table in Figure 4.12a shows the number of customers who arrived during 1-minute intervals throughout the day. For 15% of such intervals, no customer arrived, for 20%, only one arrived, etc. Currently, six clerks are employed, no lines are ever observed, and the bank management wants to know whether six clerks are too many. Would five suffice? Four? Maybe even three? Can lines be expected at any time? To answer these questions, a simulation program is written which applies the recorded data and checks different scenarios.

The number of customers depends on the value of a randomly generated number between 1 and 100. The table in Figure 4.12a identifies five ranges of numbers from 1 to 100, based on the percentages of 1-minute intervals that had 0, 1, 2, 3, or 4 customers. If the random number is 21, then the number of customers is 1; if the random

FIGURE 4.9 Array implementation of a queue.

```
//********************  genArrayQueue.h  **********************
//             generic queue implemented as an array

#ifndef ARRAY_QUEUE
#define ARRAY_QUEUE

template<class T, int size = 100>
class ArrayQueue {
public:
    ArrayQueue() {
        first = last = -1;
    }
    void enqueue(T);
    T dequeue();
    bool isFull()  {
        return first == 0 && last == size-1 || first == last + 1;
    }
    bool isEmpty() {
        return first == -1;
    }
private:
    int first, last;
    T storage[size];
};

template<class T, int size>
void ArrayQueue<T,size>::enqueue(T el) {
   if (!isFull())
        if (last == size-1 || last == -1) {
            storage[0] = el;
            last = 0;
            if (first == -1)
                first = 0;
        }
        else storage[++last] = el;
   else cout << "Full queue.\n";
}

template<class T, int size>
T ArrayQueue<T,size>::dequeue() {
```

Continues

FIGURE 4.9 *(continued)*

```
    T tmp;
    tmp = storage[first];
    if (first == last)
         last = first = -1;
    else if (first == size-1)
         first = 0;
    else first++;
    return tmp;
}

#endif
```

FIGURE 4.10 Linked list implementation of a queue.

```
//*********************  genQueue.h  *************************
//     generic queue implemented with doubly linked list

#ifndef DLL_QUEUE
#define DLL_QUEUE

#include <list>

using namespace std;

template<class T>
class Queue {
public:
    Queue() {
    }
    void clear() {
        lst.clear();
    }
    bool isEmpty() const {
        return lst.empty();
    }
    T& front() {
```

FIGURE 4.10 *(continued)*

```
        return lst.front();
    }
    T dequeue() {
        T el = lst.front();
        lst.pop_front();
        return el;
    }
    void enqueue(const T& el) {
        lst.push_back(el);
    }
private:
    list<T> lst;
};

#endif
```

FIGURE 4.11 A series of operations executed on an abstract queue (a) and the stack implemented with an array (b) and with a linked list (c).

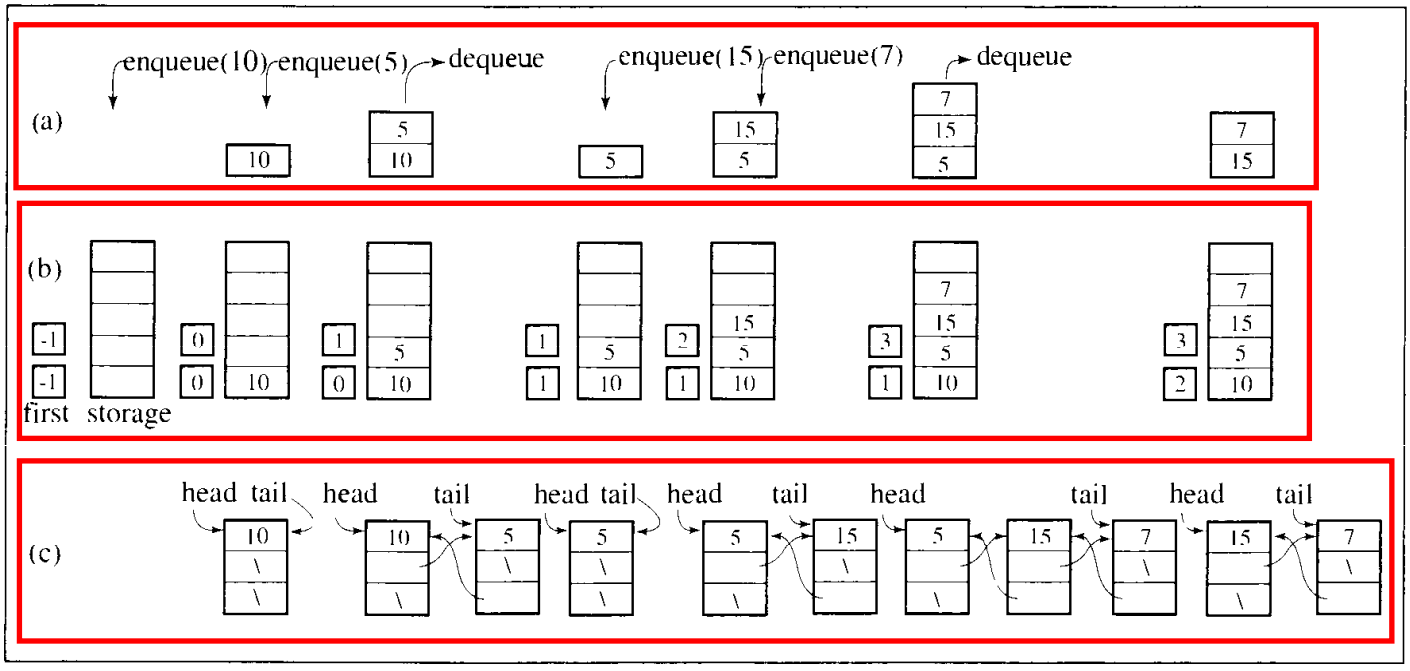


FIGURE 4.12 Bank One example: (a) data for number of arrived customers per 1-minute interval and (b) transaction time in seconds per customer.

Number of customers per minute	Percentage of one-minute intervals	Range
0	15	1–15
1	20	16–35
2	25	36–60
3	10	61–70
4	30	71–100

(a)

The amount of of time needed for service in seconds	Percentage of customers	Range
0	0	—
10	0	—
20	0	—
30	10	1–10
40	5	11–15
50	10	16–25
60	10	26–35
70	0	—
80	15	36–50
90	25	51–75
100	10	76–85
110	15	86–100

(b)

number is 90, then the number of customers is 4. This method simulates the rate of customers arriving at Bank One.

In addition, analysis of the recorded observations indicates that no customer required 10-second or 20-second transactions, 10% required 30 seconds, etc., as indicated in Figure 4.12b. The table in 4.12b includes ranges for random numbers to generate the length of a transaction in seconds.

Figure 4.13 contains the program simulating customer arrival and transaction time at Bank One. The program uses three arrays. `arrivals[]` records the percentages of 1-minute intervals depending on the number of the arrived customers. The array `service[]` is used to store the distribution of time needed for service. The amount of time is obtained by multiplying the index of a given array cell by 10. For example, `service[3]` is equal to 10, which means that 10% of the time a customer required $3 \cdot 10$ seconds for service. The array `clerks[]` records the length of transaction time in seconds.

For each minute (represented by the variable `t`), the number of arriving customers is randomly chosen, and for each customer, the transaction time is also

FIGURE 4.13 Bank One example: implementation code.

```
#include <iostream.h>
#include <stdlib.h>
#include "genQueue.h"

int option(int percents[]) {
    register int i = 0, choice = rand()%100+1, perc;
    for (perc = percents[0]; perc < choice; perc += percents[i+1], i++);
    return i;
}

void main() {
    int arrivals[] = {15,20,25,10,30};
    int service[] = {0,0,0,10,5,10,10,0,15,25,10,15};
    int clerks[] = {0,0,0,0}, numOfClerks = sizeof(clerks)/sizeof(int);
    int customers, t, i, numOfMinutes = 100, x;
    double maxWait = 0.0, currWait = 0.0, thereIsLine = 0.0;
    Queue<int> simulQ;
    cout.precision(2);
    for (t = 1; t <= numOfMinutes; t++) {
        cout << " t = " << t;
        for (i = 0; i < numOfClerks; i++)// after each minute subtract
            if (clerks[i] < 60)          // at most 60 seconds from time
                clerks[i] = 0;           // left to service the current
            else clerks[i] -= 60;        // customer by clerk i;
        customers = option(arrivals);
        for (i = 0; i < customers; i++) {// enqueue all new customers
            x = option(service)*10;      // (or rather service time
            simulQ.enqueue(x);           // they require);
            currWait += x;
        }
        // dequeue customers when clerks are available:
        for (i = 0; i < numOfClerks && !simulQ.isEmpty(); )
           if (clerks[i] < 60)        {
                x = simulQ.dequeue();   // assign more than one customer
                clerks[i] += x;         // to a clerk if service time
                currWait  -= x;         // is still below 60 sec;
           }
           else i++;
        if (!simulQ.isEmpty()) {
```

Continues

FIGURE 4.13 *(continued)*

```
            thereIsLine++;
            cout << " wait = " << currWait/60.0;
            if (maxWait < currWait)
                maxWait = currWait;
        }
        else cout << " wait = 0;";
        }
        cout << "\nFor " << numOfClerks << " clerks, there was a line "
             << thereIsLine/numOfMinutes*100.0 << "% of the time;\n"
             << "maximum wait time was " << maxWait/60.0 << " min.";
}
```

randomly determined. The function `option()` generates a random number, finds the range into which it falls, and then outputs the position, which is either the number of customers or a tenth the number of seconds.

Executions of this program indicate that six and five clerks are too many. With four clerks, service is performed smoothly; 25% of the time there is a short line of waiting customers. However, three clerks are always busy and there is always a long line of customers waiting. Bank management would certainly decide to employ four clerks.

4.3 PRIORITY QUEUES

In many situations, simple queues are inadequate, since first in/first out scheduling has to be overruled using some priority criteria. In a post office example, a handicapped person may have priority over others. Therefore, when a clerk is available, a handicapped person is served instead of someone from the front of the queue. On roads with tollbooths, some vehicles may be put through immediately, even without paying (police cars, ambulances, fire engines, etc.). In a sequence of processes, process P_2 may need to be executed before process P_1 for the proper functioning of a system, even though P_1 was put on the queue of waiting processes before P_2. In situations like these, a modified queue, or *priority queue*, is needed. In priority queues, elements are dequeued according to their priority and their current queue position.

The problem with a priority queue is in finding an efficient implementation which allows relatively fast enqueuing and dequeuing. Since elements may arrive randomly to the queue, there is no guarantee that the front elements will be the most likely to be dequeued and that the elements put at the end will be the last candidates for dequeuing. The situation is complicated because a wide spectrum of possible priority criteria can

be used in different cases such as frequency of use, birthday, salary, position, status, and others. It can also be the time of scheduled execution on the queue of processes, which explains the convention used in priority queue discussions in which higher priorities are associated with lower numbers indicating priority.

Priority queues can be represented by two variations of linked lists. In one type of linked list, all elements are entry ordered, and in another, order is maintained by putting a new element in its proper position according to its priority. In both cases, the total operational times are $O(n)$ because, for an unordered list, adding an element is immediate but searching is $O(n)$, and in a sorted list, taking an element is immediate but adding an element is $O(n)$.

Another queue representation uses a short ordered list and an unordered list, and a threshold priority is determined (Blackstone et al. 1981). The number of elements in the sorted list depends on a threshold priority. This means that in some cases this list can be empty and the threshold may change dynamically to have some elements in this list. Another way is to have always the same number of elements in the sorted list; the number $\sqrt{n}$ is a good candidate. Enqueuing takes on the average $O(\sqrt{n})$ time and dequeuing is immediate.

Another implementation of queues was proposed by J. O. Hendriksen (1977, 1983). It uses a simple linked list with an additional array of pointers to this list to find a range of elements in the list in which a newly arrived element should be included.

Experiments by Douglas W. Jones (1986) indicate that a linked list implementation, in spite of its $O(n)$ efficiency, is best for ten elements or less. The efficiency of the two-list version depends greatly on the distribution of priorities, and it may be excellent or as poor as that of the simple list implementation for large numbers of elements. Hendriksen's implementation, with its $O(\sqrt{n})$ complexity, operates consistently well with queues of any size.

4.4 Stacks in the Standard Template Library

A generic stack class is implemented in the STL as a container adaptor: It uses a container to make it behave in a specified way. The stack container is not created anew; it is an adaptation of an already existing container. By default, `deque` is the underlying container, but the user can also choose either `list` or `vector` with the following declarations:

```
stack<int> stack1;                  // deque by default
stack<int,vector<int> > stack2;     // vector
stack<int,list<int> > stack3;       // list
```

Member functions in the container `stack` are listed in Figure 4.14. Note that the return type of `pop()` is void; that is, `pop()` does not return a popped off element. To have access to the top element, the member function `top()` has to be used. Therefore, the popping operation discussed in this chapter has to be implemented with a call to `top()` followed by the call to `pop()`. Because popping operations in user programs are intended for capturing the popped off element most of the time and not only for

FIGURE 4.14 A list of `stack` member functions.

Member Function	Operation
`bool empty() const`	return `true` if the stack includes no element and `false` otherwise
`void pop()`	remove the top element of the stack
`void push(el)`	insert `el` at the top of the stack
`size_type size() const`	return the number of elements on the stack
`stack()`	create an empty stack
`T& top()`	return the top element on the stack
`const T& top() const`	return the top element on the stack

removing it, the desired popping operation is really a sequence of the two member functions from the container `stack`. To contract them to one operation, a new class can be created that inherits all operations from `stack` and redefines `pop()`. This is a solution used in the case study at the end of the chapter.

4.5 Queues in the Standard Template Library

The queue container is implemented by default as the container `deque`, and the user may opt for using the container `list` instead. An attempt to use the container `vector` results in a compilation error because `pop()` is implemented as a call to `pop_front()`, which is assumed to be a member function of the underlying container and `vector` does not include such a member function. For the list of queue's member functions, see Figure 4.15. A short program in Figure 4.16 illustrates the operations of the member functions. Note that the dequeueing operation discussed in this chapter is implemented by `front()` followed by `pop()`, and the enqueueing operation is implemented with the function `push()`.

4.6 Priority Queues in the Standard Template Library

The `priority_queue` container (Figure 4.17) is implemented with the container `vector` by default, and the user may choose the container `deque`. `priority_queue` maintains an order in the queue by keeping an element with the highest priority in

FIGURE **4.15** A list of queue member functions.

Member Function	Operation
`T& back()`	return the last element in the queue
`const T& back() const`	return the last element in the queue
`bool empty() const`	return `true` if the queue includes no element and `false` otherwise
`T& front()`	return the first element in the queue
`const T& front() const`	return the first element in the queue
`void pop()`	remove the first element in the queue
`void push(el)`	insert `el` at the end of the queue
`queue()`	create an empty queue
`size_type size() const`	return the number of elements in the queue

FIGURE **4.16** An example application of queue's member functions.

```
#include <iostream>
#include <queue>

using namespace std;

void main() {
    queue<int> q1;
    queue<int,list<int> > q2; //leave space between angle brackets > >
    q1.push(1); q1.push(2); q1.push(3);
    q2.push(4); q2.push(5); q2.push(6);
    q1.push(q2.back());
    while (!q1.empty()) {
        cout << q1.front() << ' ';    // 1 2 3 6
        q1.pop();
    }
    while (!q2.empty()) {
        cout << q2.front() << ' ';    // 4 5 6
        q2.pop();
    }
}
```

FIGURE 4.17 A list of `priority_queue` member functions.

Member Function	Operation
`bool empty() const`	return `true` if the queue includes no element and `false` otherwise
`void pop()`	remove an element in the queue with the highest priority
`void push(el)`	insert `el` in a proper location on the priority queue
`priority_queue(f())`	create an empty priority queue that uses a two-argument Boolean function `f` to order elements on the queue
`priority_queue(first, last, f())`	create a priority queue that uses a two-argument Boolean function `f` to order elements on the queue; initialize the queue with elements from the range indicated by iterators first and last
`size_type size() const`	return the number of elements in the priority queue
`T& top()`	return the element in the priority queue with the highest priority
`const T& top() const`	return the element in the priority queue with the highest priority

front of the queue. To accomplish this, a two-argument Boolean function is used by the insertion operation `push()`, which reorders the elements in the queue to satisfy this requirement. The function can be supplied by the user; otherwise, the operation < is used and the element with the highest value is considered to have the highest priority. If the highest priority is determined by the smallest value, then the function object `greater` needs to be used to indicate that `push()` should apply the operator > rather than < in making its decisions when inserting new elements to the priority queue. An example is shown in Figure 4.18. The priority `pq1` is defined as a vector-based queue that uses the operation < to determine the priority of integers in the queue. The second queue, `pq2`, uses the operation > during insertion. Finally, the queue `pq3` is of the same type as `pq1`, but it is also initialized with the numbers from the array `a`. The three `while` loops show in which order the elements from the three queues are dequeued.

It is more interesting to see an application of the user-defined objects. Consider the class `Person` defined in Section 1.8:

```
class Person {
public:
    . . . . .
    bool operator<(const Person& p) const {
        return strcmp(name,p.name) < 0;
```

FIGURE 4.18 A program that uses member functions of the container `priority_queue`.

```
#include <iostream>
#include <queue>
#include <functional>

using namespace std;

void main() {
    priority_queue<int> pq1; // plus vector<int> and less<int>
    priority_queue<int,vector<int>,greater<int> > pq2;
    pq1.push(3); pq1.push(1); pq1.push(2);
    pq2.push(3); pq2.push(1); pq2.push(2);
    int a[] = {4,6,5};
    priority_queue<int> pq3(a,a+3);
    while (!pq1.empty()) {
        cout << pq1.top() << ' ';    // 3 2 1
        pq1.pop();
    }
    while (!pq2.empty()) {
        cout << pq2.top() << ' ';    // 1 2 3
        pq2.pop();
    }
    while (!pq3.empty()) {
        cout << pq3.top() << ' ';    // 6 5 4
        pq3.pop();
    }
}
```

```
        }
        bool operator>(const Person& p) const {
            return !(*this == p) && !(*this < p);
        }
    private:
        char *name;
        int age;
    };
```

Our intention now is to create three priority queues. In the first two queues, the priority is determined by lexicographical order, but in `pqName1` it is the descending order and in `pqName2` the ascending order. To that end, `pqName1` uses the overloaded operator <. The queue `pqName2` uses the overloaded operator > as made known by defining with the function object `greater<Person>`:

```
Person p[] = {Person("Gregg",25),Person("Ann",30),Person("Bill",20)};
priority_queue<Person> pqName1(p,p+3);
priority_queue<Person,vector<Person>,greater<Person> > pqName2(p,p+3);
```

In these two declarations, the two priority queues are also initialized with objects from the array p.

In Section 1.8, there is also a Boolean function `lesserAge` used to determine the order of `Person` objects by age, not by name. How can we create a priority queue in which the highest priority is determined by age? One way to accomplish this is to define a function object,

```
class lesserAge {
public:
bool operator()(const Person& p1, const Person& p2) const {
   return p1.age < p2.age;
}
};
```

and then declare a new priority queue

```
priority_queue<Person,vector<Person>,lesserAge> pqAge(p,p+3);
```

initialized with the same objects as `pqName1` and `pqName2`. Printing elements from the three queues indicates the different priorities of the objects in different queues:

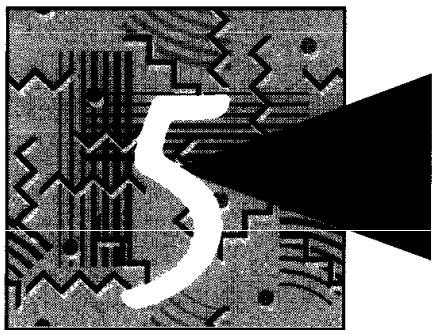
```
pqName1:   (Gregg,25) (Bill,20) (Ann,30)
pqName2:   (Ann,30) (Bill,20) (Gregg,25)
pqAge:     (Ann,30) (Gregg,25) (Bill,20)
```

4.7 Case Study: Exiting a Maze

Consider the problem of a trapped mouse that tries to find its way to an exit in a maze (Figure 4.19a). The mouse hopes to escape from the maze by systematically trying all the routes. If it reaches a dead end, it retraces its steps to the last position and begins at least one more untried path. For each position, the mouse can go in one of four directions: right, left, down, up. Regardless of how close it is to the exit, it always tries the open paths in this order, which may lead to some unnecessary detours. By retaining information that allows for resuming the search after a dead end is reached, the mouse uses a method called *backtracking*. This method is discussed further in the next chapter.

The maze is implemented as a two-dimensional character array in which passages are marked with 0s, walls by 1s, exit position by the letter e, and the initial position of the mouse by the letter m (Figure 4.19b). In this program, the maze problem is slightly generalized by allowing the exit to be in any position of the maze (picture the exit position as having an elevator that takes the mouse out of trap) and allowing passages to be on the borderline. To protect itself from falling off the array by trying to continue its path when an open cell is reached on one of the borderlines, the mouse also has to constantly check whether it is in such a borderline position or not. To avoid it, the program automatically puts a frame of 1s around the maze entered by the user.

Recursion



5.1 Recursive Definitions

One of the basic rules for defining new objects or concepts is that the definition should contain only such terms that have already been defined or that are obvious. Therefore, an object which is defined in terms of itself is a serious violation of this rule—a vicious circle. On the other hand, there are many programming concepts that define themselves. As it turns out, formal restrictions imposed on definitions such as existence and uniqueness are satisfied and no violation of the rules takes place. Such definitions are called *recursive definitions* and are used primarily to define infinite sets. When defining such a set, giving a complete list of elements is impossible, and for large finite sets, it is inefficient. Thus, a more efficient way has to be devised to determine if an object belongs to a set.

A recursive definition consists of two parts. In the first part, called the *anchor* or the *ground case*, the basic elements that are the building blocks of all other elements of the set are listed. In the second part, rules are given that allow for the construction of new objects out of basic elements or objects that have already been constructed. These rules are applied again and again to generate new objects. For example, to construct the set of natural numbers, one basic element, 0 is singled out, and the operation of incrementing by 1 is given as:

1. $0 \in \mathbf{N}$;
2. if $n \in \mathbf{N}$, then $(n + 1) \in \mathbf{N}$;
3. there are no other objects in the set **N**.

(More axioms are needed to ensure that only the set that we know as the natural numbers can be constructed by these rules.)

According to these rules, the set of natural numbers **N** consists of the following items: 0, 0 + 1, 0 + 1 + 1, 0 + 1 + 1 + 1, etc. Although the set **N** contains objects (and only such objects) that we call natural numbers, the definition results in a somewhat unwieldy list of elements. Can you imagine doing arithmetic on large numbers using such a specification? Therefore, it is more convenient to use the following definition, which encompasses the whole range of Arabic numeric heritage:

1. $0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \in \mathbf{N}$;
2. if $n \in \mathbf{N}$, then $n0, n1, n2, n3, n4, n5, n6, n7, n8, n9 \in \mathbf{N}$;
3. these are the only natural numbers.

Then the set **N** includes all possible combinations of the basic building blocks 0 through 9.

Recursive definitions serve two purposes: *generating* new elements, as already indicated, and *testing* whether or not an element belongs to a set. In the case of testing, the problem is solved by reducing it to a simpler problem, and if the simpler problem is still too complex it is reduced to an even simpler problem, and so on, until it is reduced to a problem indicated in the anchor. For instance, is 123 a natural number? According to the second condition of the definition introducing the set **N**, $123 \in \mathbf{N}$ if $12 \in \mathbf{N}$ and the first condition already says that $3 \in \mathbf{N}$; but $12 \in \mathbf{N}$ if $1 \in \mathbf{N}$ and $2 \in \mathbf{N}$, and they both belong to **N**.

The ability to decompose a problem into simpler subproblems of the same kind is sometimes a real blessing, as we shall see in the discussion of quicksort in a later chapter, or a curse, as we shall see shortly in this chapter.

Recursive definitions are frequently used to define functions and sequences of numbers. For instance, the factorial function, !, can be defined in the following manner:

$$n! = \begin{cases} 1 & \text{if } n = 0 \text{ (anchor)} \\ n \cdot (n-1)! & \text{if } n > 0 \text{ (inductive step)} \end{cases}$$

Using this definition, we can generate the sequence of numbers

$$1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, \ldots$$

which includes the factorials of the numbers $0, 1, 2, \ldots, 10, \ldots$

Another example is the definition

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ f(n-1) + \dfrac{1}{f(n-1)} & \text{if } n > 0 \end{cases}$$

which generates the sequence of rational numbers

$$1, 2, \frac{5}{2}, \frac{29}{10}, \frac{941}{290}, \frac{969581}{272890}, \ldots$$

Recursive definitions of sequences have one undesirable feature: To determine the value of an element s_n of a sequence, we first have to compute the values of some or all of the previous elements, $s_1, \ldots, s_{n-1}$. For example, calculating the value of 3! requires us to first compute the values of 0!, 1!, and 2!. Computationally, this is undesirable

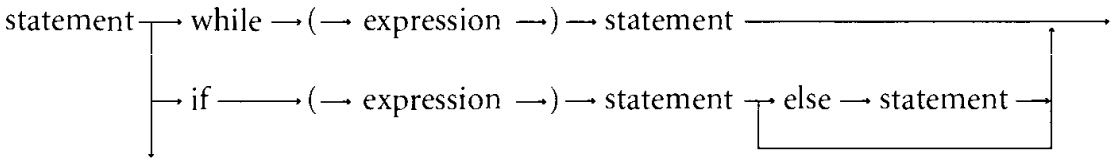
since it forces us to make calculations in a roundabout way. Therefore, we want to find an equivalent definition or formula that makes no references to other elements of the sequence. Generally, finding such a formula is a difficult problem that cannot always be solved. But the formula is preferable to a recursive definition because it simplifies the computational process and allows us to find the answer for an integer n without computing the values for integers $0, 1, \ldots, n-1$. For example, a definition of the sequence g,

$$g(n) = \begin{cases} 1 & \text{if } n = 0 \\ 2 \cdot g(n-1) & \text{if } n > 0 \end{cases}$$

can be converted into the simple formula

$$g(n) = 2^n$$

In the foregoing discussion, recursive definitions have been dealt with only theoretically, as a definition used in mathematics. Naturally, our interest is in computer science. One area where recursive definitions are used extensively is in the specification of the grammars of programming languages. Every programming language manual contains—either as an appendix or throughout the text—a specification of all valid language elements. Grammar is specified either in terms of block diagrams or in terms of the Backus-Naur form (BNF). For example, the syntactic definition of a statement in the C++ language can be presented in the block diagram form:



or in BNF:

```
<statement> ::= while (<expression>) <statement> |
                if (<expression>) <statement> |
                if (<expression>) <statement>else<statement> |
                ...
```

The language element <statement> is defined recursively, in terms of itself. Such definitions naturally express the possibility of creating such syntactic constructs as nested statements or expressions.

Recursive definitions are also used in programming. The good news is that virtually no effort is needed to make the transition from a recursive definition of a function to its implementation in C++. We simply make a translation from the formal definition into C++ syntax. Hence, for example, a C++ equivalent of factorial is the function

```
unsigned int factorial (unsigned int n) {
  if (n == 0)
       return 1;
  else return n * factorial (n - 1);
}
```

The problem now seems to be more critical since it is far from clear how a function calling itself can possibly work, let alone return the correct result. This chapter shows that it is possible for such a function to work properly. Recursive definitions on most computers are eventually implemented using a run-time stack, although the whole work of implementing recursion is done by the operating system, and the source code includes no indication of how it is performed. E. W. Dijkstra introduced the idea of using a stack to implement recursion. To better understand recursion and to see how it works, it is necessary to discuss the processing of function calls and to look at operations carried out by the system at function invocation and function exit.

5.2 Function Calls and Recursion Implementation

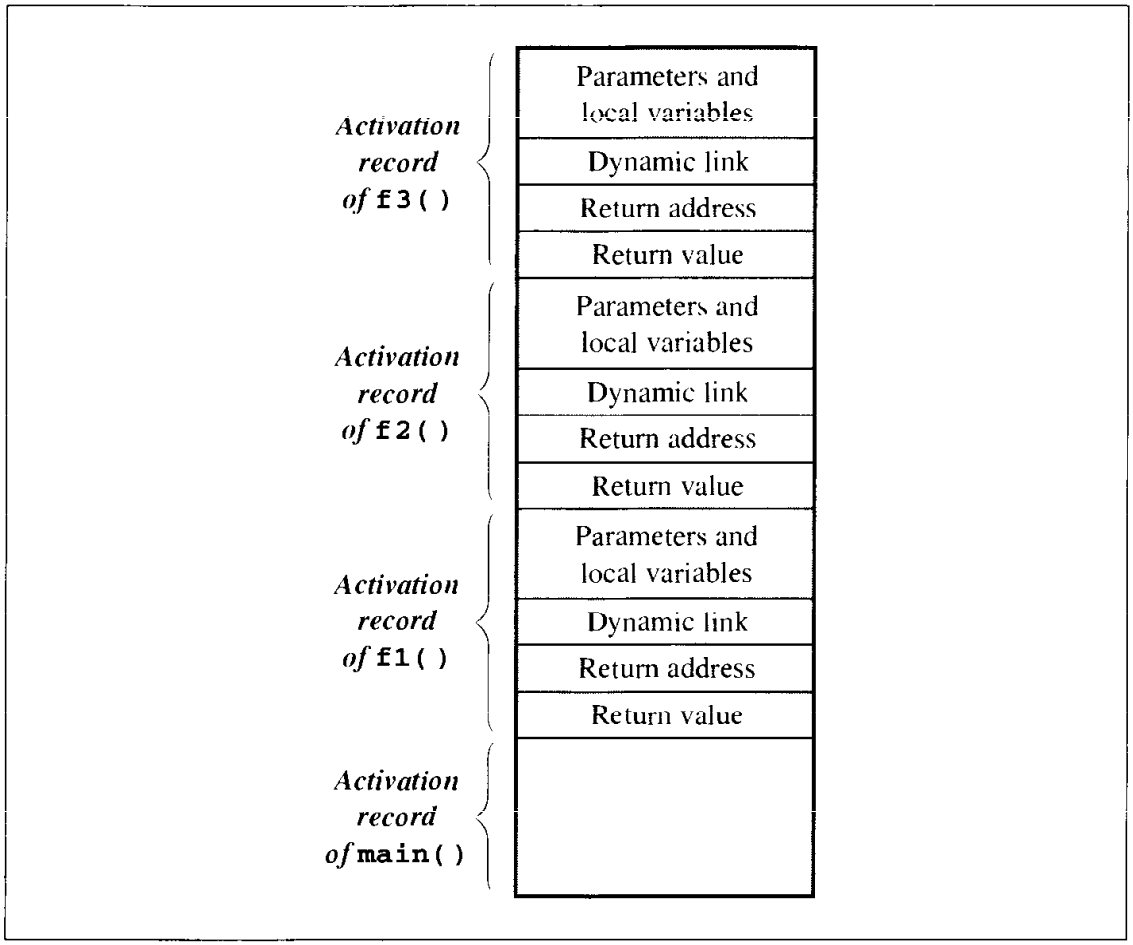
What happens when a function is called? If the function has formal parameters, they have to be initialized to the values passed as actual parameters. In addition, the system has to know where to resume execution of the program after the function has finished. The function can be called by other functions or by the main program (the function `main()`). The information indicating where it has been called from has to be remembered by the system. This could be done by storing the return address in main memory in a place set aside for return addresses, but we do not know in advance how much space might be needed, and allocating too much space for that purpose alone is not efficient.

For a function call, more information has to be stored than just a return address. Therefore, dynamic allocation using the run-time stack is a much better solution. But what information should be preserved when a function is called? First, automatic (local) variables must be stored. If function `f1()` which contains a declaration of an automatic variable `x` calls function `f2()` which locally declares the variable `x`, the system has to make a distinction between these two variables `x`. If `f2()` uses a variable `x`, then its own `x` is meant; if `f2()` assigns a value to `x`, then `x` belonging to `f1()` should be left unchanged. When `f2()` is finished, `f1()` can use the value assigned to its private `x` before `f2()` was called. This is especially important in the context of the present chapter, when `f1()` is the same as `f2()`, when a function calls itself recursively. How does the system make a distinction between these two variables `x`?

The state of each function, including `main()`, is characterized by the contents of all automatic variables, by the values of the function's parameters, and by the return address indicating where to restart its caller. The data area containing all this information is called an *activation record* or *stack frame* and is allocated on the run-time stack. An activation record exists for as long as a function owning it is executing. This record is a private pool of information for the function, a repository that stores all information necessary for its proper execution and how to return to where it was called from. Activation records usually have a short lifespan because they are dynamically allocated at function entry and deallocated upon exiting. Only the activation record of `main()` outlives every other activation record.

An activation record usually contains the following information:

FIGURE 5.1 Contents of the run-time stack when `main()` calls function `f1()`, `f1()` calls `f2()`, and `f2()` calls `f3()`.



- Values for all parameters to the function, location of the first cell if an array is passed or a variable is passed by reference, and copies of all other data items.
- Local (automatic) variables which can be stored elsewhere, in which case, the activation record contains only their descriptors and pointers to the locations where they are stored.
- The return address to resume control by the caller, the address of the caller's instruction immediately following the call.
- A dynamic link, which is a pointer to the caller's activation record.
- The returned value for a function not declared as `void`. Since the size of the activation record may vary from one call to another, the returned value is placed right above the activation record of the caller.

As mentioned above, if a function is called either by `main()` or by another function, then its activation record is created on the run-time stack. The run-time stack always reflects the current state of the function. For example, suppose that `main()` calls function `f1()`, `f1()` calls `f2()`, and `f2()` in turn calls `f3()`. If `f3()` is being executed, then the state of the run-time stack is as shown in Figure 5.1. By the nature of

the stack, if the activation record for `f3()` is popped by moving the stack pointer right below the return value of `f3()`, then `f2()` resumes execution and now has free access to the private pool of information necessary for reactivation of its execution. On the other hand, if `f3()` happens to call another function `f4()`, then the run-time stack increases its height since the activation record for `f4()` is created on the stack and the activity of `f3()` is suspended.

Creating an activation record whenever a function is called allows the system to handle recursion properly. Recursion is calling a function that happens to have the same name as the caller. Therefore, a recursive call is not literally a function calling itself, but rather an instantiation of a function calling another instantiation of the same original. These invocations are represented internally by different activation records and are thus differentiated by the system.

5.3 Anatomy of a Recursive Call

The function that defines raising any number x to a nonnegative integer power n is a good example of a recursive function. The most natural definition of this function is given by:

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot x^{n-1} & \text{if } n > 0 \end{cases}$$

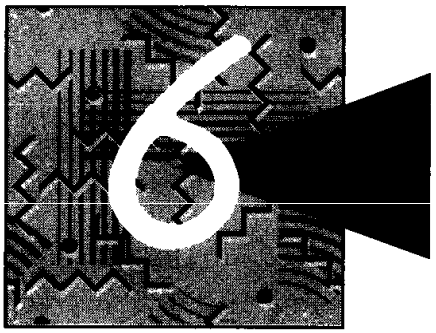
A C++ function for computing x^n can be written directly from the definition of a power:

```
/* 102 */ double power (double x, unsigned int n) {
/* 103 */     if (n == 0)
/* 104 */         return 1.0;
          // else
/* 105 */         return x * power(x,n-1);
          }
```

Using this definition, the value of x^4 can be computed in the following way:

$$\begin{aligned} x^4 = x \cdot x^3 = x \cdot (x \cdot x^2) = x \cdot (x \cdot (x \cdot x^1)) = x \cdot (x \cdot (x \cdot (x \cdot x^0))) \\ = x \cdot (x \cdot (x \cdot (x \cdot 1))) = x \cdot (x \cdot (x \cdot (x))) = x \cdot (x \cdot (x \cdot x)) \\ = x \cdot (x \cdot x \cdot x) = x \cdot x \cdot x \cdot x \end{aligned}$$

The repetitive application of the inductive step eventually leads to the anchor which is the last step in the chain of recursive calls. The anchor produces 1 as a result of raising x to the power of zero; the result is passed back to the previous recursive call. Now, that call, whose execution has been pending, returns its result, $x \cdot 1 = x$. The third call, which has been waiting for this result, computes its own result, namely, $x \cdot x$, and returns it. Next, this number $x \cdot x$ is received by the second call which multiplies it by x and returns the result, $x \cdot x \cdot x$, to the first invocation of `power()`. This call



Binary Trees

6.1 Trees, Binary Trees, and Binary Search Trees

Linked lists usually provide greater flexibility than arrays, but they are linear structures and it is difficult to use them to organize a hierarchical representation of objects. Although stacks and queues reflect some hierarchy, they are limited to only one dimension. To overcome this limitation, we create a new data type called a *tree* that consists of *nodes* and *arcs*. Unlike natural trees, these trees are depicted upside down with the *root* at the top and the *leaves* at the bottom. The root is a node that has no parent; it can have only child nodes. Leaves, on the other hand, have no children, or rather, their children are empty structures. A tree can be defined recursively as the following:

1. An empty structure is an empty tree.
2. If $t_1, \ldots, t_k$ are disjoint trees, then the structure whose root has as its children the roots of $t_1, \ldots, t_k$ is also a tree.
3. Only structures generated by rules 1 and 2 are trees.

Figure 6.1 contains examples of trees. Each node has to be reachable from the root through a unique sequence of arcs, called a *path*. The number of arcs in a path is called the *length* of the path. The *level* of a node is the length of the path from the root to the node plus 1, which is the number of nodes in the path. The *height* of a nonempty tree is the maximum level of a node in the tree. The empty tree is a legitimate tree of height 0 (by definition), and a single node is a tree of height 1. This is the only case in which a node is both the root and a leaf. The level of a node must be between 1 (the level of the root) and the height of the tree, which in the extreme case is the level of the only leaf in a degenerate tree resembling a linked list.

FIGURE 6.1 Examples of trees.

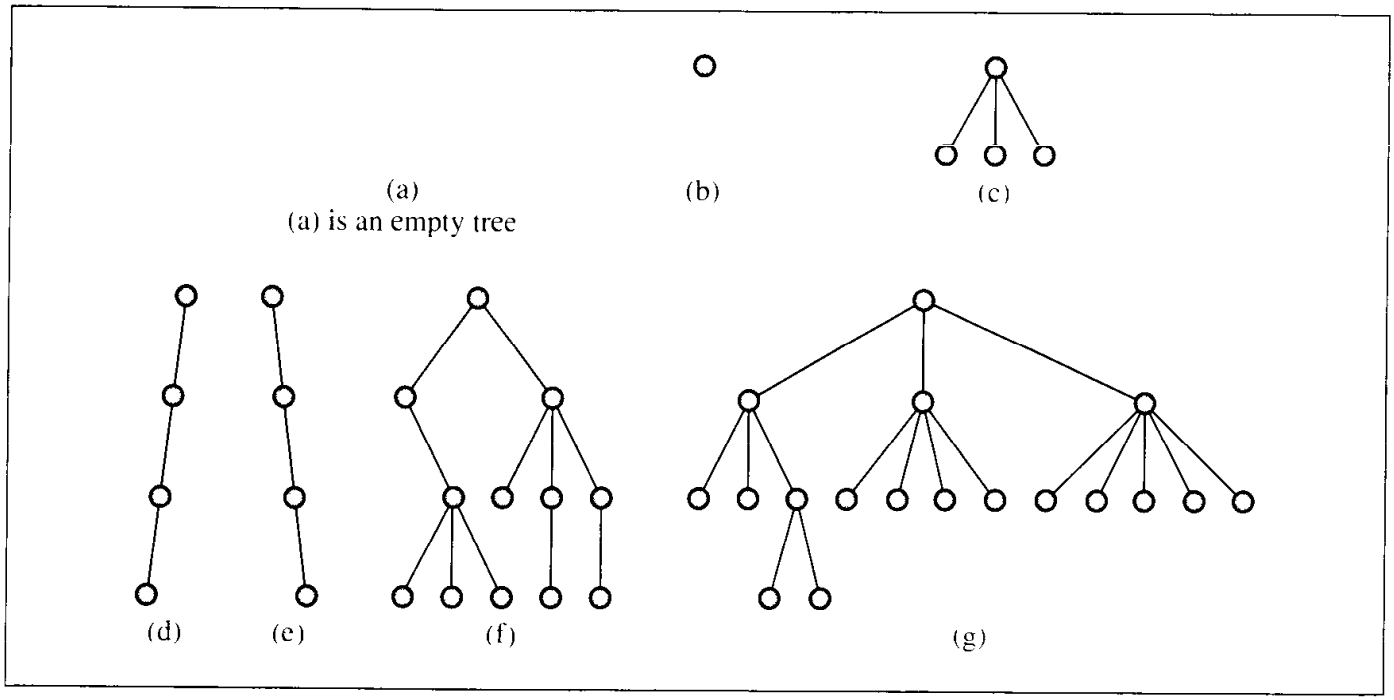


FIGURE 6.2 Hierarchical structure of a university shown as a tree.

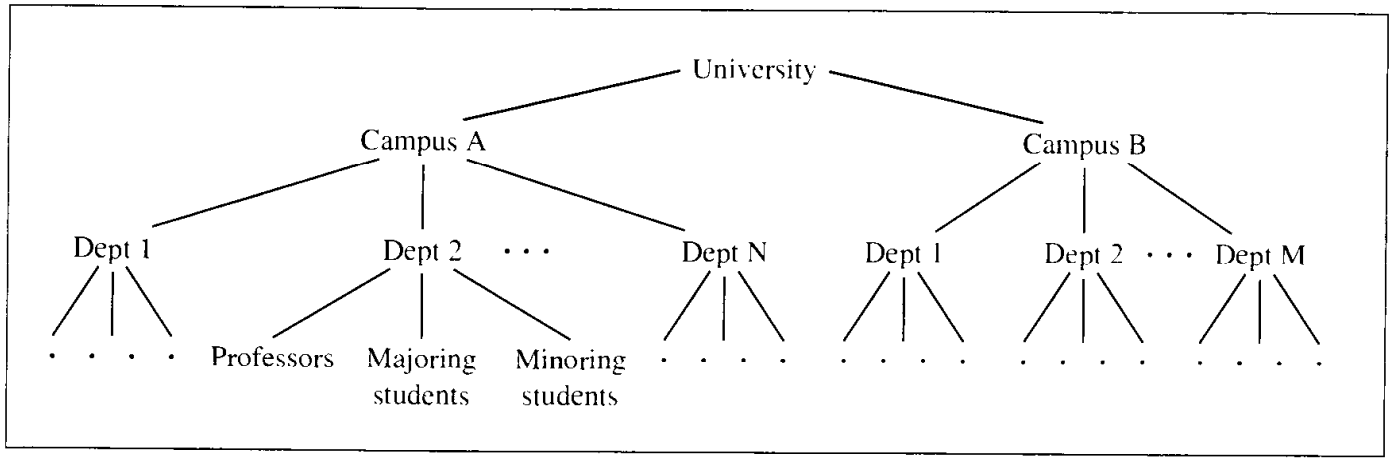
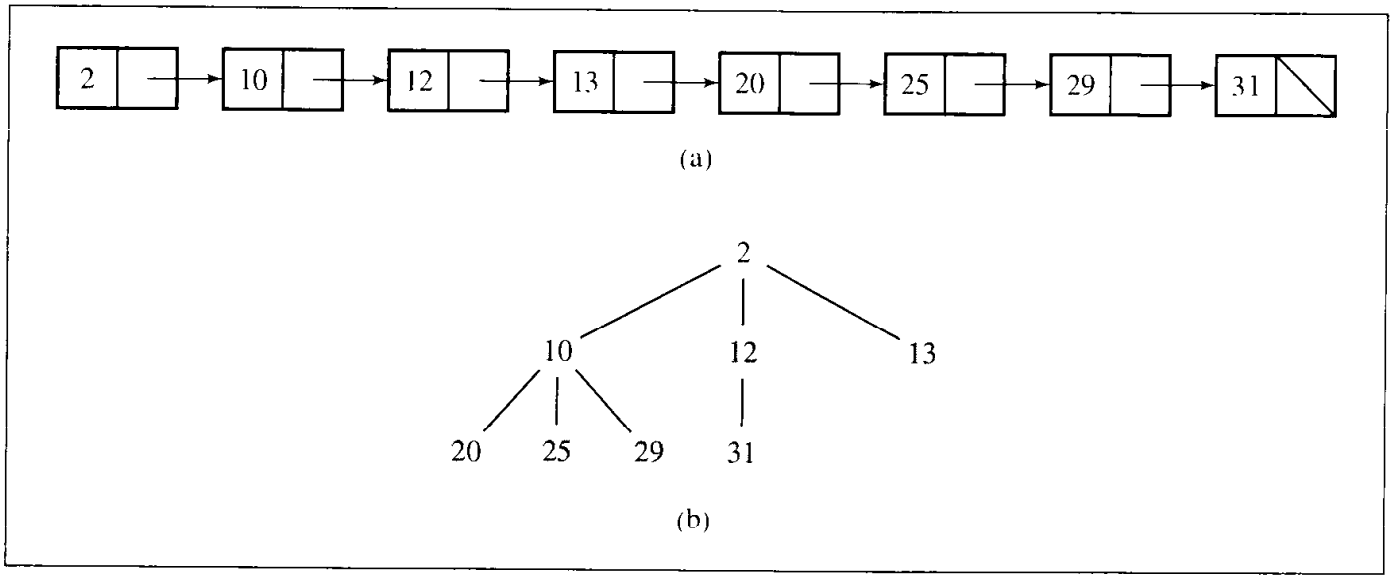


Figure 6.2 contains an example of a tree that reflects the hierarchy of a university. Other examples are genealogical trees, trees reflecting the grammatical structure of sentences, and trees showing the taxonomic structure of organisms, plants, or characters. Virtually all areas of science make use of trees to represent hierarchical structures.

The definition of a tree does not impose any condition on the number of children of a given node. This number can vary from 0 to any integer. In hierarchical trees, this is a welcome property. For example, the university has only two branches, but each campus can have a different number of departments. Such trees are used in database

FIGURE 6.3 Transforming (a) a linked list into (b) a tree.



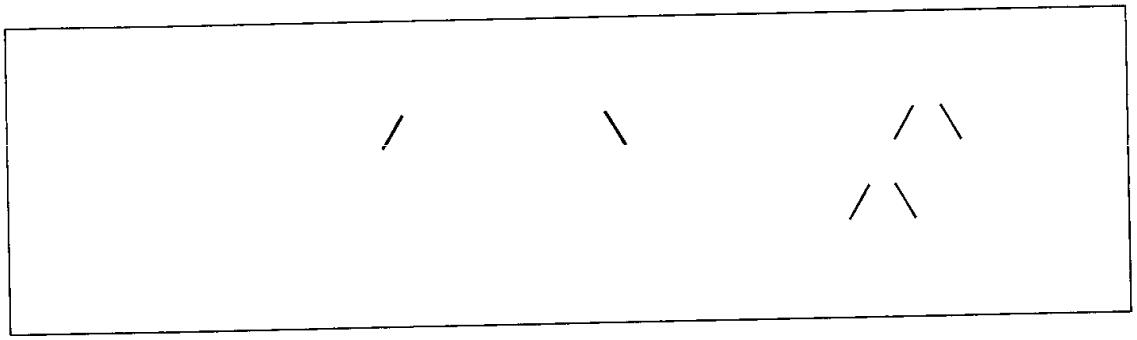
management systems, especially in the hierarchical model. But representing hierarchies is not the only reason for using trees. In fact, in the discussion to follow, that aspect of trees is treated rather lightly, mainly in the discussion of expression trees. This chapter focuses on tree operations that allow us to accelerate the search process.

Consider a linked list of n elements. To locate an element, the search has to start from the beginning of the list, and the list must be scanned until the element is found or the end of the list is reached. Even if the list is ordered, the search of the list always has to start from the first node. Thus, if the list has 10,000 nodes and the information in the last node is to be accessed, then all 9999 of its predecessors have to be traversed, an obvious inconvenience. If all the elements are stored in an *orderly tree*, a tree where all elements are stored according to some predetermined criterion of ordering, the number of tests can be reduced substantially even when the element to be located is the one furthest away. For example, the linked list in Figure 6.3a can be transformed into the tree in Figure 6.3b.

Was a reasonable criterion of ordering applied to construct this tree? To test whether 31 is in the linked list, eight tests have to be performed. Can this number be reduced further if the same elements are ordered from top to bottom and from left to right in the tree? What would an algorithm be like that forces us to make three tests only: one for the root, 2, one for its middle child, 12, and one for the only child of this child, 31? The number 31 could be located on the same level as 12, or it could be a child of 10. With this ordering of the tree, nothing really interesting is achieved in the context of searching. (The heap discussed later in this chapter uses this approach.) Consequently, a better criterion must be chosen.

Again, note that each node can have any number of children. In fact, there are algorithms developed for trees with a deliberate number of children (see the next chapter), but this chapter discusses only binary trees. A *binary tree* is a tree whose nodes have two children (possibly empty), and each child is designated as either a left child or

FIGURE 6.4 Examples of binary trees.



a right child. For example, the trees in Figure 6.4 are binary trees, whereas the university tree in Figure 6.2 is not. An important characteristic of binary trees, which is used later in assessing an expected efficiency of sorting algorithms, is the number of leaves.

As already defined, the level of a node is the number of arcs traversed from the root to the node plus one. According to this definition, the root is at level 1, its nonempty children are at level 2, and so on. If all the nodes at all levels except the last had two children, then there would be $1 = 2^0$ node at level 1, $2 = 2^1$ nodes at level 2, $4 = 2^2$ nodes at level 3, and generally, 2^i nodes at level $i + 1$. A tree satisfying this condition is referred to as a *complete binary tree*. In this tree, all nonterminal nodes have both their children, and all leaves are at the same level. Consequently, in all binary trees, there are at most 2^i nodes at level $i + 1$. In Chapter 9, we calculate the number of leaves in a *decision tree*, which is a binary tree in which all nodes have either zero or two nonempty children. Because leaves can be interspersed throughout a decision tree and appear at each level except level 1, no generally applicable formula can be given to calculate the number of nodes because it may vary from tree to tree. But the formula can be approximated by noting first that

> For all the nonempty binary trees whose nonterminal nodes have exactly two nonempty children, the number of leaves m is greater than the number of nonterminal nodes k and $m = k + 1$.

If a tree has only a root, this observation holds trivially. If it holds for a certain tree, then after attaching two leaves to one of the already existing leaves, this leaf turns into a nonterminal node, whereby m is decremented by 1 and k is incremented by 1. However, because two new leaves have been grafted onto the tree, m is incremented by 2. After these two increments and one decrement, the equation $(m - 1) + 2 = (k + 1) + 1$ is obtained and $m = k + 1$, which is exactly the result aimed at (see Figure 6.5). It implies that an $i + 1$-level complete decision tree has 2^i leaves, and on account of the preceding observation, it also has $2^i - 1$ nonterminal nodes, which makes $2^i + 2^i - 1 = 2^{i+1} - 1$ nodes in total (see also Figure 6.35).

In this chapter, the *binary search trees*, also called *ordered binary trees*, are of particular interest. A binary search tree has the following property: For each node n of the tree, all values stored in its left subtree (the tree whose root is the left child) are less than value v stored in n, and all values stored in the right subtree are greater than v.

FIGURE 6.5 Adding a leaf to tree (a), preserving the relation of the number of leaves to the number of nonterminal nodes (b).

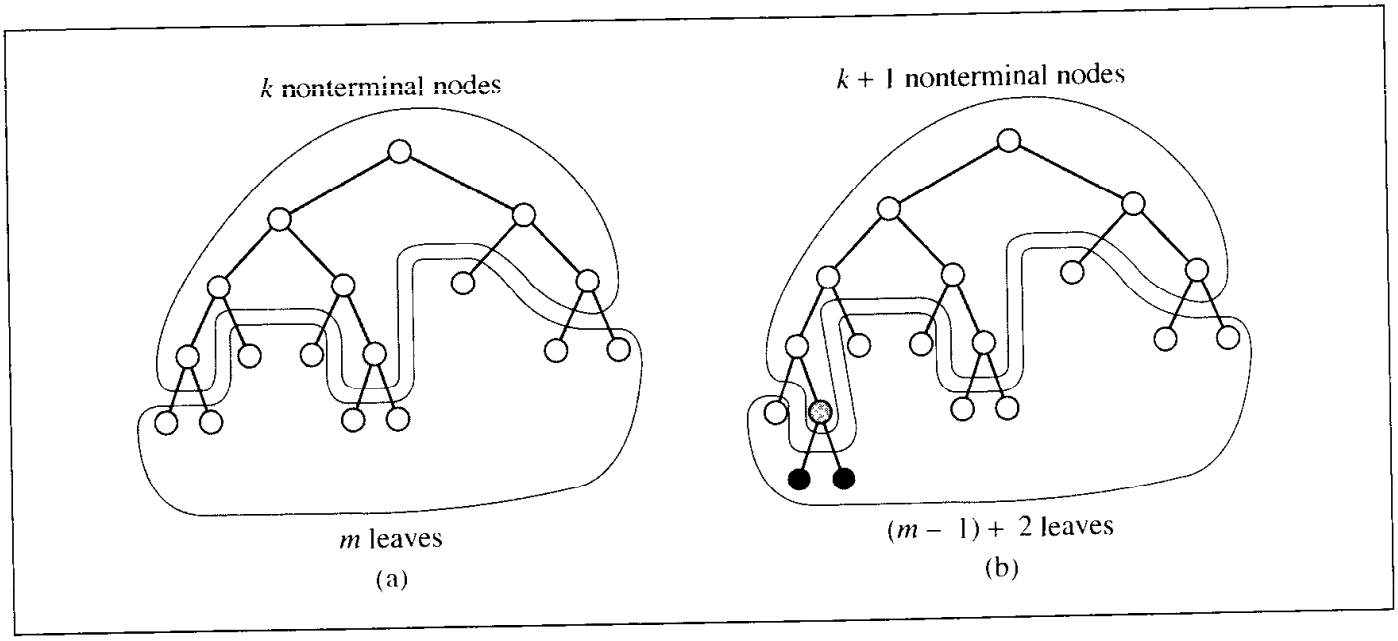
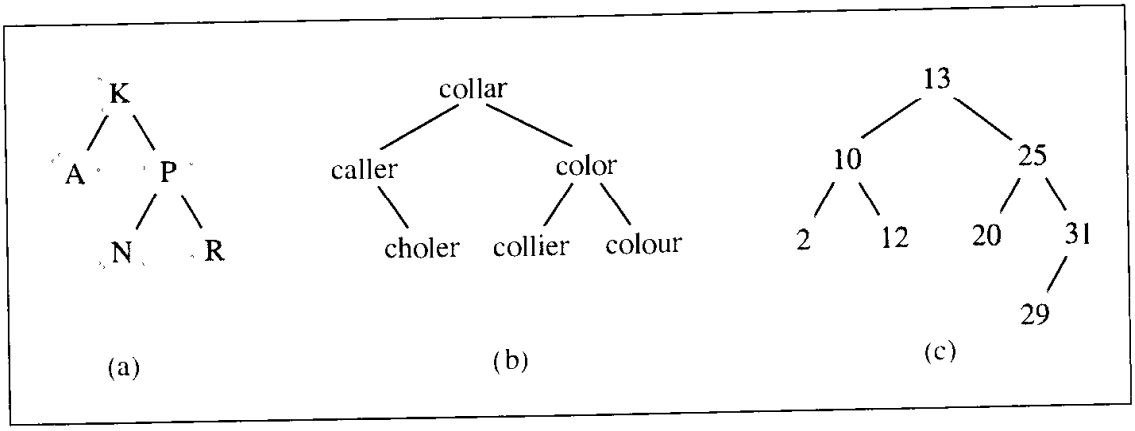


FIGURE 6.6 Examples of binary search trees.



For reasons to be discussed later, storing multiple copies of the same value in the same tree is avoided. An attempt to do so can be treated as an error. The meanings of "less than" or "greater than" depend on the type of values stored in the tree. We use operators "<" and ">" which can be overloaded depending on the content. Alphabetical order is also used in the case of strings. The trees in Figure 6.6 are binary search trees. Note that Figure 6.6c contains a tree with the same data as the linked list in Figure 6.3a whose searching was to be optimized.

6.2 Implementing Binary Trees

Binary trees can be implemented in at least two ways: as arrays and as linked structures. To implement a tree as an array, a node is declared as a structure with an information field and two "pointer" fields. These pointer fields contain the indexes of the array cells in which the left and right children are stored, if there are any. For example, the tree from Figure 6.6c can be represented as the array in Figure 6.7. The root is always located in the first cell, cell 0, and −1 indicates a null child. In this representation, the two children of node 13 are located in positions 4 and 2, and the right child of node 31 is null.

However, this implementation may be inconvenient, as is often the case with static allocation, since the size of the array has to become part of the program and must be known in advance. It is a problem because the data may overflow the array if too little space is allocated, or memory space may be wasted if too much space is allocated. This is important because trees often change and it may be hard to predict how many nodes will be created during program execution. However, sometimes an array implementation of a tree is convenient and desirable. It is used when discussing the heap sort, although an array of data rather than an array of structures is used there. But usually, a dynamic data structure is a more efficient way to represent a tree. This chapter's examples use dynamic data structures.

In the new implementation, a node is an instance of a class composed of an information member and two pointer members. This node is used and operated on by member functions in another class that pertains to the tree as a whole (see Figure 6.8). For this reason, members of `BSTNode` are declared public because they can be accessible only from nonpublic members of objects of type `BST` so that the information-hiding principle still stands. It is important to have members of `BSTNode` be public because otherwise they are not accessible to classes derived from `BST`.

FIGURE 6.7 Array representation of the tree in Figure 6.6c.

Index	Info	Left	Right
0	13	4	2
1	31	6	−1
2	25	7	1
3	12	−1	−1
4	10	5	3
5	2	−1	−1
6	29	−1	−1
7	20	−1	−1

FIGURE 6.8 Implementation of a generic binary search tree.

```
//************************  genBST.h  *************************
//                  generic binary search tree

#include <queue>
#include <stack>

using namespace std;

#ifndef BINARY_SEARCH_TREE
#define BINARY_SEARCH_TREE

template<class T>
class Stack : public stack<T> { ... } // as in Figure 4.21

template<class T>
class Queue : public queue<T> {
public:
    T dequeue() {
        T tmp = front();
        queue<T>::pop();
        return tmp;
    }
    void enqueue(const T& el) {
        push(el);
    }
};

template<class T>
class BSTNode {
public:
    BSTNode() {
        left = right = 0;
    }
    BSTNode(const T& el, BSTNode *l = 0, BSTNode *r = 0) {
        key = el; left = l; right = r;
    }
    T key;
    BSTNode *left, *right;
};
```

FIGURE 6.8 *(continued)*

```
template<class T>
class BST {
public:
    BST() {
        root = 0;
    }
    ~BST() {
        clear();
    }
    void clear() {
        clear(root); root = 0;
    }
    bool isEmpty() const {
        return root == 0;
    }
    void preorder() {
        preorder(root);                          // Figure 6.11
    }
    void inorder() {
        inorder(root);                           // Figure 6.11
    }
    void postorder() {
        postorder(root);                         // Figure 6.11
    }
    T* search(const T& el) const {
        return search(root,el);                  // Figure 6.9
    }
    void breadthFirst();                         // Figure 6.10
    void iterativePreorder();                    // Figure 6.15
    void iterativeInorder();                     // Figure 6.17
    void iterativePostorder();                   // Figure 6.16
    void MorrisInorder();                        // Figure 6.20
    void insert(const T&);                       // Figure 6.23
    void deleteByMerging(BSTNode<T>*&);          // Figure 6.29
    void findAndDeleteByMerging(const T&);       // Figure 6.29
    void deleteByCopying(BSTNode<T>*&);          // Figure 6.32
    void balance(T*,int,int);                    // Section 6.7
    . . . . . .
protected:
    BSTNode<T>* root;
```

Continues

FIGURE 6.8 *(continued)*

```
    void clear(BSTNode<T>*);
    T* search(BSTNode<T>*, const T&) const;  // Figure 6.9
    void preorder(BSTNode<T>*);              // Figure 6.11
    void inorder(BSTNode<T>*);               // Figure 6.11
    void postorder(BSTNode<T>*);             // Figure 6.11
    virtual void visit(BSTNode<T>* p) {
        cout << p->key << ' ';
    }
    . . . . . .
};

#endif
```

6.3 SEARCHING A BINARY SEARCH TREE

An algorithm for locating an element in this tree is quite straightforward as indicated by its implementation in Figure 6.9. For every node, compare the key to be located with the value stored in the node currently pointed at. If the key is less than the value, go to the left subtree and try again. If it is greater than that value, try the right subtree. If it is the same, obviously the search can be discontinued. The search is also aborted if there is no way to go, indicating that the key is not in the tree. For example, to locate the number 31 in the tree in Figure 6.6c, only three tests are performed. First, the tree is checked to see if the number is in the root node. Next, because 31 is greater than 13, the root's right child containing the value 25 is tried. Finally, since 31 is again greater than the value of the currently tested node, the right child is tried again, and the value 31 is found.

The worst case for this binary tree is when it is searched for the numbers 26, 27, 28, 29, or 30 because those searches each require four tests (why?). In the case of all other integers, the number of tests is fewer than four. It can now be seen why an element should only occur in a tree once. If it occurs more than once, then two approaches are possible. One approach locates the first occurrence of an element and disregards the others. In this case, the tree contains redundant nodes that are never used for their own sake; they are accessed only for testing. In the second approach, all occurrences of an element may have to be located. Such a search always has to finish with a leaf. For example, to locate all instances of 13 in the tree, the root node 13 has to be tested, then its right child 25, and finally the node 20. The search proceeds along

FIGURE 6.9 A function for searching binary search tree.

```
template<class T>
T* BST<T>::search(BSTNode<T>* p, const T& el) const {
    while (p != 0)
        if (el == p->key)
            return &p->key;
        else if (el < p->key)
            p = p->left;
        else p = p->right;
    return 0;
}
```

the worst-case scenario: when the leaf level has to be reached in expectation that some more occurrences of the desired element can be encountered.

The complexity of searching is measured by the number of comparisons performed during the searching process. This number depends on the number of nodes encountered on the unique path leading from the root to the node being searched for. Therefore, the complexity is the length of the path leading to this node plus 1. Complexity depends on the shape of the tree and the position of the node in the tree.

The *internal path length* (IPL) is the sum of all path lengths of all nodes, which is calculated by summing $\sum(i-1)l_i$ over all levels i, where l_i is the number of nodes on level i. A position of a node in the tree is determine by the path length. An average position, called an *average path length*, is given by the formula IPL/n, which depends on the shape of the tree. In the worst case, when the tree turns into a linked list, $path_{worst} = \frac{1}{n}\sum_{i=1}^{n}(i-1) = \frac{n-1}{2} = O(n)$, and a search can take n time units.

The best case occurs when all leaves in the tree of height h are in at most two levels, and only nodes in the next to last level can have one child. To simplify the computation, we approximate the average path length for such a tree, $path_{best}$, by the average path of a complete binary tree of the same height.

By looking at simple examples, we can determine that for the complete binary tree of height h, IPL = $\sum_{i=1}^{h-1} i2^i$. From this and from the fact that $\sum_{i=1}^{h-1} 2^i = 2^h - 2$, we have

$$\text{IPL} = 2\text{IPL} - \text{IPL} = (h-1)2^h - \sum_{i=1}^{h-1} 2^i = (h-2)2^h + 2$$

As has already been established, the number of nodes in the complete binary tree $n = 2^h - 1$, so

$$path_{best} = \text{IPL}/n = \big((h-2)2^h + 2\big)/(2^h - 1) \approx h - 2$$

which is in accordance with the fact that, in this tree, one-half of the nodes are in the leaf level with path length $h-1$. Also, in this tree, the height $h = \lg(n+1)$, so $path_{best} = \lg(n+1) - 2$; the average path length in a perfectly balanced tree is $\lceil \lg(n+1) \rceil - 2 = O(\lg n)$ where $\lceil x \rceil$ is the closest integer greater than x.

The average case in an average tree is somewhere between $\frac{n-1}{2}$ and $\lg(n+1)-2$. Is a search for a node in an average position in a tree of average shape closer to $O(n)$ or $O(\lg n)$? First, the average shape of the tree has to be represented computationally.

The root of a binary tree can have an empty left subtree and a right subtree with all $n-1$ nodes. It also can have one node in the left subtree and $n-2$ nodes in the right and so on. Finally, it can have an empty right subtree with all remaining nodes in the left. The same reasoning can be applied to both subtrees of the root, to the subtrees of these subtrees, down to the leaves. The average internal path length is the average over all these differently shaped trees.

Assume that the tree contains nodes 1 through n. If i is the root, then its left subtree has $i-1$ nodes, and its right subtree has $n-i$ nodes. If $path_{i-1}$ and $path_{n-i}$ are average paths in these subtrees, then the average path of this tree is

$$path_n(i) = ((i-1)(path_{i-1}+1)+(n-i)(path_{n-i}+1))/n$$

Assuming that elements are coming randomly to the tree, the root of the tree can be any number i, $1 \le i \le n$. Therefore, the average path of an average tree is obtained by averaging all values of $path_n(i)$ over all values of i. This gives the formula

$$path_n = \frac{1}{n}\sum_{i=1}^{n} path_n(i) = \frac{1}{n^2}\sum_{i=1}^{n}((i-1)(path_{i-1}+1)+(n-i)(path_{n-i}+1))$$

$$= \frac{2}{n^2}\sum_{i=1}^{n-1} i(path_i+1)$$

from which, and from $path_1 = 0$, we obtain $2 \ln n = 2 \ln 2 \lg n = 1.386 \lg n$ as an approximation for $path_n$ (see section A.4 in Appendix A). This is an approximation for the average number of comparisons in an average tree. This number is $O(\lg n)$, which is closer to the best case than to the worst case. This number also indicates that there is little room for improvement, since $path_{best}/path_n \approx .7215$, and the average path length in the best case is different by only 27.85% from the expected path length in the average case. Searching in a binary tree is, therefore, very efficient in most cases, even without balancing the tree. However, this is true only for randomly created trees because, in highly unbalanced and elongated trees whose shapes resemble linked lists, search time is $O(n)$, which is unacceptable considering that $O(\lg n)$ efficiency can be achieved.

6.4 Tree Traversal

Tree traversal is the process of visiting each node in the tree exactly one time. Traversal may be interpreted as putting all nodes on one line or linearizing a tree.

The definition of traversal specifies only one condition—visiting each node only one time—but it does not specify the order in which the nodes are visited. Hence, there are as many tree traversals as there are permutations of nodes; for a tree with n nodes, there are $n!$ different traversals. Most of them, however, are rather chaotic and do not indicate much regularity so that implementing such traversals lacks generality: For each n, a separate set of traversal procedures must be implemented, and only a few of them can be used for a different number of data. For example, two possible traver-

sals of the tree in Figure 6.6c that may be of some use are the sequence 2, 10, 12, 20, 13, 25, 29, 31 and the sequence 29, 31, 20, 12, 2, 25, 10, 13. The first sequence lists even numbers and then odd numbers in ascending order. The second sequence lists all nodes from level to level right to left, starting from the lowest level up to the root. The sequence 13, 31, 12, 2, 10, 29, 20, 25 does not indicate any regularity in the order of numbers or in the order of the traversed nodes. It is just a random jumping from node to node that in all likelihood is of no use. Nevertheless, all these sequences are the results of three legitimate traversals out of 8! = 40,320. In the face of such an abundance of traversals and the apparent uselessness of most of them, we would like to restrict our attention to two classes only, namely, breadth-first and depth-first traversals.

6.4.1 Breadth-First Traversal

Breadth-first traversal is visiting each node starting from the lowest (or highest) level and moving down (or up) level by level, visiting nodes on each level from left to right (or from right to left). There are thus four possibilities, and one such possibility—a top-down, left-to-right breadth-first traversal of the tree in Figure 6.6c—results in the sequence 13, 10, 25, 2, 12, 20, 31, 29.

Implementation of this kind of traversal is straightforward when a queue is used. Consider a top-down left-to-right, breadth-first traversal. After a node is visited, its children, if any, are placed at the end of the queue, and the node at the beginning of the queue is visited. Considering that for a node on level n, its children are on level $n + 1$, by placing these children at the end of the queue, they are visited after all nodes from level n are visited. Thus, the restriction that all nodes on level n must be visited before visiting any nodes on level $n + 1$ is accomplished.

An implementation of the corresponding member function is shown in Figure 6.10.

6.4.2 Depth-First Traversal

Depth-first traversal proceeds as far as possible to the left (or right), then backs up until the first crossroad, goes one step to the right (or left), and again as far as possible to the left (or right). We repeat this process until all nodes are visited. This definition, however, does not clearly specify exactly when nodes are visited: before proceeding down the tree or after backing up? There are some variations of the depth-first traversal.

There are three tasks of interest in this type of traversal:

V—visiting a node

L—traversing the left subtree

R—traversing the right subtree

An orderly traversal takes place if these tasks are performed in the same order for each node. The three tasks can themselves be ordered in 3! = 6 ways, so there are six possible ordered depth-first traversals:

VLR VRL

LVR RVL

LRV RLV

FIGURE 6.10 Top-down, left-to-right, breadth-first traversal implementation.

```
template<class T>
void BST<T>::breadthFirst() {
    Queue<BSTNode<T>*> queue;
    BSTNode<T> *p = root;
    if (p != 0) {
        queue.enqueue(p);
        while (!queue.empty()) {
            p = queue.dequeue();
            visit(p);
            if (p->left != 0)
                queue.enqueue(p->left);
            if (p->right != 0)
                queue.enqueue(p->right);
        }
    }
}
```

If the number of different orders still seems like a lot, it can be reduced to three traversals where the move is always from left to right and attention is focused on the first column. The three traversals are given these standard names:

VLR—preorder tree traversal

LVR—inorder tree traversal

LRV—postorder tree traversal

Short and elegant functions can be implemented directly from the symbolic descriptions of these three traversals, as shown in Figure 6.11.

These functions may seem too simplistic, but their real power lies in recursion, in fact, double recursion. The real job is done by the system on the run-time stack. This simplifies coding but lays a heavy burden upon the system. To better understand this process, inorder tree traversal is discussed in some detail.

In inorder traversal, the left subtree of the current node is visited first, then the node itself, and finally, the right subtree. All of this, obviously, holds if the tree is not empty. Before drawing aside the run-time curtain by analyzing the run-time stack, the output given by the inorder traversal is determined by referring to Figure 6.12. The following steps correspond to the letters in that figure:

(a) Node 15 is the root on which `inorder()` is called for the first time. The function calls itself for node 15's left child, node 4.

(b) Node 4 is not null, so `inorder()` is called on node 1. Because node 1 is a leaf (that is, both its subtrees are empty), invocations of `inorder()` on the subtrees do not result

FIGURE 6.11 Depth-first traversal implementation.

```
template<class T>
void BST<T>::inorder(BSTNode<T> *p) {
     if (p != 0) {
         inorder(p->left);
         visit(p);
         inorder(p->right);
     }
}

template<class T>
void BST<T>::preorder(BSTNode<T> *p) {
    if (p != 0) {
        visit(p);
        preorder(p->left);
        preorder(p->right);
    }
}

template<class T>
void BST<T>::postorder(BSTNode<T>* p) {
    if (p != 0) {
        postorder(p->left);
        postorder(p->right);
        visit(p);
    }
}
```

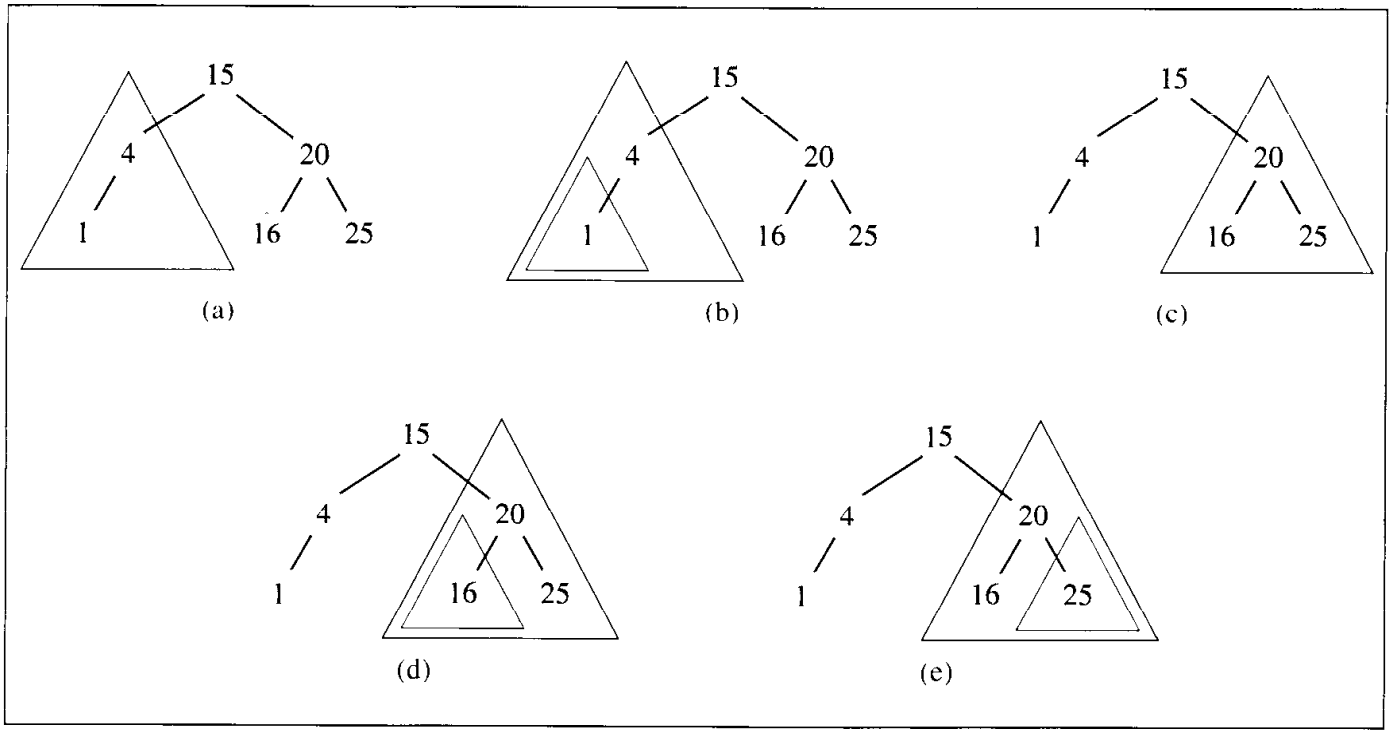
in other recursive calls of `inorder()`, as the condition in the `if` statement is not met. Thus, after `inorder()` called for the null left subtree is finished, node 1 is visited and then a quick call to `inorder()` is executed for the null right subtree of node 1. After resuming the call for node 4, node 4 is visited. Node 4 has a null right subtree; hence, `inorder()` is called only to check that, and right after resuming the call for node 15, node 15 is visited.

(c) Node 15 has a right subtree so, `inorder()` is called for node 20.

(d) `inorder()` is called for node 16, the node is visited, and then on its null left subtree, which is followed by visiting node 16. After a quick call to `inorder()` on the null right subtree of node 16 and return to the call on node 20, node 20 is also visited.

(e) `inorder()` is called on node 25, then on its empty left subtree, then node 25 is visited, and finally `inorder()` is called on node 25's empty right subtree.

FIGURE **6.12** Inorder tree traversal.



If the visit includes printing the value stored in a node, then the output is:

```
1 4 15 16 20 25
```

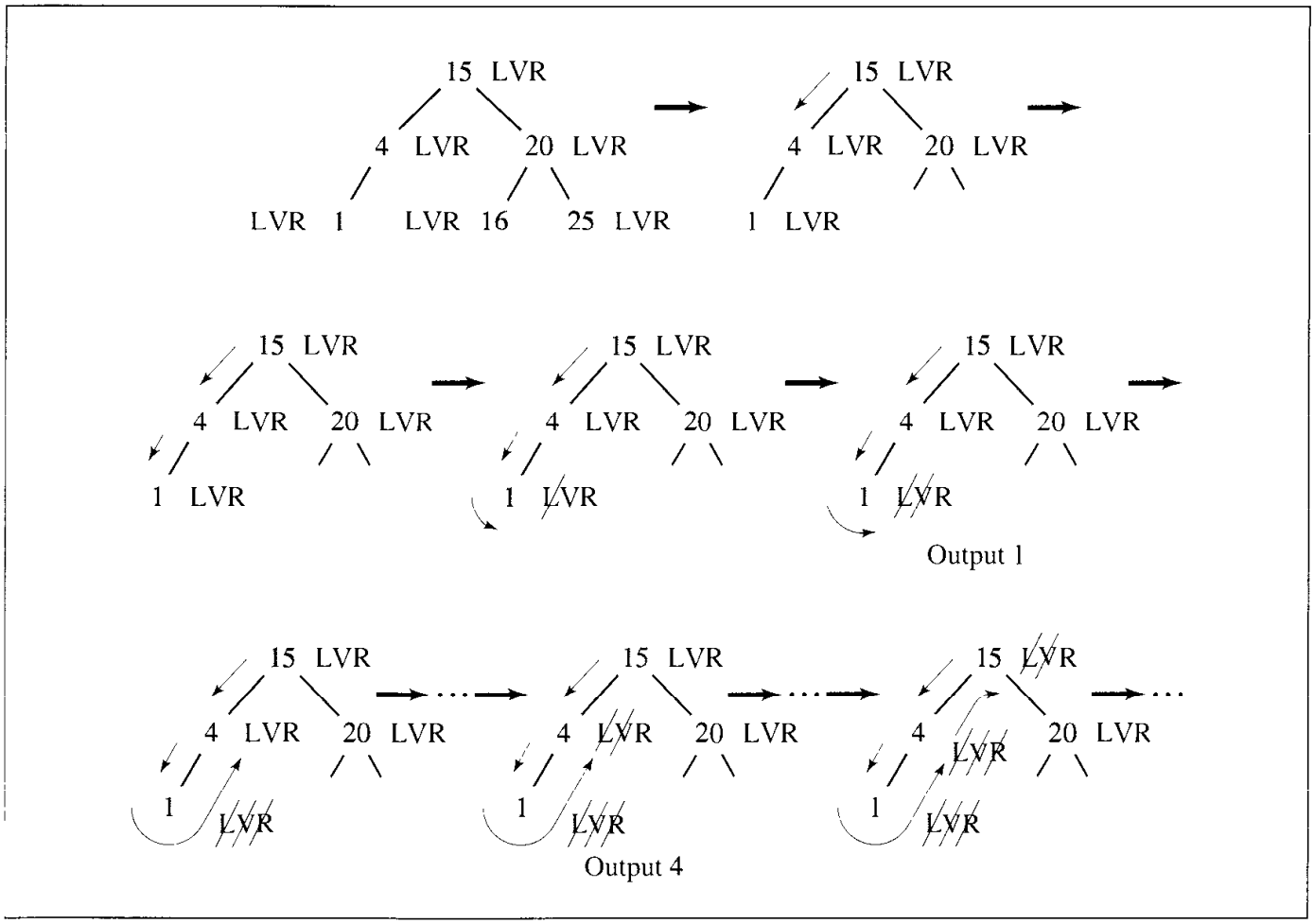
The key to the traversal is that the three tasks, L, V, and R, are performed for each node separately. This means that the traversal of the right subtree of a node is held pending until the first two tasks, L and V, are accomplished. If the latter two are finished, they can be crossed out as in Figure 6.13.

To present the way `inorder()` works, the behavior of the run-time stack is observed. The numbers in parentheses in Figure 6.14 indicate return addresses shown on the left-hand side of the code for `inorder()`.

```
        template<class T>
        void BST<T>::inorder(BSTnode<T> *node) {
            if (node ! = 0) {
/* 1 */             inorder(node->left);
/* 2 */             visit(node);
/* 3 */             inorder(node->right);
/* 4 */         }
        }
```

A rectangle with an up arrow and a number indicates the current value of `node` pushed onto the stack. For example, ↑4 means that `node` points to the node of the tree whose value is the number 4. Figure 6.14 shows the changes of the run-time stack when `inorder()` is executed for the tree in Figure 6.12.

FIGURE 6.13 Details of several of the first steps of inorder traversal.



(a) Initially, the run-time stack is empty (or rather it is assumed that the stack is empty by disregarding what has been stored on it before the first call to `inorder()`).

(b) Upon the first call, the return address of `inorder()` and the value of `node`, ↑15, are pushed onto the run-time stack. The tree, pointed to by `node`, is not empty, the condition in the `if` statement is satisfied, and `inorder()` is called again with node 4.

(c) Before it is executed, the return address, (2), and current value of `node`, ↑4, are pushed onto the stack. Since `node` is not null, `inorder()` is about to be invoked for `node`'s left child, ↑1.

(d) First, the return address, (2), and the `node`'s value are stored on the stack.

(e) `inorder()` is called with node 1's left child. The address (2) and the current value of parameter `node`, null, are stored on the stack. Since `node` is null, `inorder()` is exited immediately; upon exit, the activation record is removed from the stack.

(f) The system goes now to its run-time stack, restores the value of the `node`, ↑1, executes the statement under (2) and prints the number 1. Since `node` is not completely processed, the value of `node` and address (2) are still on the stack.

FIGURE 6.14 Changes in the run-time stack during inorder traversal.

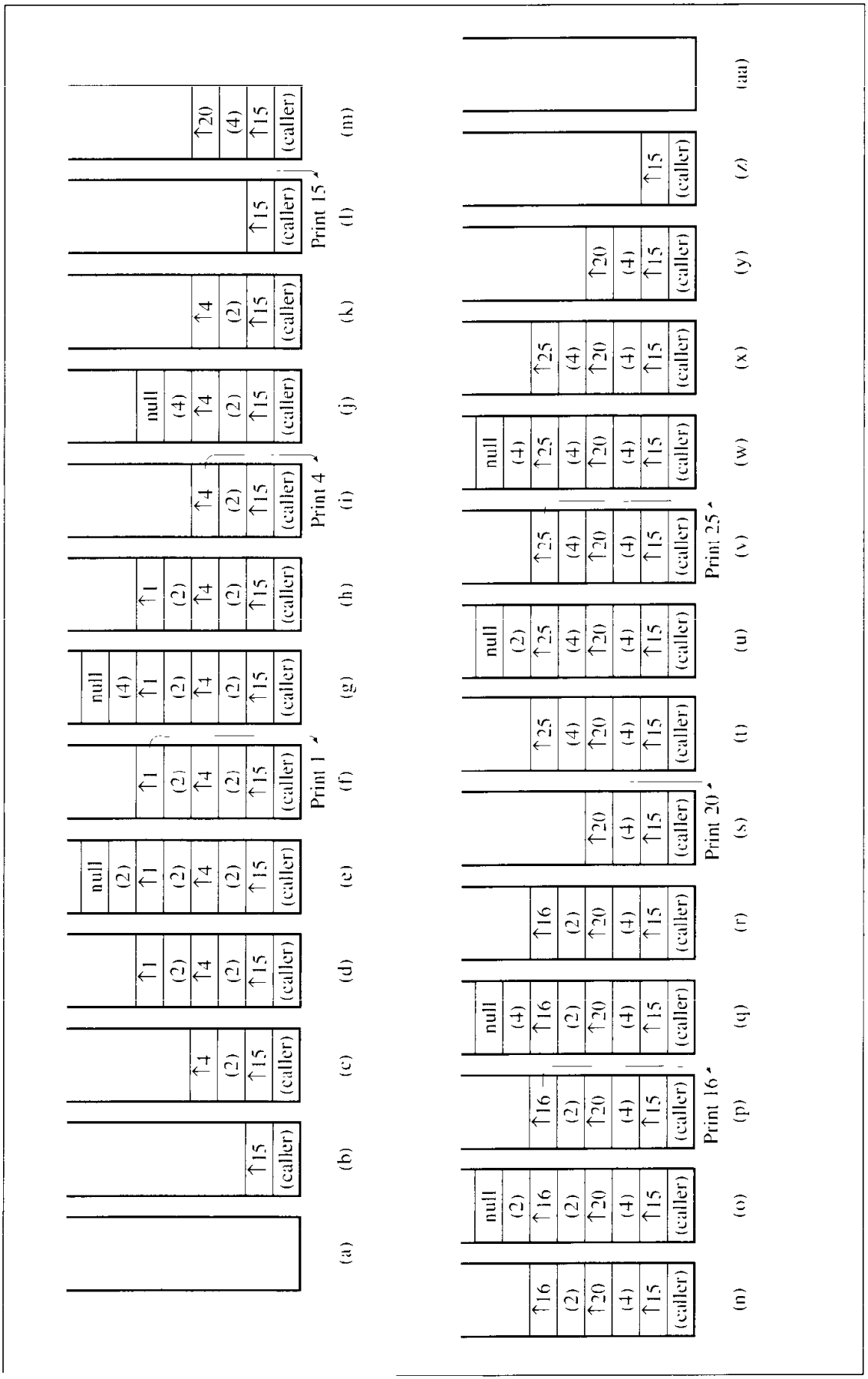


FIGURE **6.15** A non-recursive implementation of preorder tree traversal.

```
template<class T>
void BST<T>::iterativePreorder() {
    Stack<BSTNode<T>*> travStack;
    BSTNode<T> *p = root;
    if (p != 0) {
        travStack.push(p);
        while (!travStack.empty()) {
            p = travStack.pop();
            visit(p);
            if (p->right != 0)
                travStack.push(p->right);
            if (p->left != 0)   // left child pushed after right
                travStack.push(p->left); // to be on the top of
        }                        // the stack;
    }
}
```

(g) With the right child of `node` ↑1, the statement under (3) is executed, which is the next call to `inorder()`. First, however, the address (4) and `node`'s current value, null, are pushed onto the stack. Because `node` is null, `inorder()` is exited; upon exit, the stack is cleaned up.

(h) The system now restores the old value of the `node`, ↑1, and executes statement (4).

(i) Since this is `inorder()`'s exit, the system removes the current activation record and refers again to the stack, restores the `node`'s value, ↑4, and resumes execution from statement (2). This prints the number 4 and then calls `inorder()` for the right child of `node`, which is null.

These steps are just the beginning. All of the steps are shown in Figure 6.14.

At this point, consider the problem of a nonrecursive implementation of the three traversal algorithms. As indicated in Chapter 5, a recursive implementation has a tendency to be less efficient than a nonrecursive counterpart. If two recursive calls are used in a function, then the problem of possible inefficiency doubles. Can recursion be eliminated from the implementation? The answer has to be positive because if it is not eliminated in the source code, the system does it for us anyway. So the question should be rephrased: Is it expedient to do so?

Look first at a nonrecursive version of the preorder tree traversal shown in Figure 6.15. The function `iterativePreorder()` is twice as large as `preorder()`, but it is still short and legible. However, it uses a stack heavily. Therefore, supporting functions are necessary to process the stack, and the overall implementation is not so short. Although two recursive calls are omitted, there are now up to four calls per iteration of

`while` loop: up to two calls of `push()`, one call of `pop()`, and one call of `visit()`. This can hardly be considered an improvement in efficiency.

In the recursive implementations of the three traversals, note that the only difference is in the order of the lines of code. For example, in `preorder()`, first a node is visited, and then there are calls for the left and right subtrees. On the other hand, in `postorder()`, visiting a node succeeds both calls. Can we so easily transform the nonrecursive version of a left-to-right preorder traversal into a nonrecursive left-to-right postorder traversal? Unfortunately, no. In `iterativePreorder()`, visiting occurs before both children are pushed onto the stack. But this order does not really matter. If the children are pushed first and then the node is visited, that is, if `visit(p)` is placed after both calls to `push()`, the resulting implementation is still a preorder traversal. What matters here is that `visit()` has to follow `pop()` and the latter has to precede both calls of `push()`. Therefore, nonrecursive implementations of inorder and postorder traversals have to be developed independently.

A nonrecursive version of postorder traversal can be obtained rather easily if we observe that the sequence generated by a left-to-right postorder traversal (a LRV order) is the same as the reversed sequence generated by a right-to-left preorder traversal (a VRL order). In this case, the implementation of `iterativePreorder()` can be adopted to create `iterativePostorder()`. This means that two stacks have to be used, one to visit each node in the reverse order after a right-to-left preorder traversal is finished. It is, however, possible to develop a function for postorder traversal that pushes onto the stack a node that has two descendants, once before traversing its left subtree and once before traversing its right subtree. An auxiliary pointer is used to distinguish between these two cases. Nodes with one descendant are pushed only once, and leaves do not need to be pushed at all (Figure 6.16).

A nonrecursive inorder tree traversal is also a complicated matter. One possible implementation is given in Figure 6.17. In cases like this, we can clearly see the power of recursion: `iterativeInorder()` is almost unreadable, and without thorough explanation, it is not easy to determine the purpose of this function. On the other hand, recursive `inorder()` immediately demonstrates a purpose and logic. Therefore, `iterativeInorder()` can be defended in one case only: if it is shown that there is a substantial gain in execution time and that the function is called often in a program. Otherwise, `inorder()` is preferable to its iterative counterpart.

6.4.3 Stackless Depth-First Traversal

Threaded Trees

The traversal functions analyzed in the preceding section were either recursive or nonrecursive, but both kinds used a stack either implicitly or explicitly to store information about nodes whose processing has not been finished. In the case of recursive functions, the run-time stack was utilized. In the case of nonrecursive variants, an explicitly defined and user-maintained stack was used. The concern is that some additional time has to be spent to maintain the stack, and some more space has to be set aside for the stack itself. In the worst case, when the tree is unfavorably skewed, the stack may hold information about almost every node of the tree, a serious concern for very large trees.

FIGURE 6.16 A non-recursive implementation of postorder tree traversal.

```
template<class T>
void BST<T>::iterativePostorder() {
    Stack<BSTNode<T>*> travStack;
    BSTNode<T>* p = root, *q = root;
    while (p != 0) {
        for ( ; p->left != 0; p = p->left)
            travStack.push(p);
        while (p != 0 && (p->right == 0 || p->right == q)) {
            visit(p);
            q = p;
            if (travStack.empty())
                return;
            p = travStack.pop();
        }
        travStack.push(p);
        p = p->right;
    }
}
```

It is more efficient to incorporate the stack as part of the tree. This is done by incorporating *threads* in a given node. Threads are pointers to the predecessor and successor of the node according to an inorder traversal, and trees whose nodes use threads are called *threaded trees*. Four pointers are needed for each node in the tree, which again takes up valuable space.

The problem can be solved by overloading existing pointers. In trees, left or right pointers are pointers to children, but they can also be used as pointers to predecessors and successors, thereby being overloaded with meaning. How do we distinguish these meanings? For an overloaded operator, context is always a disambiguating factor. In trees, however, a new data member has to be used to indicate the current meaning of the pointers.

Because a pointer can point to one node at a time, the left pointer is either a pointer to the left child or to the predecessor. Analogously, the right pointer points either to the right subtree or to the successor (Figure 6.18a).

Figure 6.18a suggests that both pointers to predecessors and to successors have to be maintained, which is not always the case. It may be sufficient to use only one thread as shown in the implementation of the inorder traversal of a threaded tree, which requires only pointers to successors (Figure 6.18b).

The function is relatively simple. The dashed line in Figure 6.18b indicates the order in which `p` accesses nodes in the tree. Note that only one variable, `p`, is needed to traverse the tree. No stack is needed; therefore, space is saved. But is it really? As indicated, nodes require a data member indicating how the right pointer is being used. In the implementation of `threadedInorder()`, the Boolean data member `successor` plays this role as shown in Figure 6.19. Hence, `successor` requires only one bit of

FIGURE **6.17** A non-recursive implementation of inorder tree traversal.

```
template<class T>
void BST<T>::iterativeInorder() {
    Stack<BSTNode<T>*> travStack;
    BSTNode<T> *p = root;
    while (p != 0) {
        while (p != 0) {          // stack the right child (if any)
            if (p->right)         // and the node itself when going
               travStack.push(p->right); // to the left;
            travStack.push(p);
            p = p->left;
        }
        p = travStack.pop();      // pop a node with no left child
        while (!travStack.empty() && p->right == 0) { // visit it
            visit(p);             // and all nodes with no right
            p = travStack.pop(); // child;
        }
        visit(p);                 // visit also the first node with
        if (!travStack.empty())  // a right child (if any);
             p = travStack.pop();
        else p = 0;
    }
}
```

FIGURE **6.18** (a) A threaded tree and (b) an inorder traversal's path in a threaded tree with right successors only.

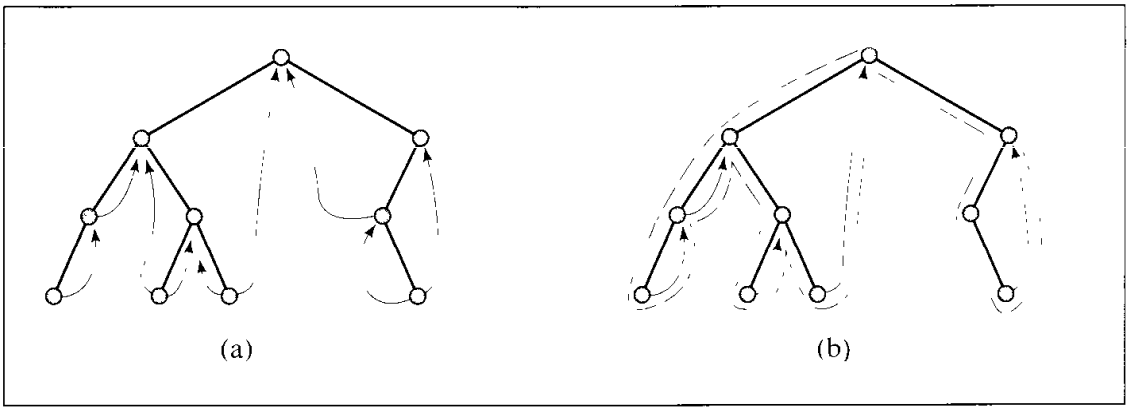


FIGURE **6.19** Implementation of the generic threaded tree and the inorder traversal of a threaded tree.

```
//************************  genThreaded.h  *********************
//              Generic binary search threaded tree

#ifndef THREADED_TREE
#define THREADED_TREE

template<class T>
class ThreadedNode {
public:
    ThreadedNode() {
        left = right = 0;
    }
    ThreadedNode(const T& el, ThreadedNode *l = 0, ThreadedNode *r = 0) {
        key = el; left = l; right = r; successor = 0;
    }
    T key;
    ThreadedNode *left, *right;
    unsigned int successor : 1;
};

template<class T>
class ThreadedTree {
public:
    ThreadedTree()  {
        root = 0;
    }
    void insert(const T&);                  // Figure 6.24
    void inorder();
   . . . . . .
protected:
    ThreadedNode<T>* root;
    . . . . . .
};

#endif

template<class T>
void ThreadedTree<T>::inorder() {
```

Continues

FIGURE 6.19 *(continued)*

```
    ThreadedNode<T> *prev, *p = root;
    if (p != 0) {                  // process only nonempty trees;
        while (p->left != 0)       // go to the leftmost node;
            p = p->left;
        while (p != 0) {
            visit(p);
            prev = p;
            p = p->right;          // go to the right node and only
            if (p != 0 && prev->successor == 0) // if it is a
                while (p->left != 0)// descendant go to the
                    p = p->left; // leftmost node, otherwise
        }                          // visit the successor;
    }
}
```

computer memory, insignificant in comparison to other fields. However, the exact details are highly dependent on the implementation. The operating system almost certainly pads a bit structure with additional bits for proper alignment of machine words. If so, `successor` needs at least one byte, if not an entire word, defeating the argument about saving space by using threaded trees.

Threaded trees can also be used for preorder and postorder traversals. In preorder traversal, the current node is visited first and then traversal continues with its left descendant, if any, or right descendant, if any. If the current node is a leaf, threads are used to go through the chain of its already visited inorder successors to restart traversal with the right descendant of the last successor.

Postorder traversal is only slightly more complicated. First, a dummy node is created that has the root as its left descendant. In the traversal process, a variable can be used to check the type of the current action. If the action is left traversal and the current node has a left descendant, then the descendant is traversed; otherwise, the action is changed to right traversal. If the action is right traversal and the current node has a right nonthread descendant, then the descendant is traversed and the action is changed to left traversal; otherwise, the action changes to visiting a node. If the action is visiting a node, then the current node is visited, and afterward, its postorder successor has to be found. If the current node's parent is accessible through a thread (that is, current node is parent's left child), then traversal is set to continue with the right descendant of the parent. If the current node has no right descendant, then it is the end of the right-extended chain of nodes. First, the beginning of the chain is reached through the thread of the current node, then the right references of nodes in the chain are reversed, and finally, the chain is scanned backward, each node is visited, and then right references are restored to their previous setting.

Traversal Through Tree Transformation

The first set of traversal algorithms analyzed earlier in this chapter needed a stack to retain some information necessary for successful processing. Threaded trees incorporated a stack as part of the tree at the cost of extending the nodes by one field to make a distinction between the interpretation of the right pointer as a pointer to the child or to the successor. Two such tag fields are needed if both successor and predecessor are considered. However, it is possible to traverse a tree without using any stack or threads. There are many such algorithms, all of them made possible by making temporary changes in the tree during traversal. These changes consist of reassigning new values to some pointers. However, the tree may temporarily lose its tree structure which needs to be restored before traversal is finished. The technique is illustrated by an elegant algorithm devised by Joseph M. Morris applied to inorder traversal.

First, it is easy to notice that inorder traversal is very simple for degenerate trees, in which no node has a left child (see Figure 6.1e). No left subtree has to be considered for any node. Therefore, the usual three steps, LVR (visit left subtree, visit node, visit right subtree), for each node in inorder traversal turn into two steps, VR. No information needs to be retained about the current status of the node being processed before traversing its left child, simply because there is no left child. Morris's algorithm takes into account this observation by temporarily transforming the tree so that the node being processed has no left child; hence, this node can be visited and its right subtree processed. The algorithm can be summarized as follows:

```
MorrisInorder()
    while not finished
      if node has no left descendant
        visit it;
        go to the right;
      else make this node right child of the rightmost node in its left descendant;
        go to this left descendant;
```

This algorithm successfully traverses the tree but only once, since it destroys its original structure. Therefore, some information has to be retained to allow the tree to restore its original form. This is achieved by retaining the left pointer of the node moved down its right subtree as in the case of nodes 10 and 5 in Figure 6.21.

An implementation of the algorithm is shown in Figure 6.20, and the details of the execution are illustrated in Figure 6.21. The following description is divided into actions performed in consecutive iterations of the outer `while` loop:

1. Initially, `p` points to the root, which has a left child. As a result, the inner `while` loop takes `tmp` to node 7, which is the rightmost node of the left child of node 10, pointed to by `p` (Figure 6.21a). Since no transformation has been done, `tmp` has no right child, and in the inner `if` statement, the root, node 10, is made the right child of `tmp`. Node 10 retains its left pointer to node 5, its original left child. Now, the tree is not a tree anymore, since it contains a cycle (Figure 6.21b). This completes the first iteration.
2. Pointer `p` points to node 5, which also has a left child. First, `tmp` reaches the largest node in this subtree, which is 3 (Figure 6.21c), and then the current root, node 5,

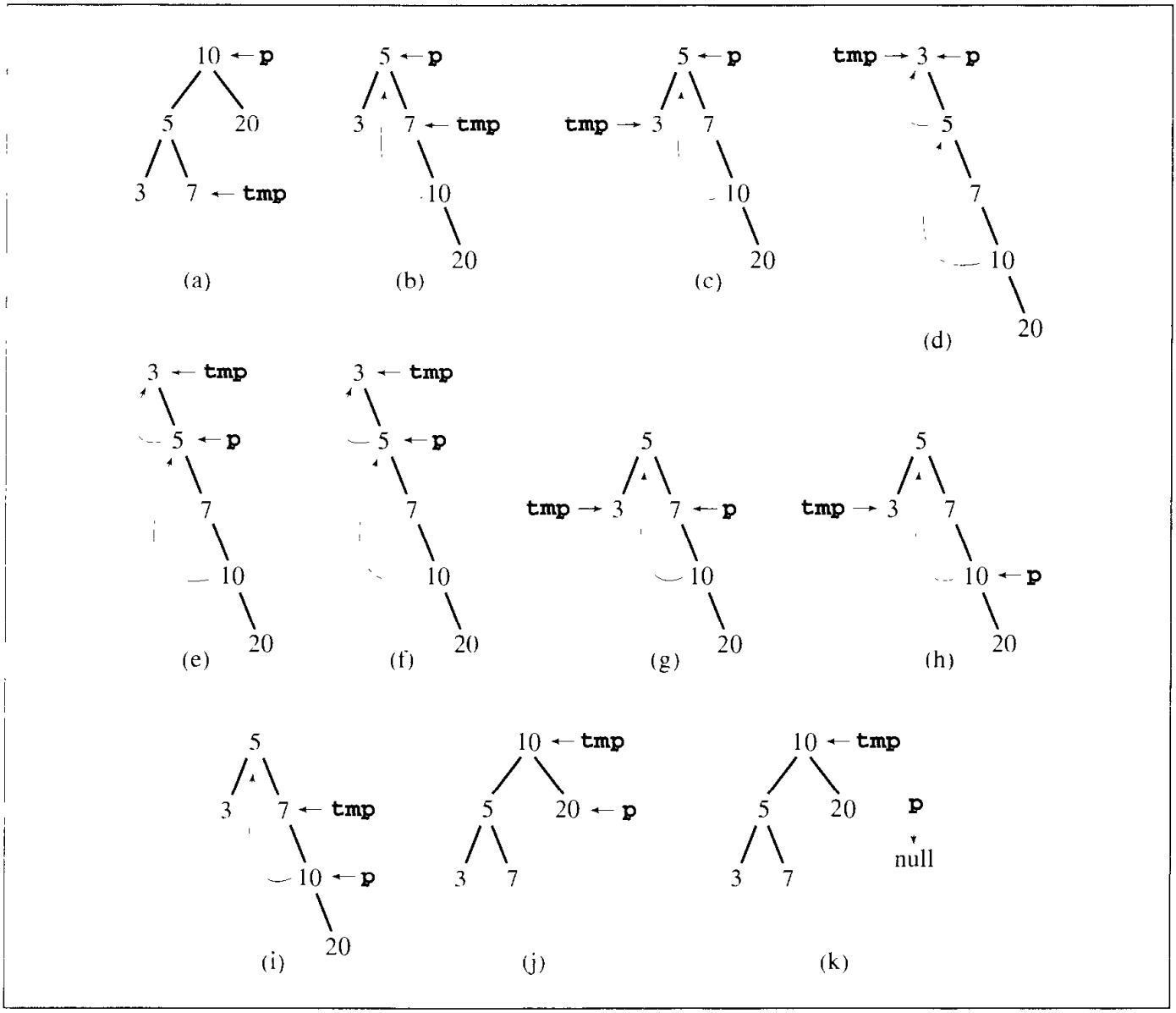
FIGURE 6.20 Implementation of the Morris algorithm for inorder traversal.

```
template<class T>
void BST<T>::MorrisInorder() {
    BSTNode<T> *p = root, *tmp;
    while (p != 0)
        if (p->left == 0) {
            visit(p);
            p = p->right;
        }
        else {
            tmp = p->left;
            while (tmp->right != 0 && // go to the rightmost node
                   tmp->right != p)   // of the left subtree or
                tmp = tmp->right;     // to the temporary parent
            if (tmp->right == 0) {    // of p; if 'true'
                tmp->right = p;       // rightmost node was
                p = p->left;          // reached, make it a
            }                         // temporary parent of the
            else {                    // current root, else
                                      // a temporary parent has
                visit(p);             // been found; visit node p
                tmp->right = 0;       // and then cut the right
                                      // pointer of the current
                p = p->right;         // parent, whereby it
            }                         // ceases to be a parent;
        }
}
```

becomes the right child of node 3 while retaining contact with node 3 through its left pointer (Figure 6.21d).

3. Because node 3, pointed to by p, has no left child, in the third iteration, this node is visited, and p is reassigned to its right child, node 5 (Figure 6.21e).
4. Node 5 has a nonnull left pointer, so tmp finds a temporary parent of node 5, which is the same node currently pointed to by tmp (Figure 6.21f). Next, node 5 is visited, and configuration of the tree in Figure 6.21b is reestablished by setting the right pointer of node 3 to null (Figure 6.21g).
5. Node 7, pointed to now by p, is visited, and p moves down to its right child (6.21h).
6. tmp is updated to point to the temporary parent of node 10 (Figure 6.21i). Next, node 10 is visited and then reestablished to its status of root by nullifying the right pointer of node 7 (Figure 6.21j).
7. Finally, node 20 is visited without further ado, since it has no left child, nor has its position been altered.

FIGURE **6.21** Tree traversal with the Morris method.



This completes the execution of Morris's algorithm. Notice that there are seven iterations of the outer `while` loop for only five nodes in the tree in Figure 6.21. This is due to the fact that there are two left children in the tree, so the number of extra iterations depends on the number of left children in the entire tree. The algorithm performs worse for trees with a large number of such children.

Preorder traversal is easily obtainable from inorder traversal by moving `visit()` from inner `else` clause to the inner `if` clause. In this way, a node is visited before a tree transformation.

Postorder traversal can also be obtained from inorder traversal by first creating a dummy node whose left descendant is the tree being processed and whose right descendant is null. Then this temporarily extended tree is a subject of traversal as in inorder

traversal except that in the inner `else` clause, after finding a temporary parent, nodes between `p->left` (included) and `p` (excluded) extended to the right in a modified tree are processed in the reverse order. To process them in constant time, the chain of nodes is scanned down and right pointers are reversed to point to parents of nodes. Then the same chain is scanned upward, each node is visited, and the right pointers are restored to their original setting.

How efficient are the traversal procedures discussed in this section? All of them run in $\Theta(n)$ time, threaded implementation requires $\Theta(n)$ more space for threads than nonthreaded binary search trees, and both recursive and iterative traversals require $O(n)$ additional space (on the run-time stack or user-defined stack). Several dozens of runs on randomly generated trees of 5000 nodes indicate that for preorder and inorder traversal routines (recursive, iterative, Morris, and threaded), the difference in the execution time is only on the order of 5–10%. Morris traversals have one undeniable advantage over other types of traversals: They do not require additional space. Recursive traversals rely on the run-time stack which can be overflowed when traversing trees of large height. Iterative traversals also use a stack, and although the stack can be overflowed as well, the problem is not as imminent as in the case of the run-time stack. Threaded trees use nodes that are larger than the nodes used by nonthreaded trees, which usually should not pose a problem. But both iterative and threaded implementations are much less intuitive than their recursive counterparts; therefore, the clarity of implementation and comparable run time clearly favors in most situations recursive implementations over other implementations.

6.5 Insertion

Searching a binary tree does not modify the tree. It scans the tree in a predetermined way to access some or all of the keys in the tree, but the tree itself remains undisturbed after such an operation. Tree traversals can change the tree but they may also leave it in the same condition. Whether or not the tree is modified depends on the actions prescribed by `visit()`. There are certain operations that always make some systematic changes in the tree, such as adding nodes, deleting them, modifying elements, merging trees, and balancing trees to reduce their height. This section deals only with inserting a node into a binary search tree.

To insert a new node, called *n_node*, a tree node, called *t_node*, with a dead end has to be reached, and the new node has to be attached to it. A *t_node* is found using the same technique that tree searching used; the key of the *n_node* to be inserted is compared to the value of a node, denoted as *c_node*, currently being examined during a tree scan. If it is less than that value, the left child (if any) is tried; otherwise, the right child is tested. If the child of the *c_node* to be tested is empty, the scanning is discontinued and the *n_node* becomes this child. The procedure is illustrated in Figure 6.22. Figure 6.23 contains the algorithm to insert a node.

In analyzing the problem of traversing binary trees, three approaches have been presented: traversing with the help of a stack, traversing with the aid of threads, and

FIGURE **6.22** Inserting nodes into binary search trees.

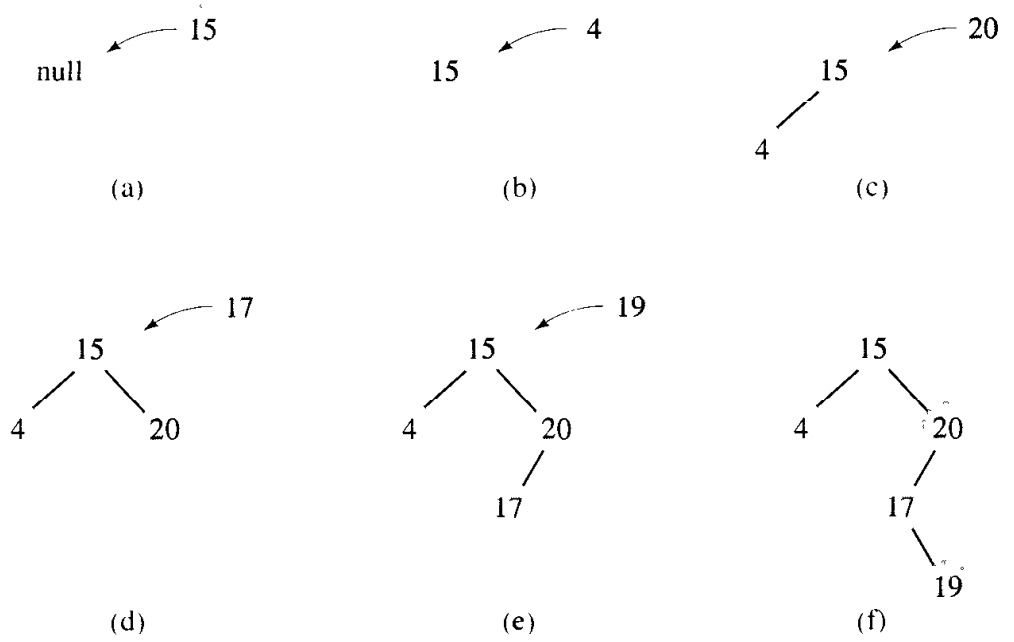


FIGURE **6.23** Implementation of the insertion algorithm.

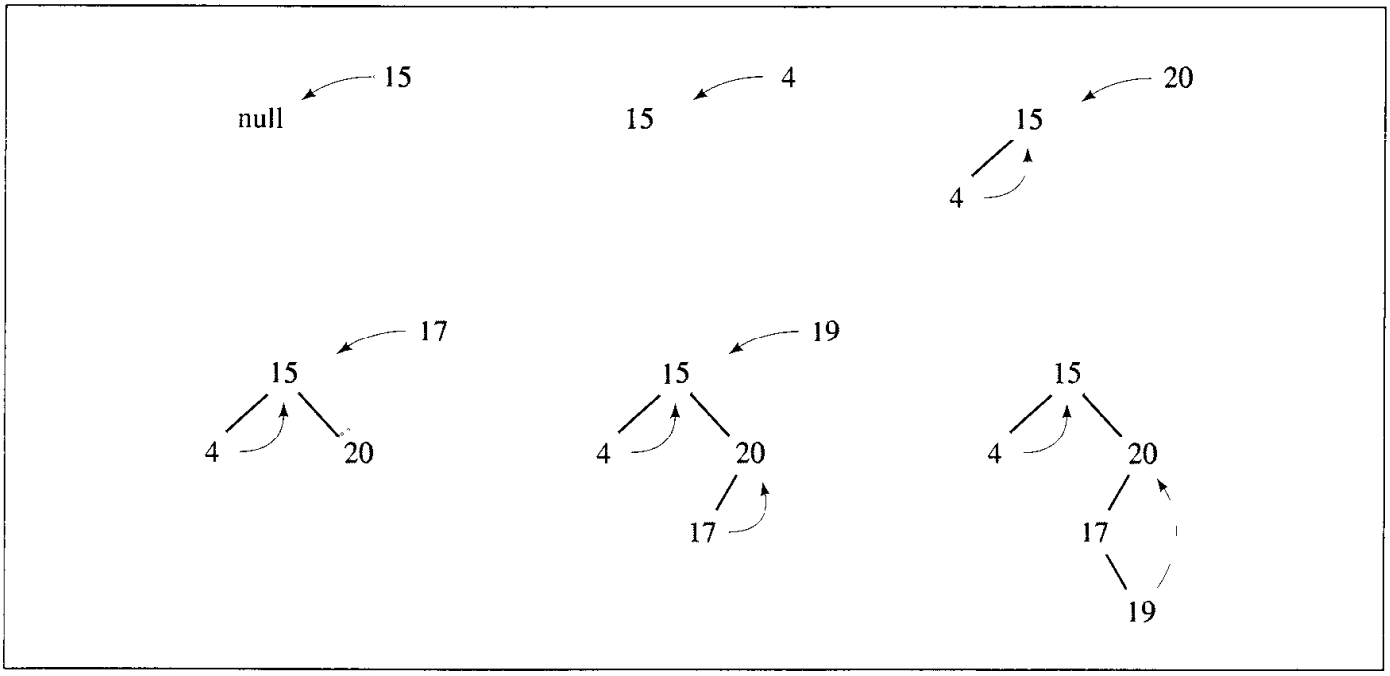
```
template<class T>
void BST<T>::insert(const T& el) {
    BSTNode<T> *p = root, *prev = 0;
    while (p != 0) {        // find a place for inserting new node;
        prev = p;
        if (p->key < el)
            p = p->right;
        else p = p->left;
    }
    if (root == 0)    // tree is empty;
        root = new BSTNode<T>(el);
    else if (prev->key < el)
        prev->right = new BSTNode<T>(el);
    else prev->left  = new BSTNode<T>(el);
}
```

FIGURE 6.24 Implementation of the algorithm to insert node nto a threaded tree.

```
template<class T>
void ThreadedTree<T>::insert(const T& el) {
    ThreadedNode<T> *p, *prev = 0, *newNode;
    newNode = new ThreadedNode<T>(el);
    if (root == 0) {          // tree is empty;
        root = newNode;
        return;
     }
     p = root;                // find a place to insert newNode;
     while (p != 0) {
         prev = p;
         if (p->key > el)
              p = p->left;
         else if (p->successor == 0) // go to the right node only if it
              p = p->right;   // is a descendant, not a successor;
         else break;          // don't follow successor link;
     }
     if (prev->key > el) {    // if newNode is left child of
          prev->left  = newNode;     // its parent, the parent
          newNode->successor = 1;    // also becomes its successor;
          newNode->right = prev;
     }
     else if (prev->successor == 1) {// if the parent of newNode
          newNode->successor = 1;    // is not the rightmost node,
          prev->successor = 0;       // make parent's successor
          newNode->right = prev->right; // newNode's successor,
          prev->right = newNode;
     }
     else prev->right = newNode;     // otherwise it has no successor;
}
```

traversing through tree transformation. The first approach does not change the tree during the process. The third approach changes it, but restores it to the same condition as before it started. Only the second approach needs some preparatory operations on the tree to become feasible: It requires threads. These threads may be created each time before the traversal procedure starts its task and removed each time it is finished. If the traversal is performed infrequently, this becomes a viable option. Another approach is to maintain the threads in all operations on the tree when inserting a new element in the binary search tree.

FIGURE 6.25 Inserting nodes into a threaded tree.



The function for inserting a node in a threaded tree is a simple extension of `insert()` for regular binary search trees to adjust threads whenever applicable. This function is for inorder tree traversal and it only takes care of successors, not predecessors.

A node with a right child has a successor some place in its right subtree. Therefore, it does not need a successor thread. Such threads are needed to allow climbing the tree, not going down it. A node with no right child has its successor somewhere above it. Except for one node, all nodes with no right children will have threads to their successors. If a node becomes the right child of another node, it inherits the successor from its new parent. If a node becomes a left child of another node, this parent becomes its successor. Figure 6.24 contains the implementation of this algorithm. The first few insertions are shown in Figure 6.25.

6.6 DELETION

Deleting a node is another operation necessary to maintain a binary search tree. The level of complexity in performing the operation depends on the position of the node to be deleted in the tree. It is by far more difficult to delete a node having two subtrees than to delete a leaf; the complexity of the deletion algorithm is proportional to the number of children the node has. There are three cases of deleting a node from the binary search tree:

1. The node is a leaf; it has no children. This is the easiest case to deal with. The appropriate pointer of its parent is set to null and the node is disposed of by `delete` as in Figure 6.26.

FIGURE 6.26 Deleting a leaf.

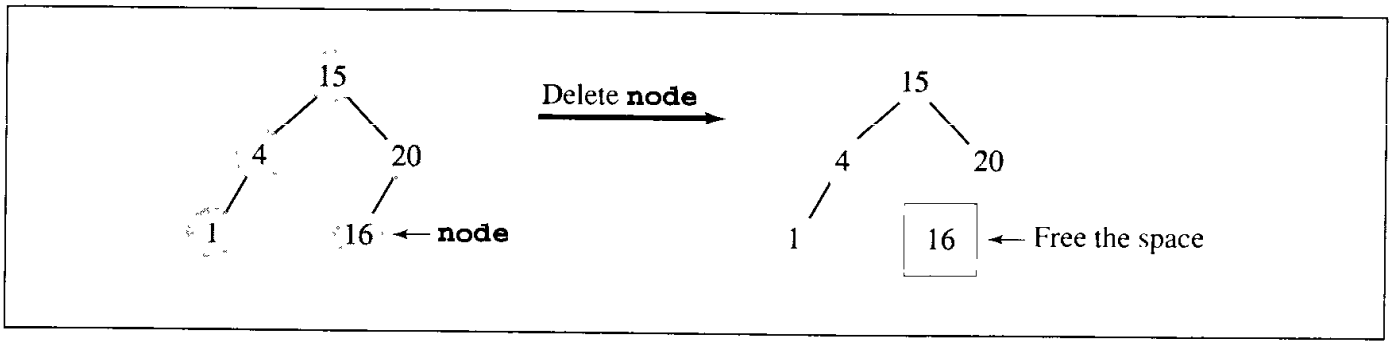
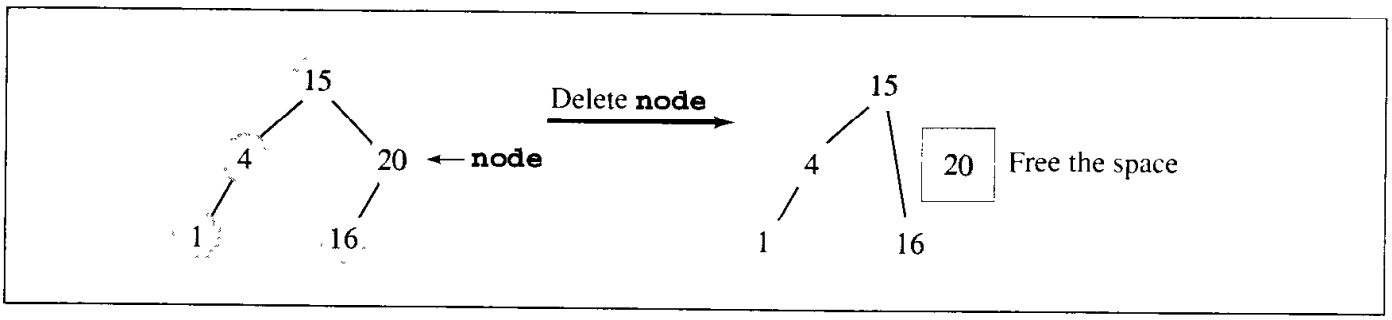


FIGURE 6.27 Deleting a node with one child.



2. The node has one child. This case is not complicated. The parent's pointer to the node is reset to point to the node's child. In this way, the node's children are lifted up by one level and all great-great-. . . grandchildren lose one "great" from their kinship designations. For example, the node containing 20 (see Figure 6.27) is deleted by setting the right pointer of its parent containing 15 to point to 20's only child, which is 16.
3. The node has two children. In this case, no one-step operation can be performed since the parent's right or left pointer cannot point to both node's children at the same time. This section discusses two different solutions to this problem.

6.6.1 Deletion by Merging

This solution makes one tree out of the two subtrees of the node and then attaches it to the node's parent. This technique is called *deleting by merging*. But how can we merge these subtrees? By the nature of binary search trees, every value of the right subtree is greater than every value of the left subtree, so the best thing to do is to find in the left subtree the node with the greatest value and make it a parent of the right subtree. Symmetrically, the node with the lowest value can be found in the right subtree and made a parent of the left subtree.

The desired node is the rightmost node of the left subtree. It can be located by moving along this subtree and taking right pointers until null is encountered. This

means that this node will not have a right child, and there is no danger of violating the property of binary search trees in the original tree by setting that rightmost node's right pointer to the right subtree. The same could be done by setting the left pointer of the leftmost node of the right subtree to the left subtree. Figure 6.28 depicts this operation. Figure 6.29 contains the implementation of the algorithm.

FIGURE **6.28** Summary of deleting by merging.

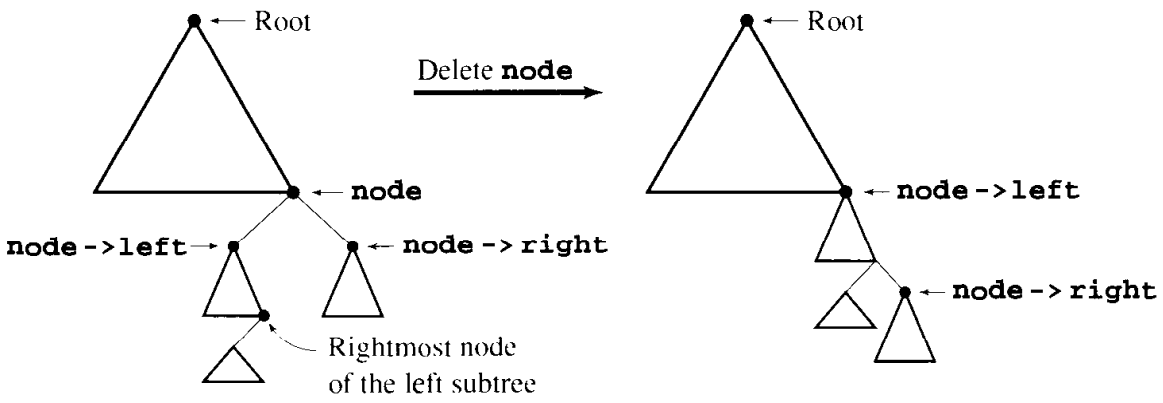


FIGURE **6.29** Implementation of algorithm for deleting by merging.

```
template<class T>
void BST<T>::deleteByMerging(BSTNode<T>*& node) {
    BSTNode<T> *tmp = node;
    if (node != 0) {
        if (!node->right)           // node has no right child: its left
            node = node->left;      // child (if any) is attached to its
                                    // parent;
        else if (node->left == 0)   // node has no left child: its right
            node = node->right;     // child is attached to its parent;
        else {                      // be ready for merging subtrees;
            tmp = node->left;       // 1. move left
            while (tmp->right == 0) // 2. and then right as far as
                                    // possible;
               tmp = tmp->right;
            tmp->right =            // 3. establish the link between
               node->right;         //    the rightmost node of the left
```

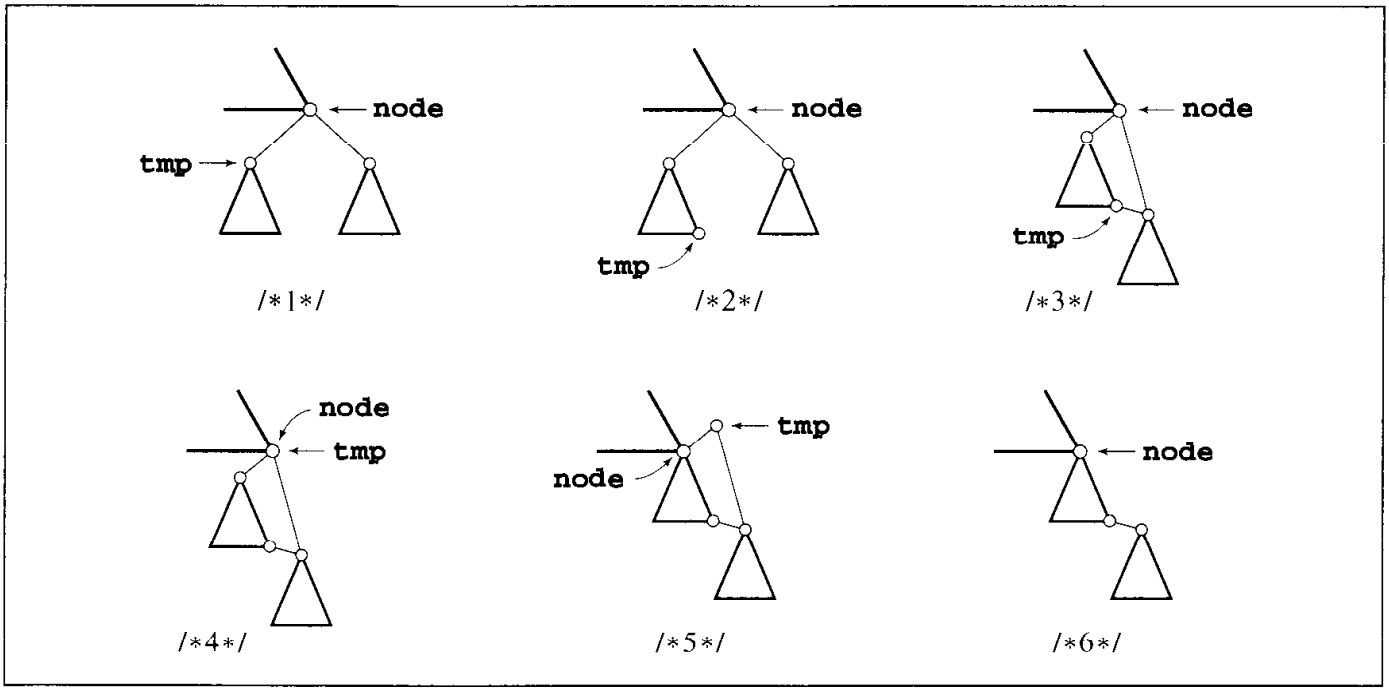
Continues

FIGURE 6.29 *(continued)*

```
                                        //    subtree and the right subtree;
            tmp = node;                 // 4.
            node = node->left;          // 5.
        }
        delete tmp;                     // 6.
    }
}

template<class T>
void BST<T>::findAndDeleteByMerging(const T& el) {
    BSTNode<T> *node = root, *prev = 0;
    while (node != 0) {
        if (node->key == el)
            break;
        prev = node;
        if (node->key < el)
            node = node->right;
        else node = node->left;
    }
    if (node != 0 && node->key == el)
        if (node == root)
            deleteByMerging(root);
        else if (prev->left == node)
            deleteByMerging(prev->left);
        else deleteByMerging(prev->right);
    else if (root != 0)
        cout << "key " << el << " is not in the tree\n";
    else cout << "the tree is empty\n";
}
```

It may appear that `findAndDeleteByMerging()` contains redundant code. Instead of calling `search()` before invoking `deleteByMerging()`, `findAndDeleteByMerging()` seems to forget about `search()` and searches for the node to be deleted using its private code. But using `search()` in function `findAndDeleteByMerging()` is a treacherous simplification. `search()` returns a pointer to the node containing `key`. In `findAndDeleteByMerging()`, it is important to have this pointer stored specifically in one of the pointers of the node's parent. In other words, a caller to `search()` is satisfied if it can access the node from any direction whereas `findAndDeleteByMerging()` wants to access it either from its parent's left or right pointer data member. Otherwise, access to the entire subtree having this node as its

FIGURE 6.30 Details of deleting by merging.



root would be lost. One reason for this is the fact that `search()` focuses on the node's key, and `findAndDeleteByMerging()` focuses on the node itself as an element of a larger structure, namely, a tree.

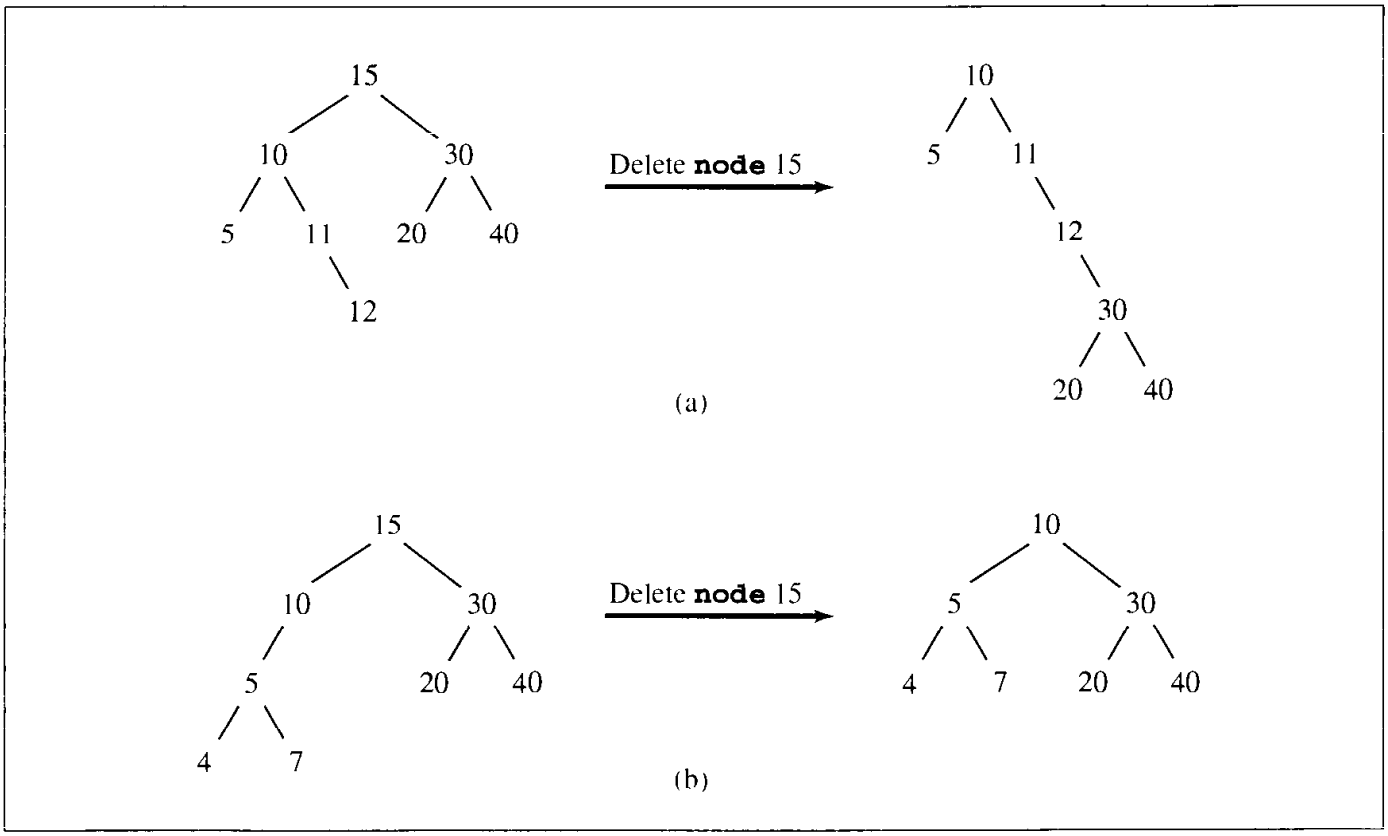
Figure 6.30 shows each step of this operation. It shows what changes are made when `findAndDeleteByMerging()` is executed. The numbers in this figure correspond to numbers put in comments in the code in Figure 6.29.

The algorithm for deletion by merging may result in increasing the height of the tree. In some cases, the new tree may be highly unbalanced, as Figure 6.31a illustrates. Sometimes the height may be reduced (see Figure 6.31b). This algorithm is not necessarily inefficient, but it is certainly far from perfect. There is a need for an algorithm that does not give the tree the chance to increase its height when deleting one of its nodes.

6.6.2 Deletion by Copying

Another solution, called *deletion by copying*, was proposed by Thomas Hibbard and Donald Knuth. If the node has two children, it can be reduced to one of two simple cases: The node is a leaf or the node has only one nonempty child. This can be done by replacing the key being deleted with its immediate predecessor (or successor). As already indicated in the algorithm deletion-by-merging, a key's predecessor is the key in the rightmost node in the left subtree (and analogically, its immediate successor is the key in the leftmost node in the right subtree). First, the predecessor has to be located. This is done, again, by moving one step to the left by first reaching the root of the node's left subtree and then moving as far to the right as possible. Next, the key of the located node replaces the key to be deleted. And that is where one of two simple cases

FIGURE 6.31 The height of a tree can be (a) extended or (b) reduced after deleting by merging.



comes into play. If the rightmost node is a leaf, the first case applies; however, if it has one child, the second case is relevant. In this way, deletion by copying removes a key k_1 by overwriting it by another key k_2 and then removing the node that holds k_2, whereas deletion by merging consisted of removing a key k_1 along with the node that holds it.

An implementation of this algorithm is in Figure 6.32. A step-by-step trace is shown in Figure 6.33, and the numbers under the diagrams refer to the numbers indicated in comments included in the implementation of `deleteByCopying()`.

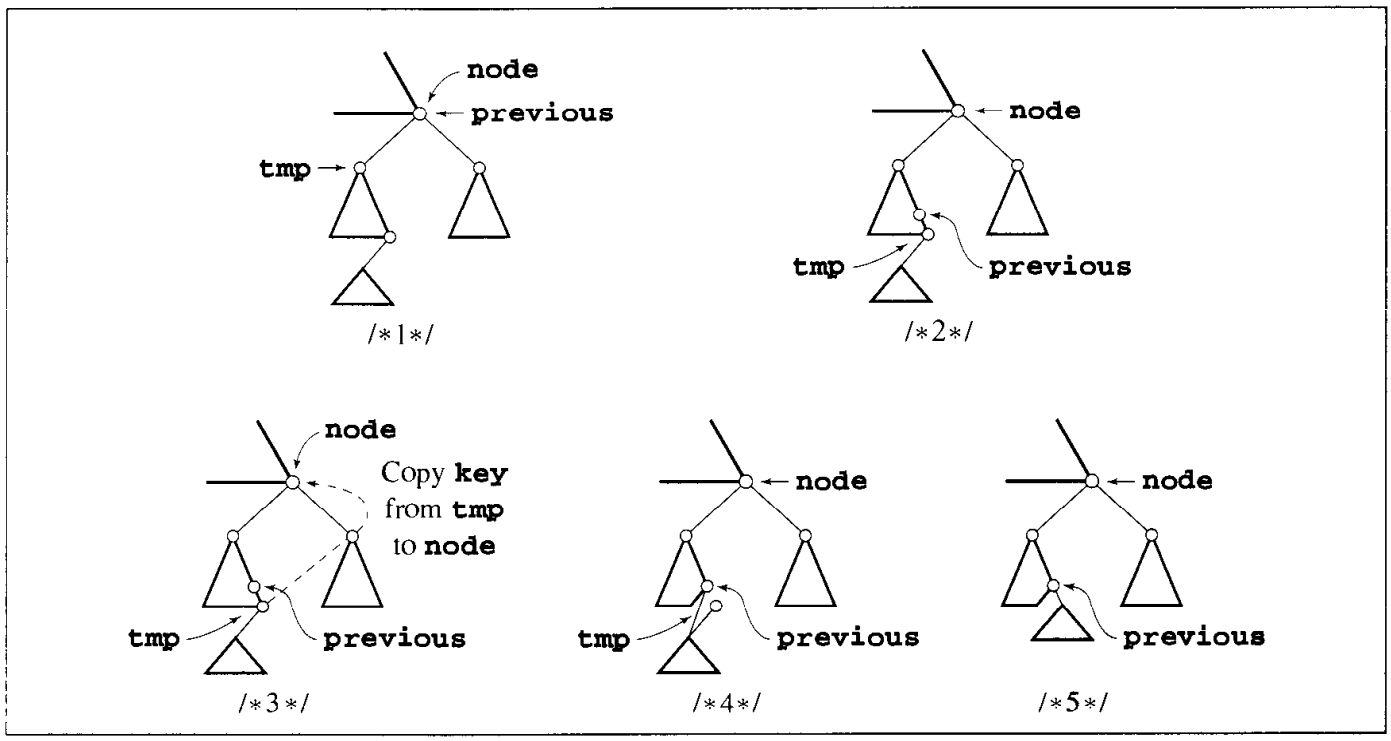
This algorithm does not increase the height of the tree, but it still causes a problem if it is applied many times along with insertion. The algorithm is asymmetric; it always deletes the node of the immediate predecessor of information in node, possibly reducing the height of the left subtree and leaving the right subtree unaffected. Therefore, the right subtree of node can grow after later insertions, and if the information in node is again deleted, the height of the right tree remains the same. After many insertions and deletions, the entire tree becomes right unbalanced, with the right tree bushier and larger than the left subtree.

To circumvent this problem, a simple improvement can make the algorithm symmetrical. The algorithm can alternately delete the predecessor of the information in node from the left subtree and delete its successor from the right subtree. The improvement is significant. Simulations performed by Jeffrey Eppinger show that an expected internal path length for many insertions and asymmetric deletions is

FIGURE **6.32** Implementation of algorithm for deleting by copying.

```
template<class T>
void BST<T>::deleteByCopying(BSTNode<T>*& node) {
    BSTNode<T> *prev, *tmp = node;
     if (node->right == 0)                // node has no right child;
          node = node->left;
     else if (node->left == 0)            // node has no left child;
          node = node->right;
     else {
          tmp = node->left;               // node has both children;
          prev = node;                    // 1.
          while (tmp->right != 0) {       // 2.
              prev = tmp;
              tmp = tmp->right;
          }
          node->key = tmp->key;           // 3.
          if (prev == node)
               prev->left  = tmp->left;   // 4.
          else prev->right = tmp->left;   // 5.
     }
     delete tmp;
}
```

FIGURE **6.33** Deleting by copying.



$\Theta(n \lg^3 n)$ for n nodes, and when symmetric deletions are used, the expected IPL becomes $\Theta(n \lg n)$. Theoretical results obtained by J. Culberson confirm these conclusions. According to Culberson, insertions and asymmetric deletions give $\Theta(n\sqrt{n})$ for the expected IPL and $\Theta(\sqrt{n})$ for the average search time (average path length), whereas symmetric deletions lead to $\Theta(\lg n)$ for the average search time, and as before, $\Theta(n \lg n)$ for the average IPL.

These results may be of moderate importance for practical applications. Experiments show that for a 2048-node binary tree, only after 1.5 million insertions and asymmetric deletions does the IPL become worse than in a randomly generated tree.

Theoretical results are only fragmentary because of the extraordinary complexity of the problem. Arne Jonassen and Donald Knuth analyzed the problem of random insertions and deletions for a tree of only three nodes, which required using Bessel functions and bivariate integral equations, and the analysis turned out to rank among "the more difficult of all exact analyses of algorithms that have been carried out to date." Therefore, the reliance on experimental results is not surprising.

6.7 Balancing a Tree

At the beginning of this chapter, two arguments were presented in favor of trees: They are well suited to represent the hierarchical structure of a certain domain, and the search process is much faster using trees instead of linked lists. The second argument, however, does not always hold. It all depends on what the tree looks like. Figure 6.34 shows three binary search trees. All of them store the same data, but obviously, the tree in Figure 6.34a is the best and Figure 6.34c is the worst. In the worst case, three tests are needed in the former and six tests are needed in the latter to locate an object. The problem with the trees in Figures 6.34b and 6.34c is that they are somewhat unsymmetrical, or lopsided; that is, objects in the tree are not distributed evenly to the extent that the tree in Figure 6.34c practically turned into a linked list, although, formally, it is still a tree. Such a situation does not arise in balanced trees.

A binary tree is *height-balanced* or simply *balanced* if the difference in height of both subtrees of any node in the tree is either zero or one. For example, for node K in Figure 6.34b, the difference between the heights of its subtrees being equal to one is acceptable. But for node B this difference is three, which means that the entire tree is unbalanced. For the same node B in 6.34c, the difference is the worst possible, namely, five. Also, a tree is considered *perfectly balanced* if it is balanced and all leaves are to be found on one level or two levels.

Figure 6.35 shows how many nodes can be stored in binary trees of different heights. Since each node can have two children, the number of nodes on a certain level is double the number of parents residing on the previous level (except, of course, the root). For example, if 10,000 elements are stored in a perfectly balanced tree, then the tree is of height $\lceil \lg(10{,}001) \rceil = \lceil 13.289 \rceil = 14$. In practical terms, this means that if 10,000 elements are stored in a perfectly balanced tree, then at most 14 nodes have to

FIGURE 6.34 Different binary search trees with the same information.

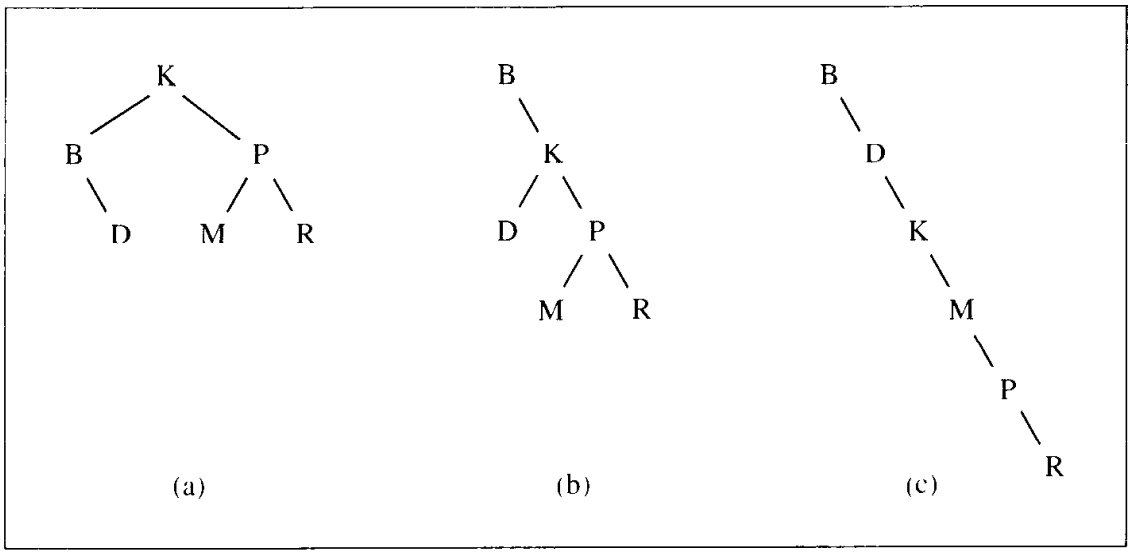
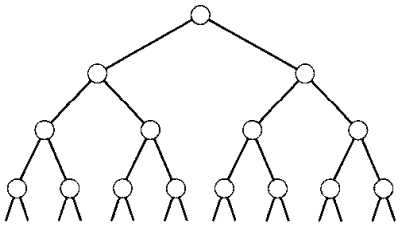


FIGURE 6.35 Maximum number of nodes in binary trees of different heights.



Height	*Nodes at one level*	*Nodes at all levels*
1	$2^0 = 1$	$1 = 2^1 - 1$
2	$2^1 = 2$	$3 = 2^2 - 1$
3	$2^2 = 4$	$7 = 2^3 - 1$
4	$2^3 = 8$	$15 = 2^4 - 1$
⋮		
11	$2^{10} = 1024$	$2047 = 2^{11} - 1$
⋮		
14	$2^{13} = 8192$	$16383 = 2^{14} - 1$
⋮		
h	2^{h-1}	$n = 2^h - 1$
⋮		

be checked to locate a particular element. This is a substantial difference compared to the 10,000 tests needed in a linked list (in the worst case). Therefore, it is worth the effort to build a balanced tree or modify an existing tree so that it is balanced.

There are a number of techniques to properly balance a binary tree. Some of them consist of constantly restructuring the tree when elements arrive and lead to an

unbalanced tree. Some of them consist of reordering the data themselves and then building a tree, if an ordering of the data guarantees that the resulting tree is balanced. This section presents a simple technique of this kind.

The linked listlike tree of Figure 6.34c is the result of a particular stream of data. Thus, if the data arrive in ascending or descending order, then the tree resembles a linked list. The tree in Figure 6.34b is lopsided because the first element that arrived was the letter B, which precedes almost all other letters, except A; the left subtree of B is guaranteed to have just one node. The tree in Figure 6.34a looks very good, since the root contains an element near the middle of all the possible elements, and P is more or less in the middle of K and Z. This leads us to an algorithm based on binary search technique.

When data arrive, store all of them in an array. If all possible data arrived, sort the array using one of the efficient algorithms discussed in Chapter 9. Now, designate for the root the middle element in the array. The array now consists of two subarrays: one between the beginning of the array and the element just chosen for the root and one between the root and the end of the array. The left child of the root is taken from the middle of the first subarray, its right child an element in the middle of the second. Now, building the level containing the children of the root is finished. The next level, with children of children of the root, is constructed in the same fashion using four subarrays and the middle elements from each of them.

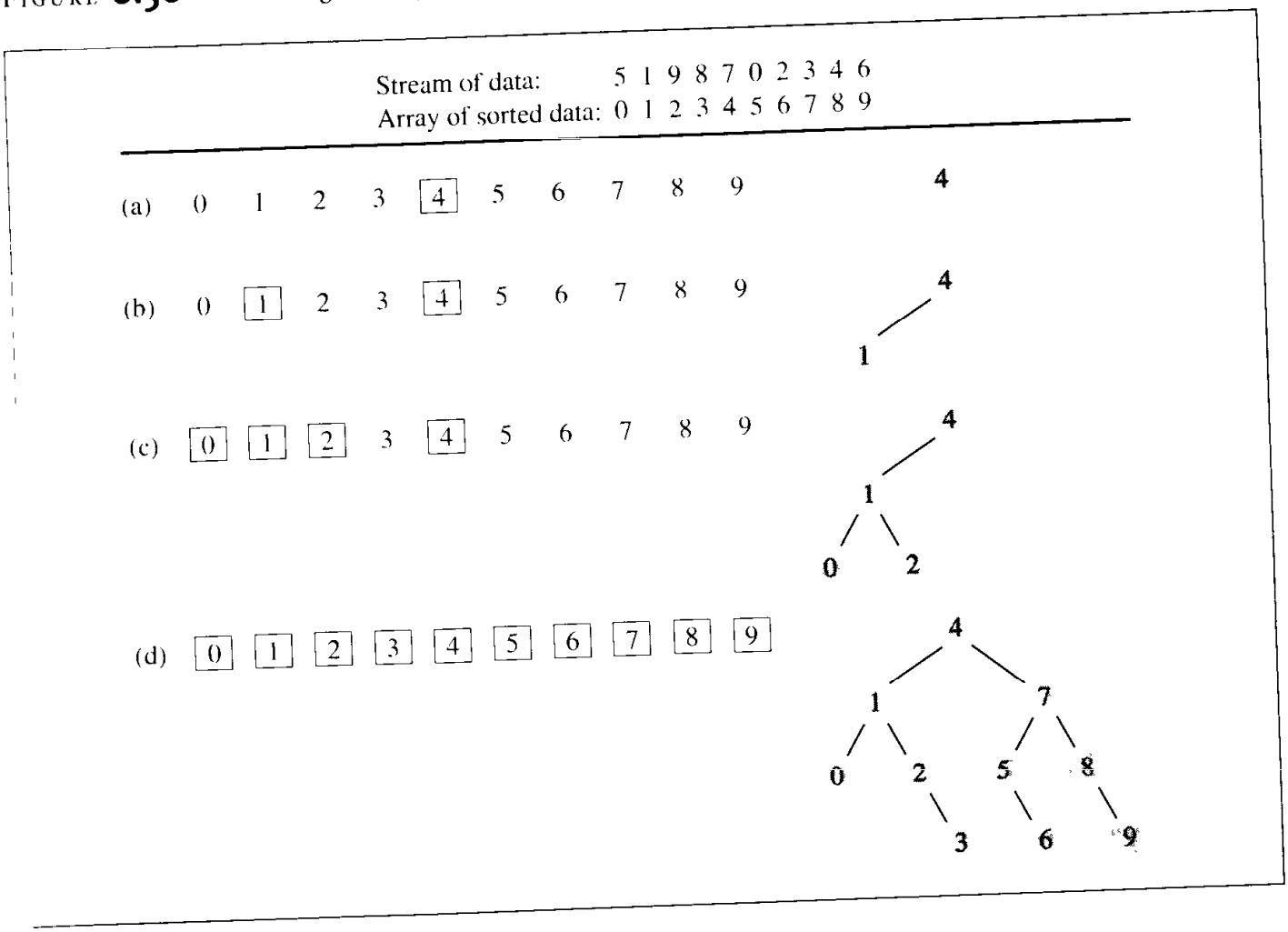
In this description, first the root is inserted into an initially empty tree, then its left child, then its right child, and so on level by level. An implementation of this algorithm is greatly simplified if the order of insertion is changed: First insert the root, then its left child, then left child of this left child, and so on. This allows for using the following simple recursive implementation:

```
template<class T>
void BST<T>::balance(T data[], int first, int last) {
  if (first <= last) {
     int middle = (first + last)/2;
     insert(data[middle]);
     balance (data,first,middle-1);
     balance (data,middle+1,last);
  }
}
```

An example of application of `balance()` is shown in Figure 6.36. First, number 4 is inserted (Figure 6.36a), then 1 (Figure 6.36b), then 0 and 2 (Figure 6.36c), and finally, 6, 5, 7, and 8 (Figure 6.36d).

This algorithm has one serious drawback: All data must be put in an array before the tree can be created. They can be stored in the array directly from the input. In this case, the algorithm may be unsuitable when the tree has to be used while the data to be included in the tree are still coming. But the data can be transferred from an unbalanced tree to the array using inorder traversal. The tree can now be deleted and recreated using `balance()`. This at least does not require using any sorting algorithm to put data in order.

FIGURE 6.36 Creating a binary search tree from an ordered array.



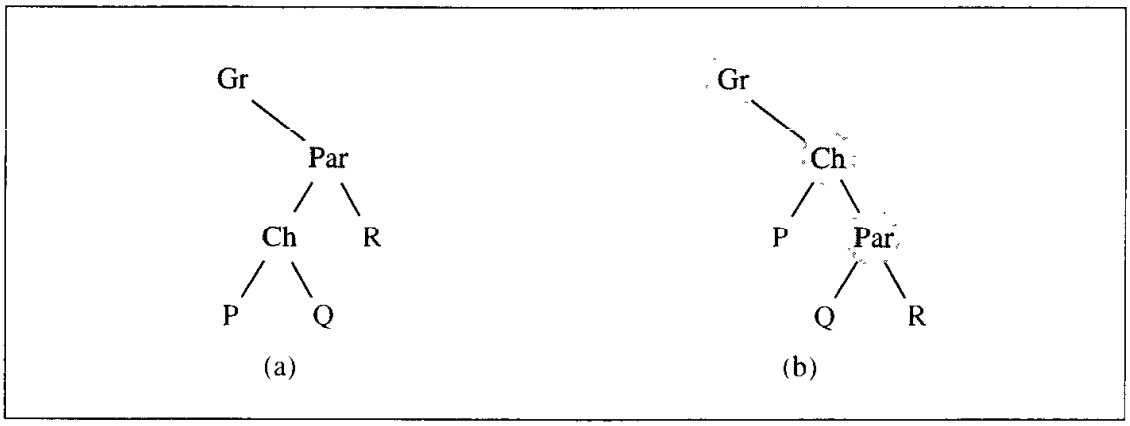
6.7.1 The DSW Algorithm

The algorithm discussed in the previous section was somewhat inefficient in that it required an additional array which needed to be sorted before the construction of a perfectly balanced tree began. To avoid sorting, it required deconstructing and then reconstructing the tree, which is inefficient except for relatively small trees. There are, however, algorithms that require little additional storage for intermediate variables and use no sorting procedure. The very elegant DSW algorithm was devised by Colin Day and later improved by Quentin F. Stout and Bette L. Warren.

The building block for tree transformations in this algorithm is the *rotation*. There are two types of rotation, left and right, which are symmetrical to one another. The right rotation of the node `Ch` about its parent `Par` is performed according to the following algorithm:

```
rotateRight (Gr, Par, Ch)
  if Par is not the root of the tree // i.e., if Gr is not null
    grandparent Gr of child Ch becomes Ch's parent by replacing Par;
  right subtree of Ch becomes left subtree of Ch's parent Par;
  node Ch acquires Par as its right child;
```

FIGURE 6.37 Right rotation of child Ch about parent Par.



The steps involved in this compound operation are shown in Figure 6.37. The third step is the core of the rotation, when `Par`, the parent node of child `Ch`, becomes the child of `Ch`, when the roles of a parent and its child change. However, this exchange of roles cannot affect the principal property of the tree, namely, that it is a search tree. The first and the second steps of `rotateRight()` are needed to ensure that, after the rotation, the tree remains a search tree.

Basically, the DSW algorithm transfigures an arbitrary binary search tree into a linked listlike tree called a *backbone* or *vine*. Then this elongated tree is transformed in a series of passes into a perfectly balanced tree by repeatedly rotating every second node of the backbone about its parent.

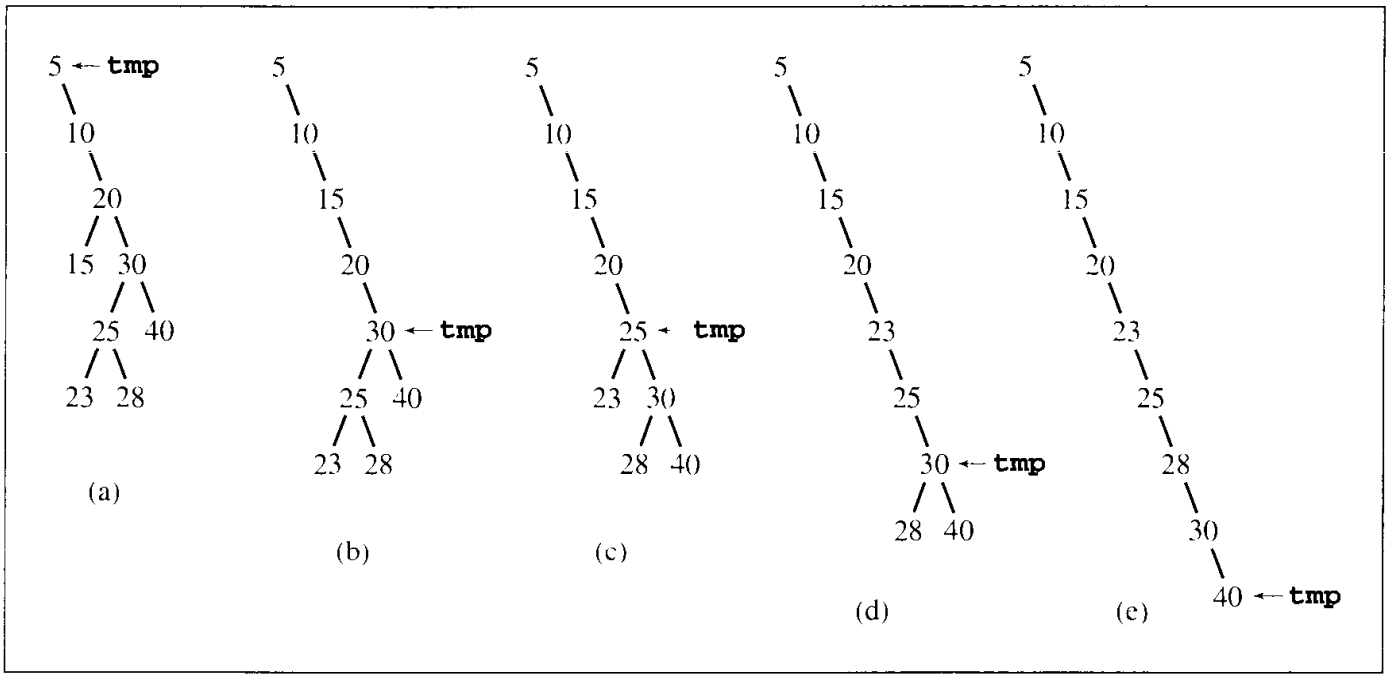
In the first phase, a backbone is created using the following routine:

```
createBackbone(root, n)
  tmp = root;
  while (tmp != 0)
    if tmp has a left child
        rotate this child about tmp; // hence the left child
                                     // becomes parent of tmp;
        set tmp to the child which just became parent;
    else set tmp to its right child;
```

This algorithm is illustrated in Figure 6.38. Note that a rotation requires knowledge about the parent of `tmp`, so another pointer has to be maintained when implementing the algorithm.

In the best case, when the tree is already a backbone, the `while` loop is executed n times and no rotation is performed. In the worst case, when the root does not have a right child, the `while` loop executes $2n - 1$ times with $n - 1$ rotations performed, where n is the number of nodes in the tree; that is, the run time of the first phase is $O(n)$. In this case, for each node except the one with the smallest value, the left child of `tmp` is rotated about `tmp`. After all rotations are finished, `tmp` points to the root, and after n iterations, it descends down the backbone to become null.

FIGURE 6.38 Transforming a binary search tree into a backbone.



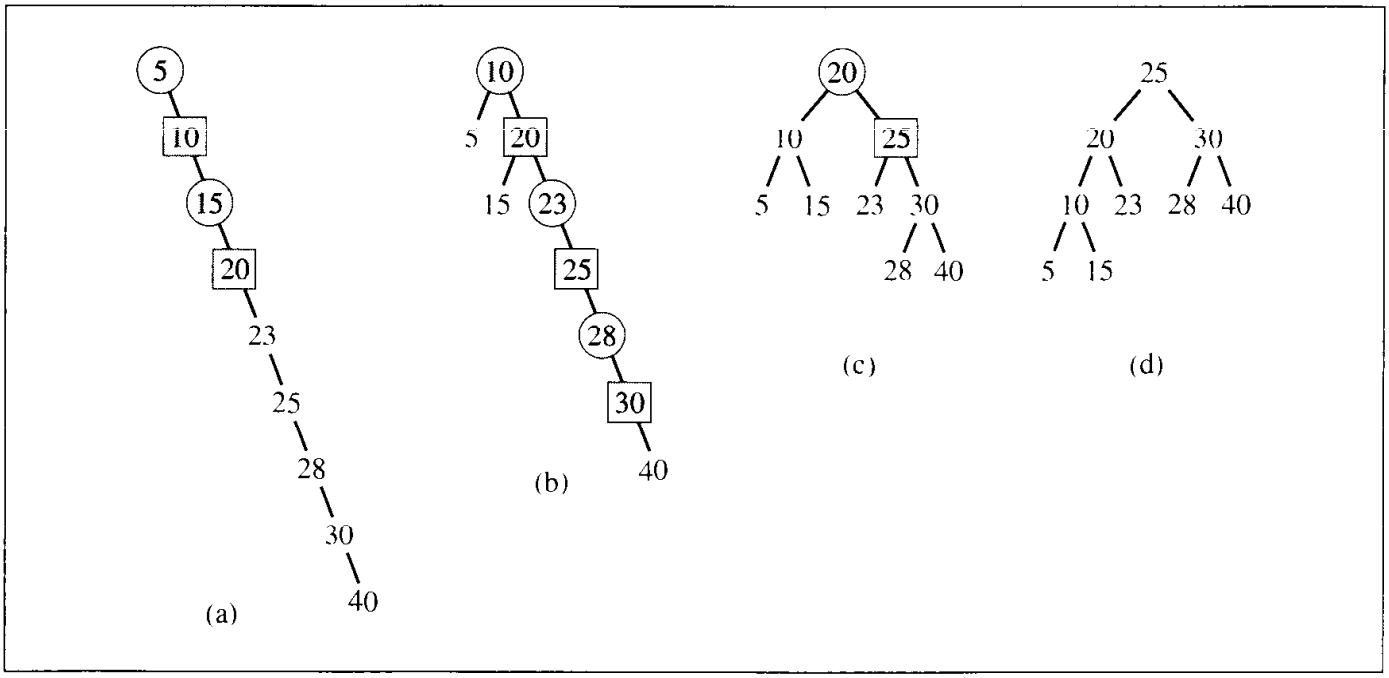
In the second phase, the backbone is transformed into a tree, but this time, the tree is perfectly balanced by having leaves only on two adjacent levels. In each pass down the backbone, every second node down to a certain point is rotated about its parent. The first pass is used to account for the difference between the number n of nodes in the current tree and the number $2^{\lfloor \lg(n+1) \rfloor} - 1$ of nodes in the closest complete binary tree where $\lfloor x \rfloor$ is the closest integer less than x. That is, the overflowing nodes are treated separately.

```
createPerfectTree(n)
  m = 2^⌊lg(n+1)⌋-1;
  make n-m rotations starting from the top of backbone;
  while (m > 1)
    m = m/2;
    make m rotations starting from the top of backbone;
```

Figure 6.39 contains an example. The backbone in Figure 6.38e has nine nodes and is preprocessed by one pass outside the loop to be transformed into the backbone shown in Figure 6.39b. Now, two passes are executed. In each backbone, the nodes to be promoted by one level by left rotations are shown as squares; their parents, about which they are rotated, are circles.

To compute the complexity of the tree building phase, observe that the number of iterations performed by the `while` loop equals

FIGURE 6.39 Transforming a backbone into a perfectly balanced tree.



$$(2^{\lg(m+1)-1}-1)+\cdots+15+7+3+1=\sum_{i=1}^{\lg(m+1)-1}(2^i-1)=m-\lg(m+1)$$

The number of rotations can now be given by the formula

$$n-m+(m-\lg(m+1))=n-\lg(m+1)=n-\lfloor\lg(n+1)\rfloor$$

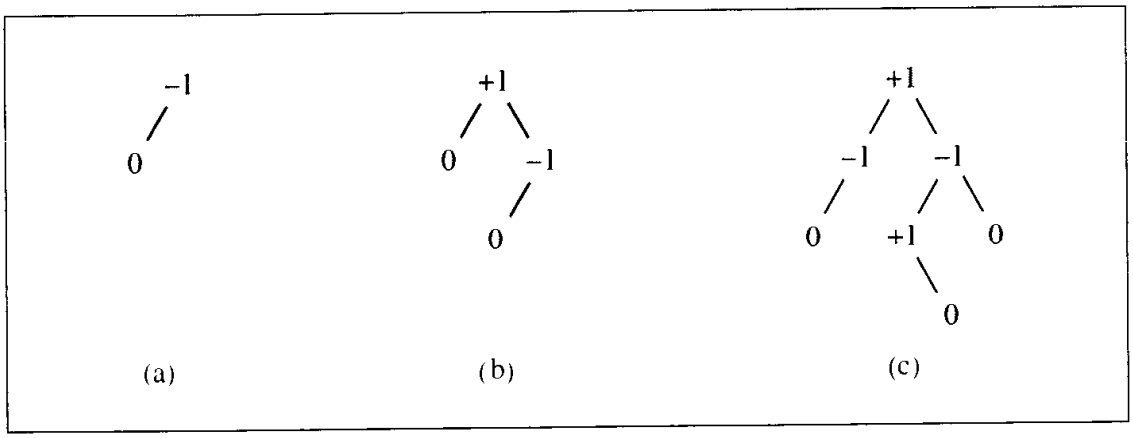
that is, the number of rotations is $O(n)$. Because creating a backbone also required at most $O(n)$ rotations, the cost of global rebalancing with the DSW algorithm is optimal in terms of time because it grows linearly with n and requires a very small and fixed amount of storage.

6.7.2 AVL Trees

The previous two sections discussed algorithms which rebalanced the tree globally; each and every node could have been involved in rebalancing either by moving data from nodes or by reassigning new values to pointers. Tree rebalancing, however, can be performed locally if only a portion of the tree is affected when changes are required after an element is inserted into or deleted from the tree. One classical method has been proposed by Adel'son-Vel'skii and Landis, which is commemorated in the name of the tree modified with this method: the AVL tree.

An *AVL tree* (originally called an *admissible tree*) is one in which the height of left and right subtrees of every node differ by at most one. For example, all the trees in Figure 6.40 are AVL trees. Numbers in the nodes indicate the *balance factors* which are the differences between the heights of the left and right subtrees. A balance factor is

FIGURE 6.40 Examples of AVL trees.



the height of the right subtree minus the height of the left subtree. For an AVL tree, all balance factors should be +1, 0, or −1. Notice that the definition of the AVL tree is the same as the definition of the balanced tree. However, the concept of the AVL tree always implicitly includes the techniques for balancing the tree. Moreover, unlike the two methods previously discussed, the technique for balancing AVL trees does not guarantee that the resulting tree is perfectly balanced.

The definition of an AVL tree indicates that the minimum number of nodes in a tree is determined by the recurrence equation

$$AVL_h = AVL_{h-1} + AVL_{h-2} + 1$$

where $AVL_0 = 0$ and $AVL_1 = 1$ are the initial conditions.[1] As shown by Adel'son-Vel'skii and Landis, this formula leads to the following bounds on the height h of an AVL tree depending on the number of nodes n:

$$\lg(n+1) \leq h < 1.44\lg(n+2) - 0.328$$

Therefore, h is bounded by $O(\lg n)$; the worst case search requires $O(\lg n)$ comparisons. For a perfectly balanced binary tree of the same height, $h = \lceil \lg(n+1) \rceil$. Therefore, the search time in the worst case in an AVL tree is 44% worse (it requires 44% more comparisons) than in the best case tree configuration. Empirical studies indicate that the average number of searches is much closer to the best case than to the worst and is equal to $\lg n + 0.25$ for large n (Knuth 1998). Therefore, AVL trees are definitely worth studying.

If the balance factor of any node in an AVL tree becomes less than −1 or greater than 1, the tree has to be balanced. An AVL tree can become out of balance in four situations, but only two of them need to be analyzed; the remaining two are symmetrical. The first case, the result of inserting a node in the right subtree of the right child, is illustrated in Figure 6.41. The heights of the participating subtrees are indicated within these subtrees. In the AVL tree in Figure 6.41a, a node is inserted somewhere in

[1] Numbers generated by this recurrence formula are called *Leonardo numbers.*

FIGURE 6.41 Balancing a tree after insertion of a node in the right subtree of node Q.

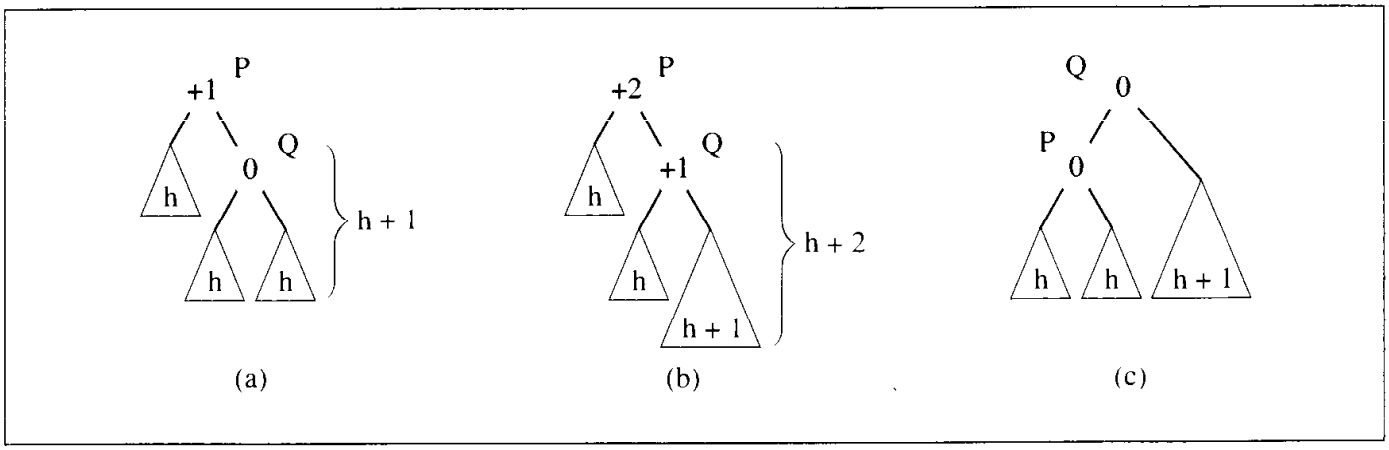
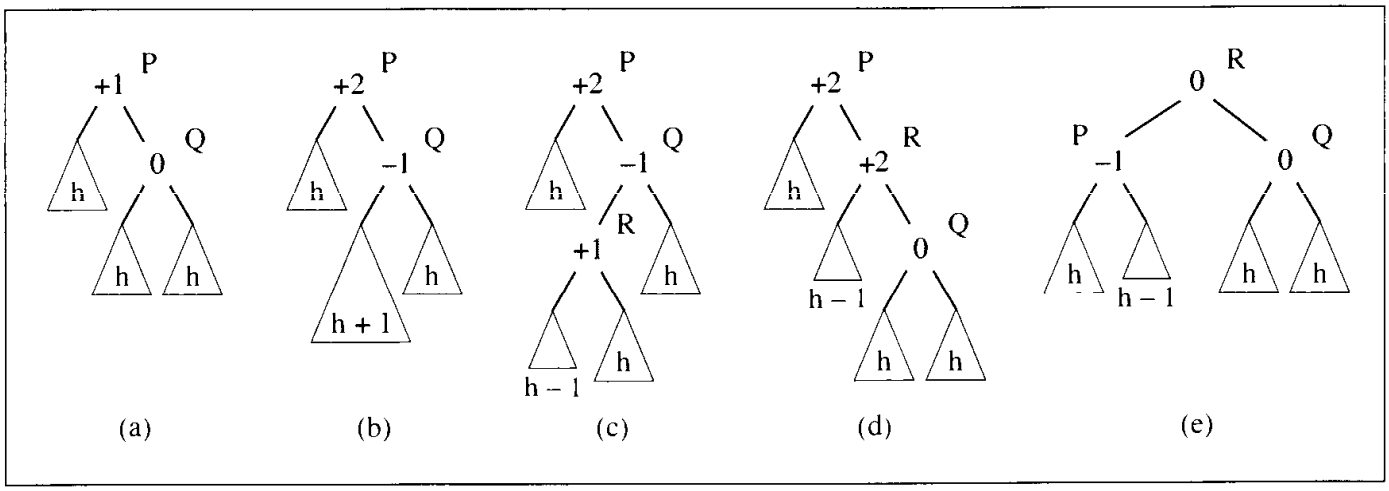


FIGURE 6.42 Balancing a tree after insertion of a node in the left subtree of node Q.

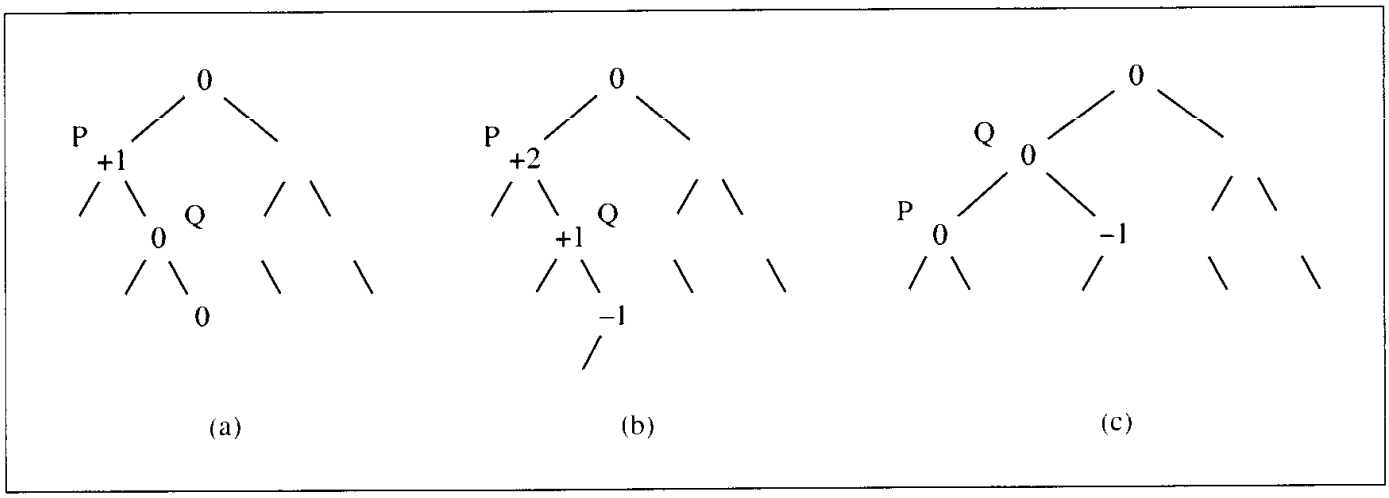


the right subtree of Q (Figure 6.41b), which disturbs the balance of the tree P. In this case, the problem can be easily rectified by rotating node Q about its parent P (Figure 6.41c) so that the balance factor of both P and Q becomes zero, which is even better than at the outset.

The second case, the result of inserting a node in the left subtree of the right child, is more complex. A node is inserted into the tree in Figure 6.42a; the resulting tree is shown in Figure 6.42b and in more detail in Figure 6.42c. To bring the tree back into balance, a double rotation is performed. The balance of the tree P is restored by rotating R about node Q (Figure 6.42d) and then by rotating R again, this time about node P (Figure 6.42e).

In these two cases, the tree P is considered a stand-alone tree. However, P can be part of a larger AVL tree; it can be a child of some other node in the tree. If a node is entered into the tree and the balance of P is disturbed and then restored, does extra

FIGURE 6.43 An example of inserting a new node (b) in an AVL tree (a), which requires one rotation (c) to restore the height balance.



work need to be done to the predecessor(s) of *P*? Fortunately not. Note that the heights of the trees in Figures 6.41c and 6.42e resulting from the rotations are the same as the heights of the trees before insertion (Figures 6.41a and 6.42a) and are equal to $h + 2$. This means that the balance factor of the parent of the new root (*Q* in Figure 6.41c and *R* in Figure 6.42e) remains the same as it was before the insertion, and the changes made to the subtree *P* are sufficient to restore the balance of the entire AVL tree. The problem is in finding a node *P* for which the balance factor becomes unacceptable after a node has been inserted into the tree.

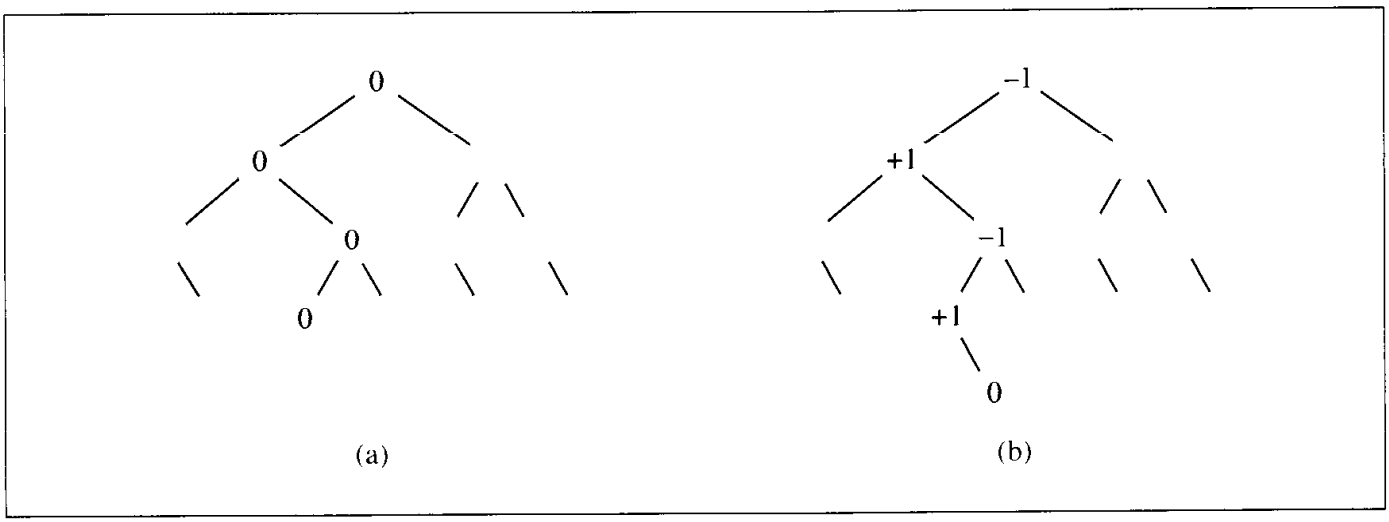
This node can be detected by moving up toward the root of the tree from the position in which the new node has been inserted and by updating the balance factors of the nodes encountered. Then, if a node with a ± 1 balance factor is encountered, the balance factor may be changed to ± 2, and the first node whose balance factor is changed in this way becomes the root *P* of a subtree for which the balance has to be restored. Note that the balance factors do not have to be updated above this node since they remain the same.

In Figure 6.43a, a path is marked with one balance factor equal to +1. Insertion of a new node at the end of this path results in an unbalanced tree (Figure 6.43b), and the balance is restored by one left rotation (Figure 6.43c).

However, if the balance factors on the path from the newly inserted node to the root of the tree are all zero, all of them have to be updated, but no rotation is needed for any of the encountered nodes. In Figure 6.44a, the AVL tree has a path of all zero balance factors. After a node has been appended to the end of this path (Figure 6.44b), no changes are made in the tree except for updating the balance factors of all nodes along this path.

Deletion may be more time-consuming than insertion. First, we apply `deleteByCopying()` to delete a node. This technique allows us to reduce the problem of deleting a node with two descendants to deleting a node with at most one descendant.

FIGURE 6.44 In an AVL tree (a) a new node is inserted (b) requiring no height adjustments.



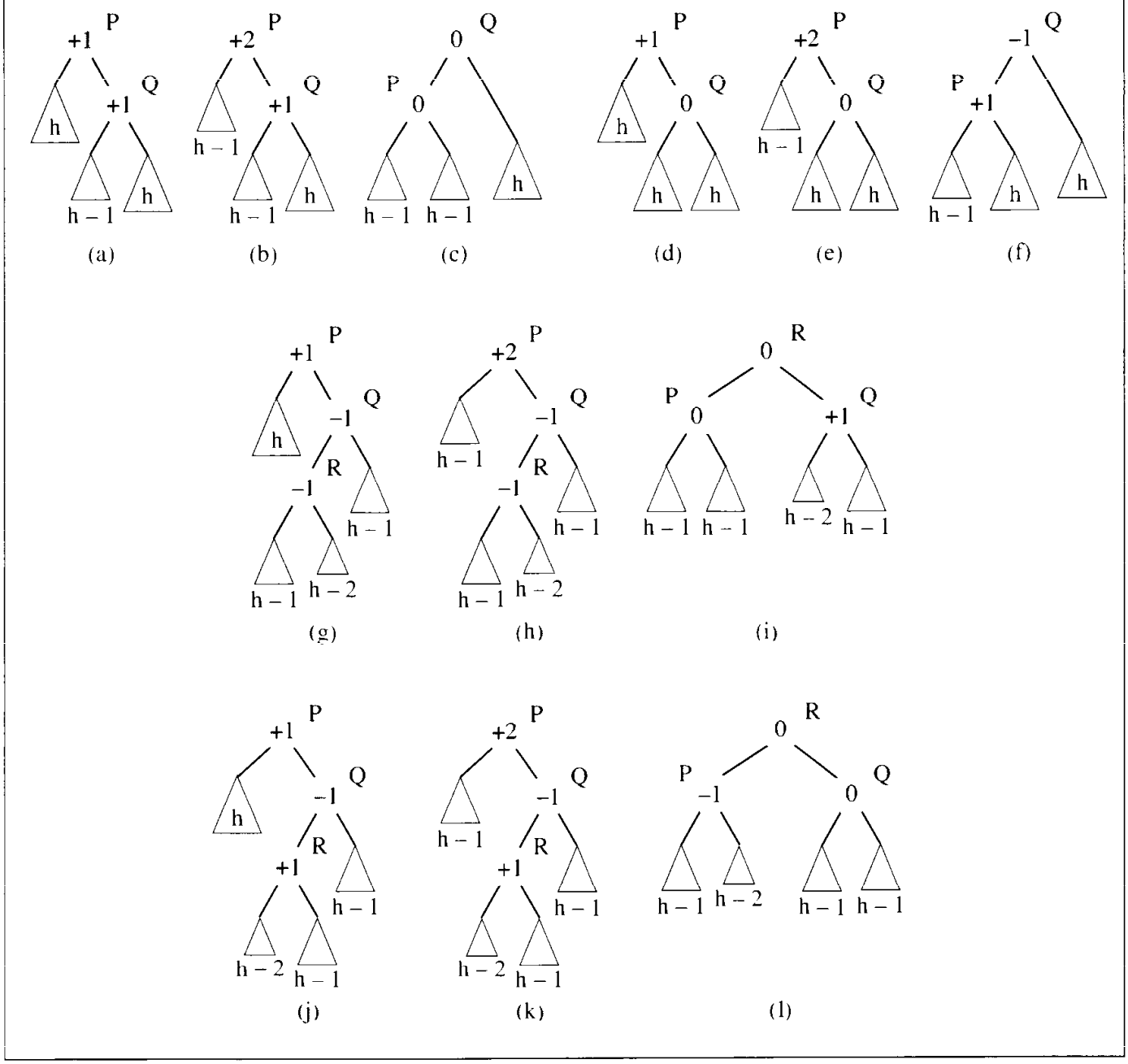
After a node has been deleted from the tree, balance factors are updated from the parent of the deleted node up to the root. For each node in this path whose balance factor becomes ±2, a single or double rotation has to be performed to restore the balance of the tree. Importantly, the rebalancing does not stop after the first node P is found for which the balance factor would become ±2, as is the case with insertion. This also means that deletion leads to at most $O(\lg n)$ rotations, since in the worst case, every node on the path from the deleted node to the root may require rebalancing.

Deletion of a node does not have to necessitate an immediate rotation because it may improve the balance factor of its parent (by changing it from ±1 to 0), but it may also worsen the balance factor for the grandparent (by changing it from ±1 to ±2). We illustrate only those cases that require immediate rotation. There are four such cases (plus four symmetric cases). In each of these cases, we assume that the left child of node P is deleted.

In the first case, the tree in Figure 6.45a turns after deletion into the tree in Figure 6.45b. The tree is rebalanced by rotating Q about P (Figure 6.45c). In the second case, P has a balance factor equal to +1, and its right subtree Q has a balance factor equal to 0 (Figure 6.45d). After deleting a node in the left subtree of P (Figure 6.45e), the tree is rebalanced by the same rotation as in the first case (Figure 6.45f). In this way, cases one and two can be processed together in an implementation after checking that the balance factor of Q is +1 or 0. If Q is −1, we have two other cases, which are more complex. In the third case, the left subtree R of Q has a balance factor equal to −1 (Figure 6.45g). To rebalance the tree, first R is rotated about Q and then about P (Figures 6.45h–i). The fourth case differs from the third in that R's balance factor equals +1 (Figure 6.45j), in which case the same two rotations are needed to restore the balance factor of P (Figures 6.45k–l). Cases three and four can be processed together in a program processing AVL trees.

The previous analyses indicate that insertions and deletions require at most 1.44 $\lg(n + 2)$ searches. Also, insertion can require one single or one double rotation, and

FIGURE 6.45 Rebalancing an AVL tree after deleting a node.



deletion can require 1.44 lg(n + 2) rotations in the worst case. But as also indicated, the average case requires lg(n) + .25 searches, which reduces the number of rotations in case of deletion to this number. To be sure, insertion in the average case may lead to one single/double rotation. Experiments also indicate that deletions in 78% of cases require no rebalancing at all. On the other hand, only 53% of insertions do not bring the tree out of balance (Karlton et al. 1976). Therefore, the more time-consuming deletion occurs less frequently than the insertion operation, not markedly endangering the efficiency of rebalancing AVL trees.

AVL trees can be extended by allowing the difference in height $\Delta > 1$ (Foster 1973). Not unexpectedly, the worst-case height increases with Δ and

$$h = \begin{cases} 1.81 \lg(n) - 0.71 & \text{if } \Delta = 2 \\ 2.15 \lg(n) - 1.13 & \text{if } \Delta = 3 \end{cases}$$

As experiments indicate, the average number of visited nodes increases by one half in comparison to pure AVL trees ($\Delta = 1$), but the amount of restructuring can be decreased by a factor of 10.

6.8 Self-Adjusting Trees

The main concern in balancing trees is to keep them from becoming lopsided and, ideally, to allow leaves to occur only at one or two levels. Therefore, if a newly arriving element endangers the tree balance, the problem is immediately rectified by restructuring the tree locally (the AVL method) or by re-creating the tree (the DSW method). However, we may question whether or not such a restructuring is always necessary. Binary search trees are used to insert, retrieve, and delete elements quickly, and the speed of performing these operations is the issue, not the shape of the tree. Performance can be improved by balancing the tree, but this is not the only method that can be used.

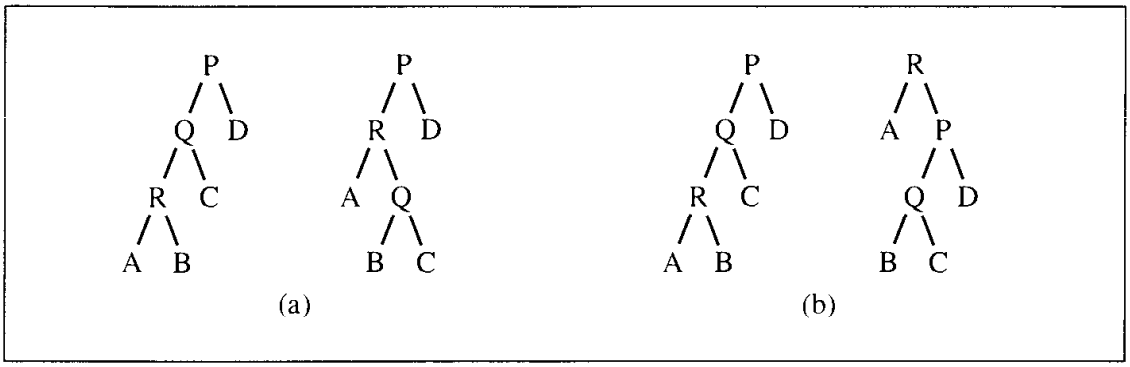
Another approach begins with the observation that not all elements are used with the same frequency. For example, if an element on the tenth level of the tree is used only infrequently, then the execution of the entire program is not greatly impaired by accessing this level. However, if the same element is constantly being accessed, then it makes a big difference whether it is on the tenth level or close to the root. Therefore, the strategy in self-adjusting trees is to restructure trees only by moving up the tree those elements that are used more often, creating a kind of "priority tree." The frequency of accessing nodes can be determined in a variety of ways. Each node can have a counter field which records the number of times the element has been used for any operation. Then the tree can be scanned to move the most frequently accessed elements toward the root. In a less sophisticated approach, it is assumed that an element being accessed has a good chance of being accessed again soon. Therefore, it is moved up the tree. No restructuring is performed for new elements. This assumption may lead to promoting elements which are occasionally accessed, but the overall tendency is to move up elements with a high frequency of access, and for the most part, these elements will populate the first few levels of the tree.

6.8.1 Self-Restructuring Trees

A strategy proposed by Brian Allen and Ian Munro and by James Bitner consists of two possibilities:

1. Single rotation: Rotate a child about its parent if an element in a child is accessed unless it is the root (Figure 6.46a).
2. Moving to the root: Repeat the child-parent rotation until the element being accessed is in the root (Figure 6.46b).

FIGURE 6.46 Restructuring a tree by using (a) a single rotation or (b) moving to the root when accessing node *R*.



Using the single rotation strategy, frequently accessed elements are eventually moved up close to the root so that later accesses are faster than previous ones. In the move-to-the-root strategy, it is assumed that the element being accessed has a high probability to be accessed again, so it percolates right away up to the root. Even if it is not used in the next access, the element remains close to the root. These strategies, however, do not work very well in unfavorable situations, when the binary tree is elongated as in Figure 6.47. In this case, the shape of the tree improves slowly. Nevertheless, it has been determined that the cost of moving a node to the root converges to the cost of accessing the node in optimal tree times 2 ln 2; that is, it converges to (2 ln 2)lg n. The result holds for any probability distribution (that is, independently of the probability that a particular request is issued). However, the average search time when all requests are equally likely is for the single rotation technique equal to $\sqrt{\pi n}$.

6.8.2 Splaying

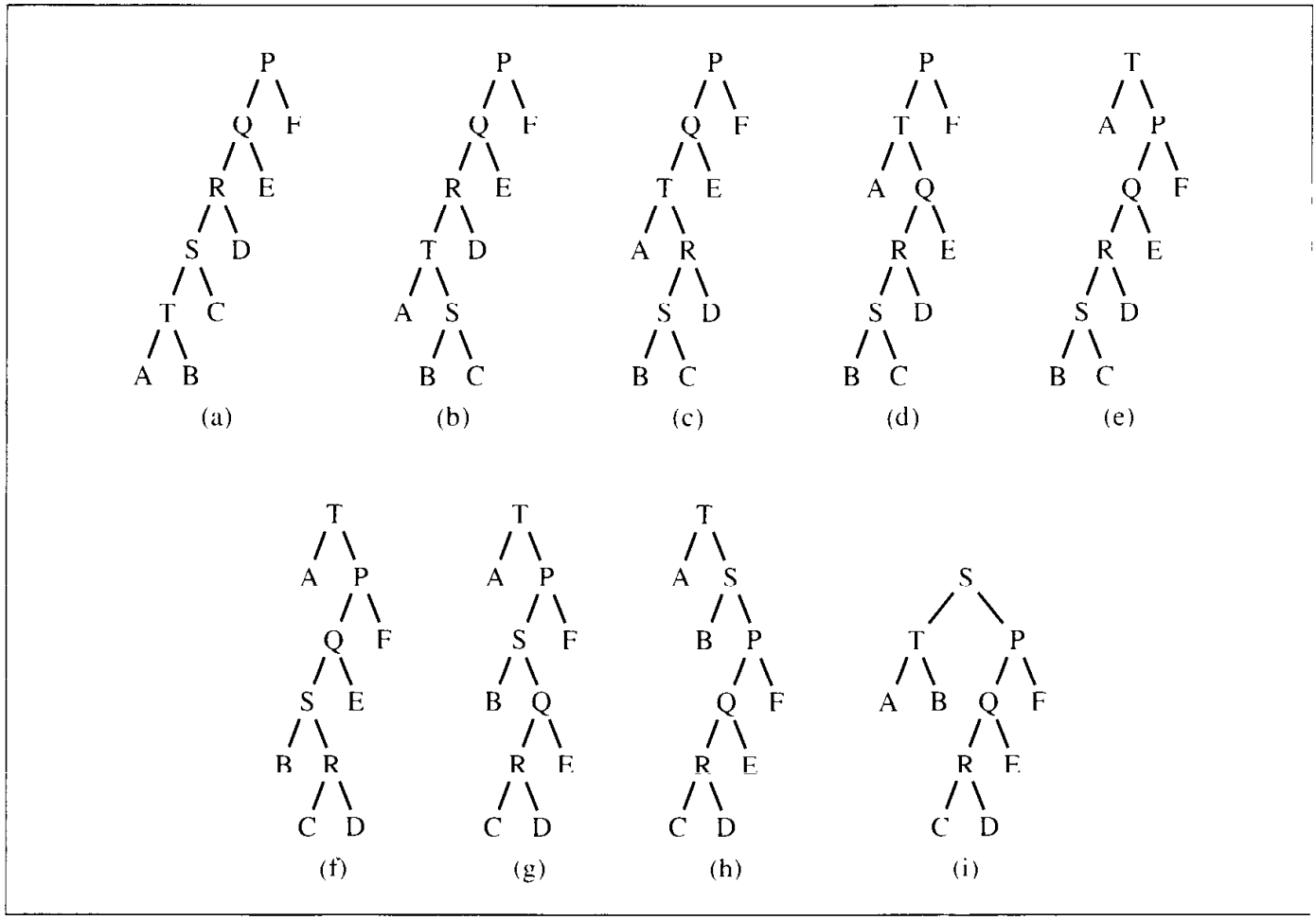
A modification of the move-to-the-root strategy is called *splaying*, which applies single rotations in pairs in an order depending on the links between the child, parent, and grandparent (Sleator and Tarjan 1985). First, three cases are distinguished depending on the relationship between a node R being accessed and its parent Q and grandparent P (if any) nodes:

Case 1: Node R's parent is the root.

Case 2: Homogeneous configuration: Node R is the left child of its parent Q, and Q is the left child of its parent P, or R and Q are both right children.

Case 3: Heterogeneous configuration: Node R is the right child of its parent Q, and Q is the left child of its parent P, or R is the left child of Q, and Q is the right child of P.

FIGURE 6.47 (a–e) Moving element *T* to the root and then (e–i) moving element *S* to the root.



The algorithm to move a node *R* being accessed to the root of the tree is as follows:

```
splaying(P,Q,R)
    while R is not the root
      if R's parent is the root
            perform a singular splay, rotate R about its parent (Figure 6.48a);
      else if R is in homogeneous configuration with its predecessors
           perform a homogeneous splay, first rotate Q about P
           and then R about Q (Figure 6.48b);
      else // if R is in heterogeneous configuration
           // with its predecessors
           perform a heterogeneous splay, first rotate R about Q
           and then about P (Figure 6.48c);
```

The difference in restructuring a tree is illustrated in Figure 6.49, where the tree from Figure 6.47a is used to access node *T* located at the fifth level. The shape of the tree is immediately improved. Then, node *R* is accessed (Figure 6.49c) and the shape of the tree becomes even better (Figure 6.49d).

FIGURE 6.48 Examples of splaying.

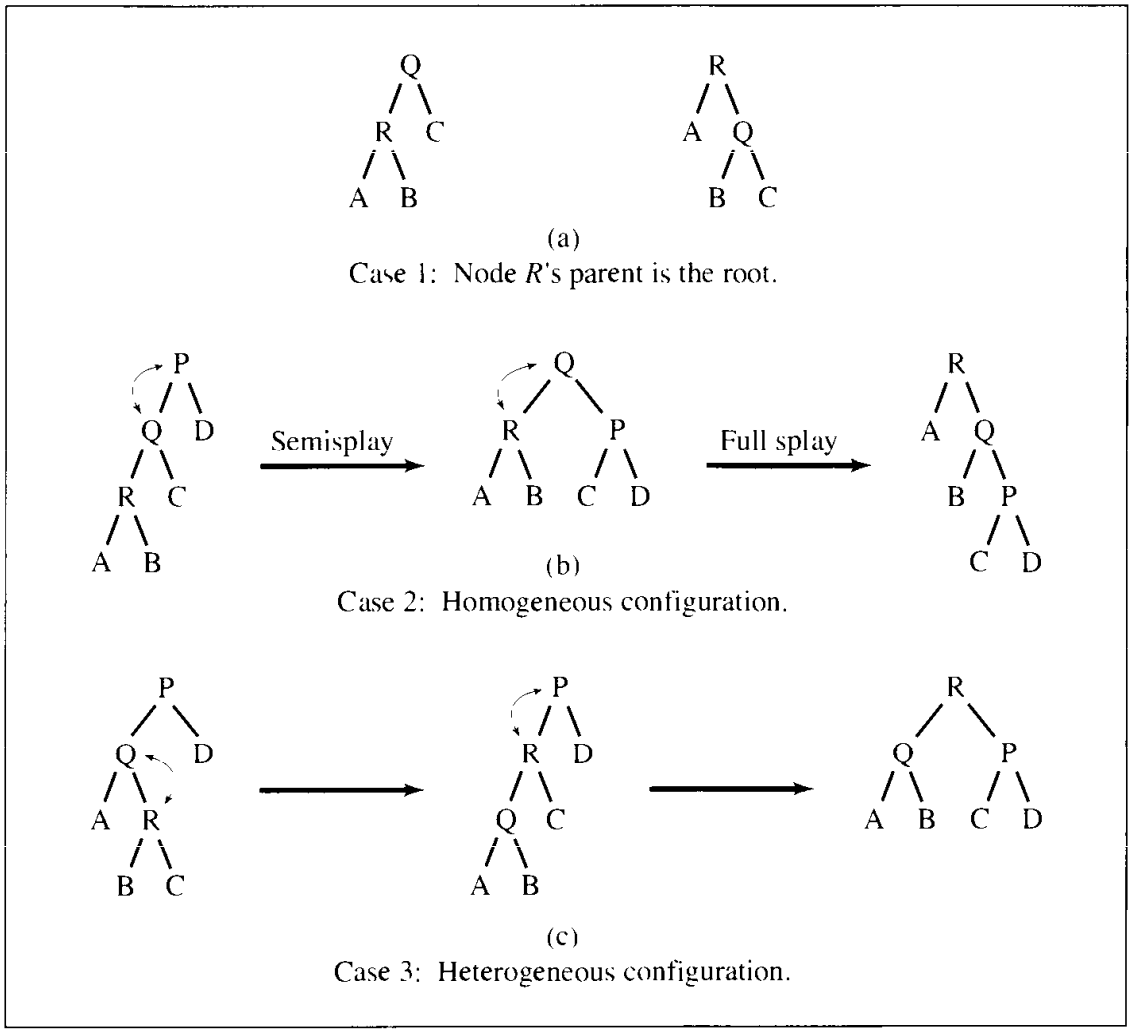
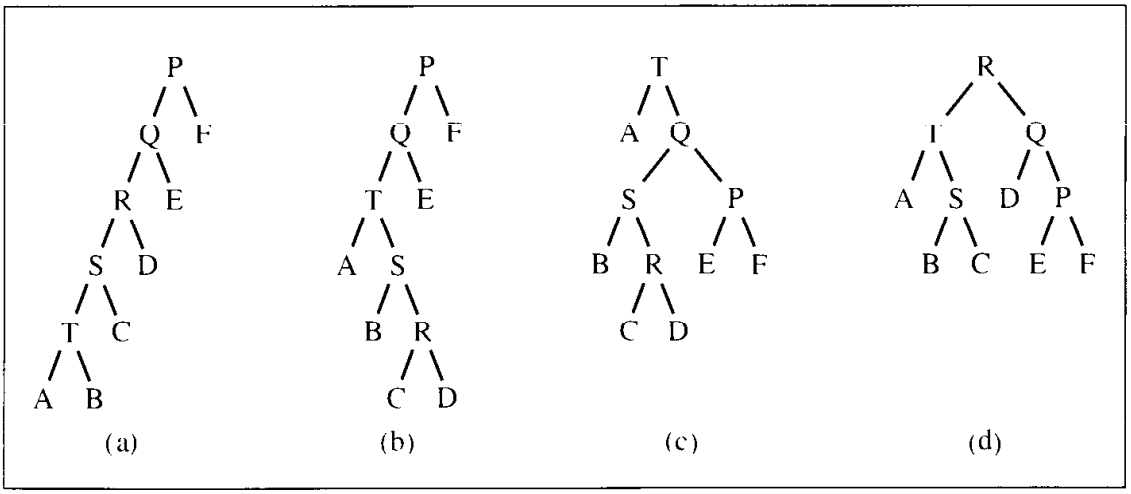


FIGURE 6.49 Restructuring a tree with splaying (a–c) after accessing T and (c–d) then R.



Although splaying is a combination of two rotations except when next to the root, these rotations are not always used in the bottom-up fashion, as in self-adjusting trees. For the homogeneous case (left-left or right-right), first the parent and the grandparent of the node being accessed are rotated, and only afterward are the node and its parent rotated. This has the effect of moving an element to the root and flattening the tree, which has a positive impact on the accesses to be made.

The number of rotations may seem excessive, and it certainly would be if an accessed element happened to be in a leaf every time. In the case of a leaf, the access time is usually $O(\lg n)$, except for some initial accesses when the tree is not balanced. But accessing elements close to the root may make the tree unbalanced. For example, in the tree in Figure 6.49a, if the left child of the root is always accessed, then eventually, the tree would also be elongated, this time extending to the right.

To establish the efficiency of accessing a node in a binary search tree that utilizes the splaying technique, an amortized analysis will be used.

Consider a binary search tree t. Let $nodes(x)$ be the number of nodes in the subtree whose root is x, $rank(x) = \lg(nodes(x))$, so that $rank(root(t)) = \lg(n)$, and $potential(t) = \sum_{x \text{ is a node of } t} rank(x)$. It is clear that $nodes(x) + 1 \leq nodes(parent(x))$; therefore, $rank(x) < rank(parent(x))$. Let the amortized cost of accessing node x be defined as the function

$$amCost(x) = cost(x) + potential_s(t) - potential_0(t)$$

where $potential_s(t)$ and $potential_0(t)$ are the potentials of the tree before access takes place and after it is finished. It is very important to see that one rotation changes ranks only of the node x being accessed, its parent, and its grandparent. This is the reason for basing the definition of the amortized cost of accessing node x on the change in the potential of the tree, which amounts to the change of ranks of the nodes involved in splaying operations that promote x to the root. We can state now a lemma specifying the amortized cost of one access.

Access lemma (Sleator and Tarjan 1985). For the amortized time to splay the tree t at a node x,

$$amCost(x) < 3(\lg(n) - rank(x)) + 1$$

The proof of this conjecture is divided into three parts, each dealing with the different case indicated in Figure 6.48. Let $par(x)$ be a parent of x and $gpar(x)$ a grandparent of x (in Figure 6.48, $x = R$, $par(x) = Q$, and $gpar(x) = P$).

Case 1: One rotation is performed. This can be only the last splaying step in the sequence of such steps which move node x to the root of the tree t, and if there are a total of s splaying steps in the sequence, then the amortized cost of the last splaying step s is

$$amCost_s(x) = cost_s(x) + potential_s(t) - potential_{s-1}(t)$$

$$= 1 + (rank_s(x) - rank_{s-1}(x)) + (rank_s(par(x)) - rank_{s-1}(par(x)))$$

where $cost_s(x) = 1$ represents the actual cost, the cost of the one splaying step (which in this step is limited to one rotation); $potential_{s-1}(t) = rank_{s-1}(x) + rank_{s-1}(par(x)) +$

C and $potential_s(t) = rank_s(x) + rank_s(par(x)) + C$, because x and $par(x)$ are the only nodes whose ranks are modified. Now because $rank_s(x) = rank_{s-1}(par(x))$

$$amCost_s(x) = 1 - rank_{s-1}(x) + rank_s(par(x))$$

and because $rank_s(par(x)) < rank_s(x)$

$$amCost_s(x) < 1 - rank_{s-1}(x) + rank_s(x).$$

Case 2: Two rotations are performed during a homogeneous splay. As before, number 1 represents the actual cost of one splaying step.

$$amCost_i(x) = 1 + (rank_i(x) - rank_{i-1}(x)) + (rank_i(par(x)) - rank_{i-1}(par(x))) + (rank_i(gpar(x)) - rank_{i-1}(gpar(x)))$$

Because $rank_i(x) = rank_{i-1}(gpar(x))$

$$amCost_i(x) = 1 - rank_{i-1}(x) + rank_i(par(x)) - rank_{i-1}(par(x)) + rank_i(gpar(x))$$

Because $rank_i(gpar(x)) < rank_i(par(x)) < rank_i(x)$

$$amCost_i(x) < 1 - rank_{i-1}(x) - rank_{i-1}(par(x)) + 2rank_i(x)$$

and because $rank_{i-1}(x) < rank_{i-1}(par(x))$, that is, $-rank_{i-1}(par(x)) < -rank_{i-1}(x)$

$$amCost_i(x) < 1 - 2rank_{i-1}(x) + 2rank_i(x).$$

To eliminate number 1, consider the inequality $rank_{i-1}(x) < rank_{i-1}(gpar(x))$; that is, $1 \leq rank_{i-1}(gpar(x)) - rank_{i-1}(x)$. From this, we obtain

$$amCost_i(x) < rank_{i-1}(gpar(x)) - rank_{i-1}(x) - 2rank_{i-1}(x) + 2rank_i(x)$$

$$amCost_i(x) < rank_{i-1}(gpar(x)) - 3rank_{i-1}(x) + 2rank_i(x)$$

and because $rank_i(x) = rank_{i-1}(gpar(x))$

$$amCost_i(x) < -3rank_{i-1}(x) + 3rank_i(x)$$

Case 3: Two rotations are performed during a heterogeneous splay. The only difference in this proof is making the assumption that $rank_i(gpar(x)) < rank_i(x)$ and $rank_i(par(x)) < rank_i(x)$ instead of $rank_i(gpar(x)) < rank_i(par(x)) < rank_i(x)$, which renders the same result.

The total amortized cost of accessing a node x equals the sum of amortized costs of all the spaying steps executed during this access. If the number of steps equals s, then at most one (the last) step requires only one rotation (case 1) and thus

$$amCost(x) = \sum_{i=1}^{s} amCost_i(x) = \sum_{i=1}^{s-1} amCost_i(x) + amCost_s(x)$$

$$< \sum_{i=1}^{s-1} 3(rank_i(x) - rank_{i-1}(x)) + rank_s(x) - rank_{s-1}(x) + 1$$

Because $rank_s(x) > rank_{s-1}(x)$,

$$amCost(x) < \sum_{i=1}^{s-1} 3(rank_i(x) - rank_{i-1}(x)) + 3(rank_s(x) - rank_{s-1}(x)) + 1$$

$$= 3(rank_s(x) - rank_0(x)) + 1 = 3(\lg n - rank_0(x)) + 1 = O(\lg n)$$

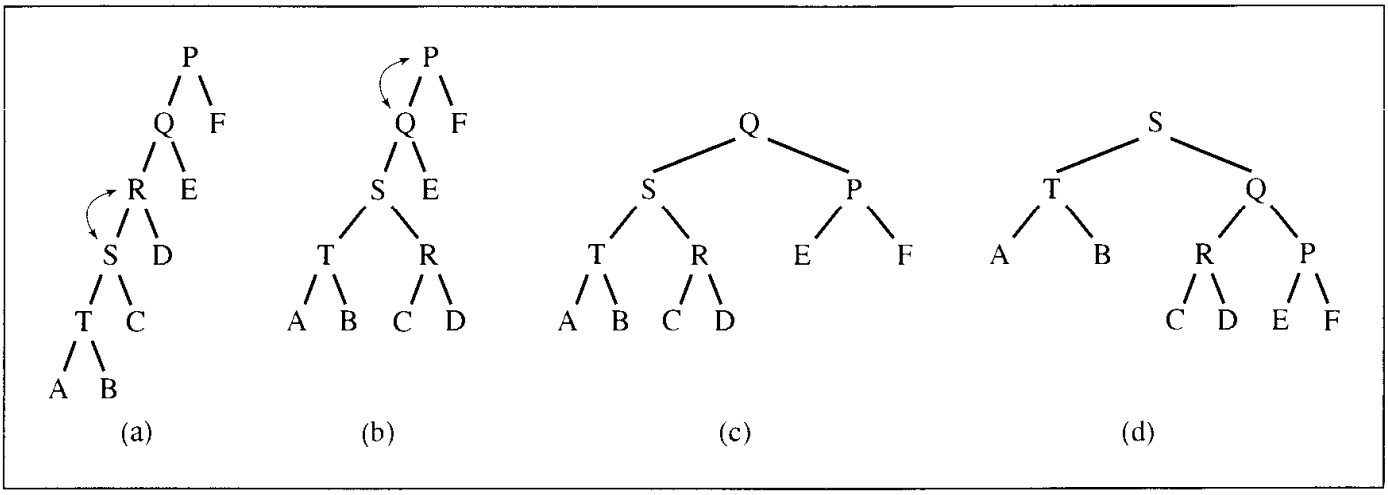
This indicates that the amortized cost of an access to a node in a tree that is restructured with the splaying technique equals $O(\lg n)$, which is the same as the worst case in balanced trees. However, to make the comparison more adequate, we should compare a sequence of m accesses to nodes rather than one access because, with the amortize cost, one isolated access can still be on the order of $O(n)$. The efficiency of a tree that applies splaying is thus comparable to that of a balanced tree for a sequence of accesses and equals $O(m \lg n)$.

Splaying is a strategy focusing upon the elements rather than the shape of the tree. It may perform well in situations in which some elements are used much more frequently than others. If elements near the root are accessed with about the same frequency as elements on the lowest levels, then splaying may not be the best choice. In this case, a strategy which stresses balancing the tree rather than frequency is better; a modification of the splaying method is a more viable option.

Semisplaying is a modification that requires only one rotation for a homogeneous splay and continues splaying with the parent of the accessed node, not with the node itself. It is illustrated in Figure 6.48b. After R is accessed, its parent Q is rotated about P and splaying continues with Q, not with R. Rotation of R about Q is not performed, as would be the case for splaying.

Figure 6.50 illustrates the advantages of semisplaying. The elongated tree from Figure 6.49a becomes more balanced with semisplaying after accessing T (Figures 6.50a–c), and after T is accessed again, the tree in Figure 6.50d has basically the same number of levels as the tree in Figure 6.46a. (It may have one more level if E or F was a subtree higher than any of subtrees A, B, C, or D.) For implementation of this tree strategy, see the case study at the end of this chapter.

FIGURE **6.50** (a–c) Accessing T and restructuring the tree with semisplaying; (c–d) accessing T again.



6.9 HEAPS

A particular kind of binary tree, called a *heap*, has the following two properties:

1. The value of each node is not less than the values stored in each of its children.
2. The tree is perfectly balanced, and the leaves in the last level are all in the leftmost positions.

To be exact, these two properties define a *max heap*. If "less" in the first property is replaced with "greater," then the definition specifies a *min heap*. This means that the root of a max heap contains the largest element, whereas the root of a min heap contains the smallest. A tree has the *heap property* if each nonleaf has the first property. Due to the second condition, the number of levels in the tree is $O(\lg n)$.

The trees in Figure 6.51a are all heaps; the trees in Figure 6.51b violate the first property, and the trees in Figure 6.51c violate the second.

Interestingly, heaps can be implemented by arrays. For example, the array `data` = [2 8 6 1 10 15 3 12 11] can represent a nonheap tree in Figure 6.52. The elements are placed at sequential locations representing the nodes from top to bottom and in each level from left to right. The second property reflects the fact that the array is packed, with no gaps. Now, a heap can be defined as an array `heap` of length n in which

$$\texttt{heap[i]} \geq \texttt{heap[2 * i + 1]}, \text{for } 0 \leq i < \frac{n-1}{2}$$

FIGURE 6.51 Examples of (a) heaps and (b–c) nonheaps.

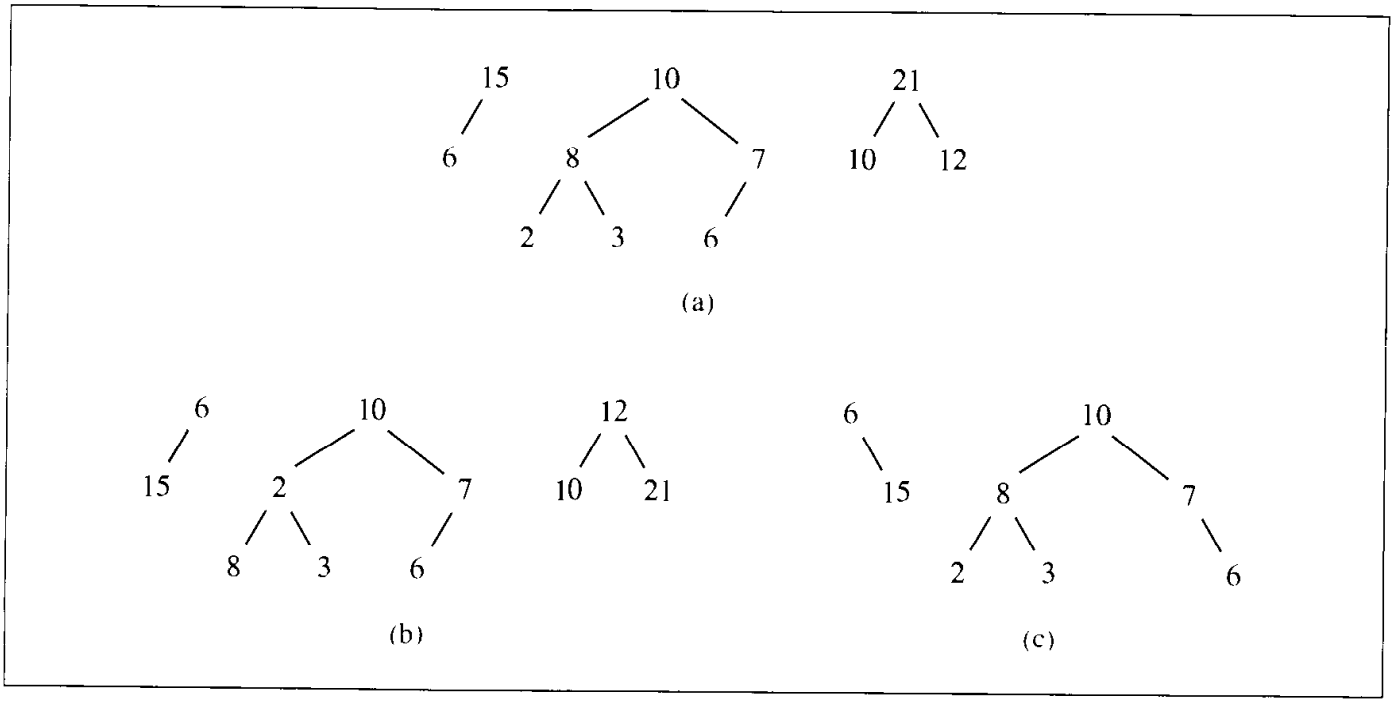


FIGURE 6.52 The array [2 8 6 1 10 15 3 12 11] seen as a tree.

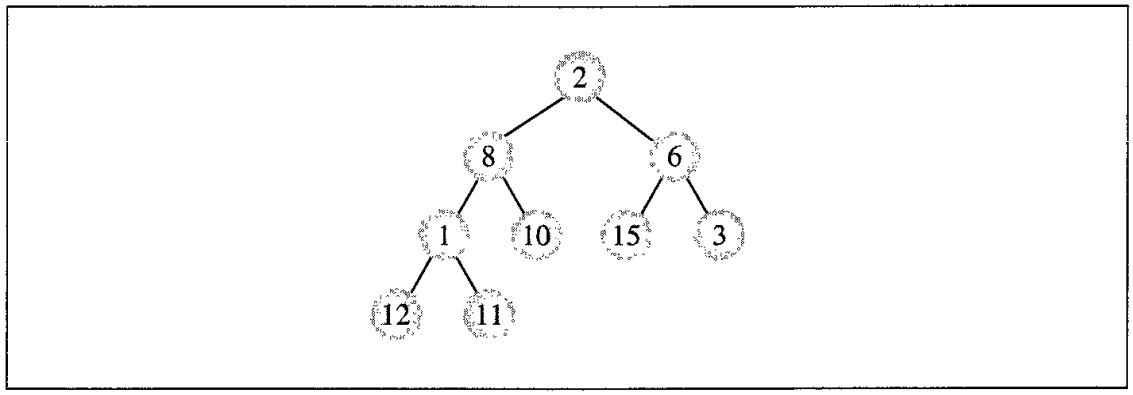
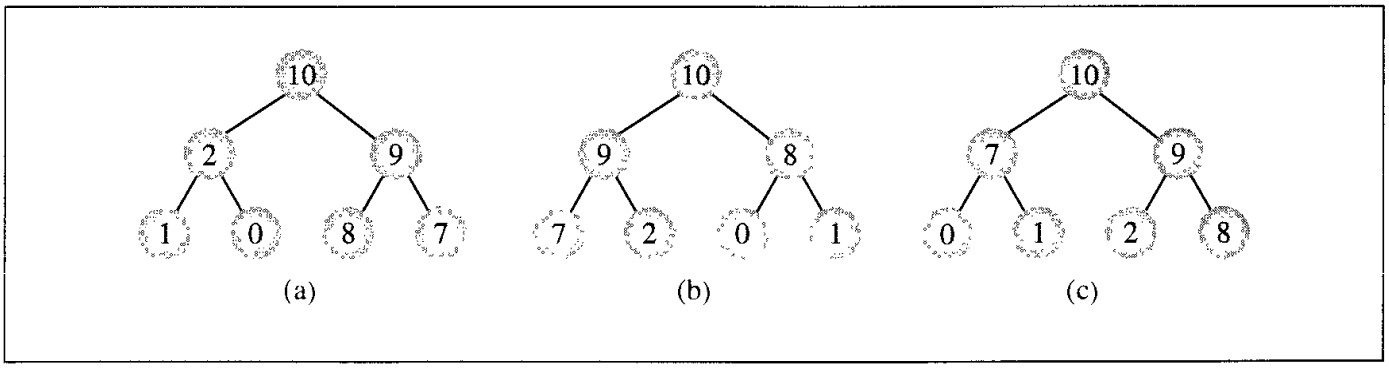


FIGURE 6.53 Different heaps constructed with the same elements.



and

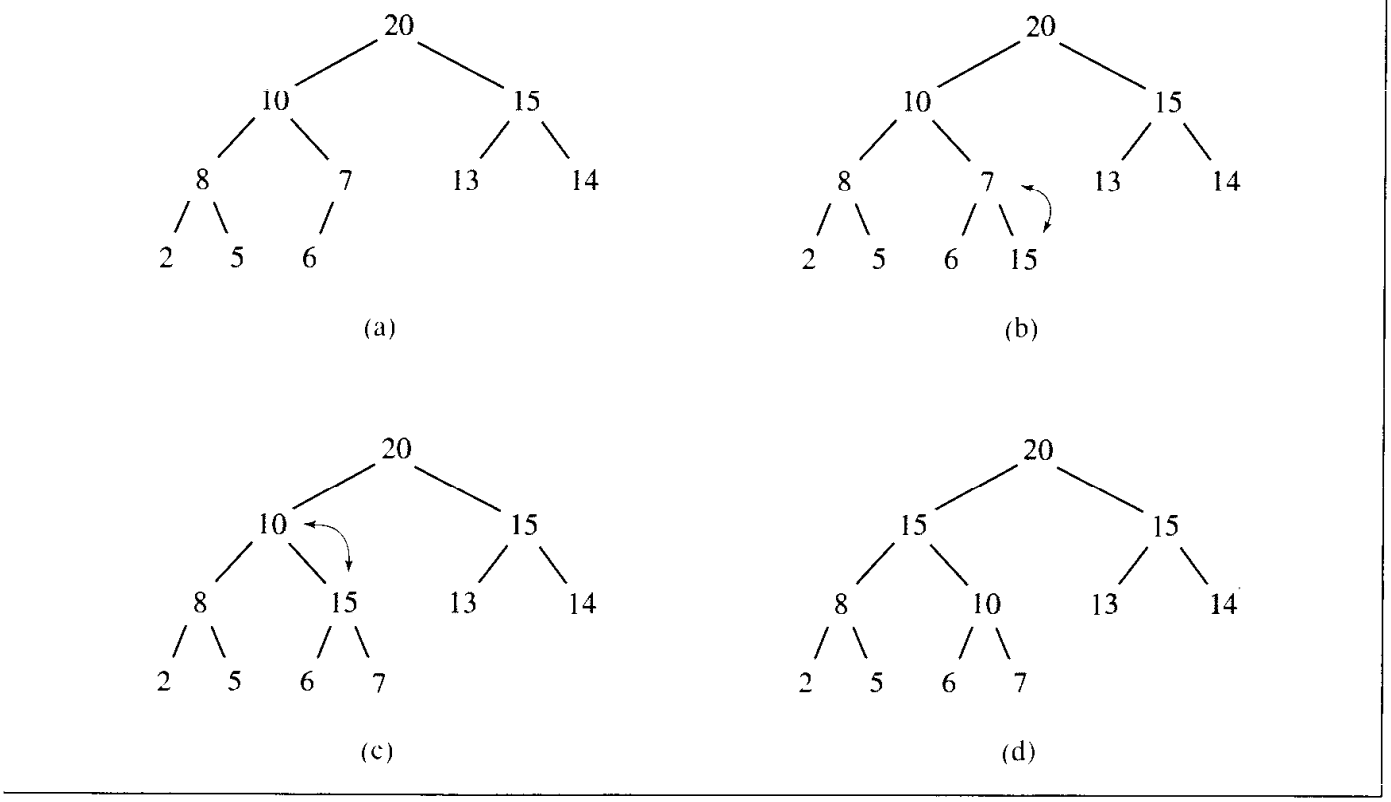
$$\texttt{heap[i]} \geq \texttt{heap[2 * i + 2]}, \text{for } 0 \leq i < \frac{n-2}{2}$$

Elements in a heap are not perfectly ordered. We know only that the largest element is in the root node and that, for each node, all its descendants are less than or equal to that node. But the relation between sibling nodes or, to continue the kinship terminology, between uncle and nephew nodes is not determined. The order of the elements obeys a linear line of descent, disregarding lateral lines. For this reason, all the trees in Figure 6.53 are legitimate heaps, although the heap in Figure 6.53b is ordered best.

6.9.1 Heaps as Priority Queues

A heap is an excellent way to implement a priority queue. Section 4.3 used linked lists to implement priority queues, structures for which the complexity was expressed in terms of $O(n)$ or $O(\sqrt{n})$. For large n, this may be too ineffective. On the other hand, a heap is a perfectly balanced tree; hence, reaching a leaf requires $O(\lg n)$ searches. This efficiency is very promising. Therefore, heaps can be used to implement priority

FIGURE 6.54 Enqueuing an element to a heap.



queues. To this end, however, two procedures have to be implemented to enqueue and dequeue elements on a priority queue.

To enqueue an element, the element is added at the end of the heap as the last leaf. Restoring the heap property in the case of enqueuing is achieved by moving from the last leaf toward the root.

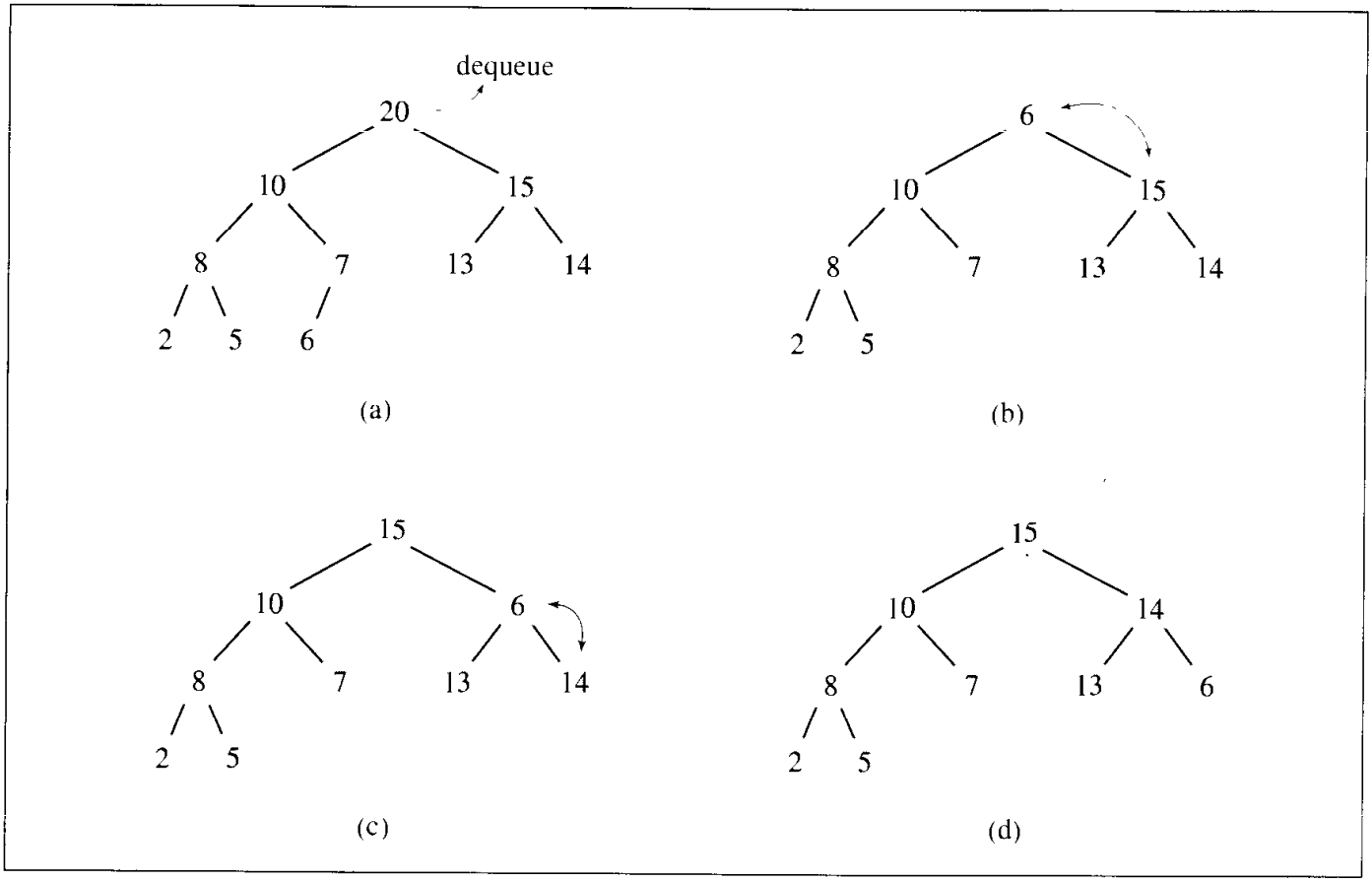
The algorithm for enqueuing is as follows:

```
heapEnqueue(el)
  put el at the end of heap;
  while el is not in the root and el > parent(el)
    swap el with its parent;
```

For example, the number 15 is added to the heap in Figure 6.54a as the next leaf (Figure 6.54b), which destroys the heap property of the tree. To restore this property, 15 has to be moved up the tree until either it ends up in the root or finds a parent that is not less than 15. In this example, the latter case occurs, and 15 has to be moved only twice without reaching the root.

Dequeuing an element from the heap consists of removing the root element from the heap, since by the heap property it is the element with the greatest priority. Then the last leaf is put in its place, and the heap property almost certainly has to be restored, this time by moving from the root down the tree.

FIGURE 6.55 Dequeuing an element from a heap.



The algorithm for dequeuing is as follows:

```
heapDequeue( )
   extract the element from the root;
   put the element from the last leaf in its place;
   remove the last leaf;
   // both subtrees of the root are heaps;
   p = the root;
   while p is not a leaf and p < any of its children
      swap p with the larger child;
```

For example, 20 is dequeued from the heap in Figure 6.55a and 6 is put in its place (Figure 6.55b). To restore the heap property, 6 is swapped first with its larger child, number 15 (Figure 6.55c), and once again with the larger child, 14 (Figure 6.55d).

The last three lines of the dequeuing algorithm can be treated as a separate algorithm that restores the heap property only if it has been violated by the root of the tree. In this case, the root element is moved down the tree until it finds a proper position. This algorithm, which is the key to the heap sort is presented in one possible implementation in Figure 6.56.

FIGURE 6.56 Implementation of algorithm to move the root element down a tree.

```
template<class T>
void moveDown (T data[], int first, int last) {
    int largest = 2*first + 1;
    while (largest <= last) {
        if (largest < last && // first has two children (at 2*first+1 and
            data[largest] < data[largest+1]) // 2*first+2) and the second
             largest++;                      // is larger than the first;

        if (data[first] < data[largest]) {    // if necessary,
             swap(data[first],data[largest]); // swap child and parent,
             first = largest;                 // and move down;
             largest = 2*first+1;
        }
        else largest = last+1;  // to exit the loop: the heap property
    }                           // isn't violated by data[first];
}
```

6.9.2 Organizing Arrays as Heaps

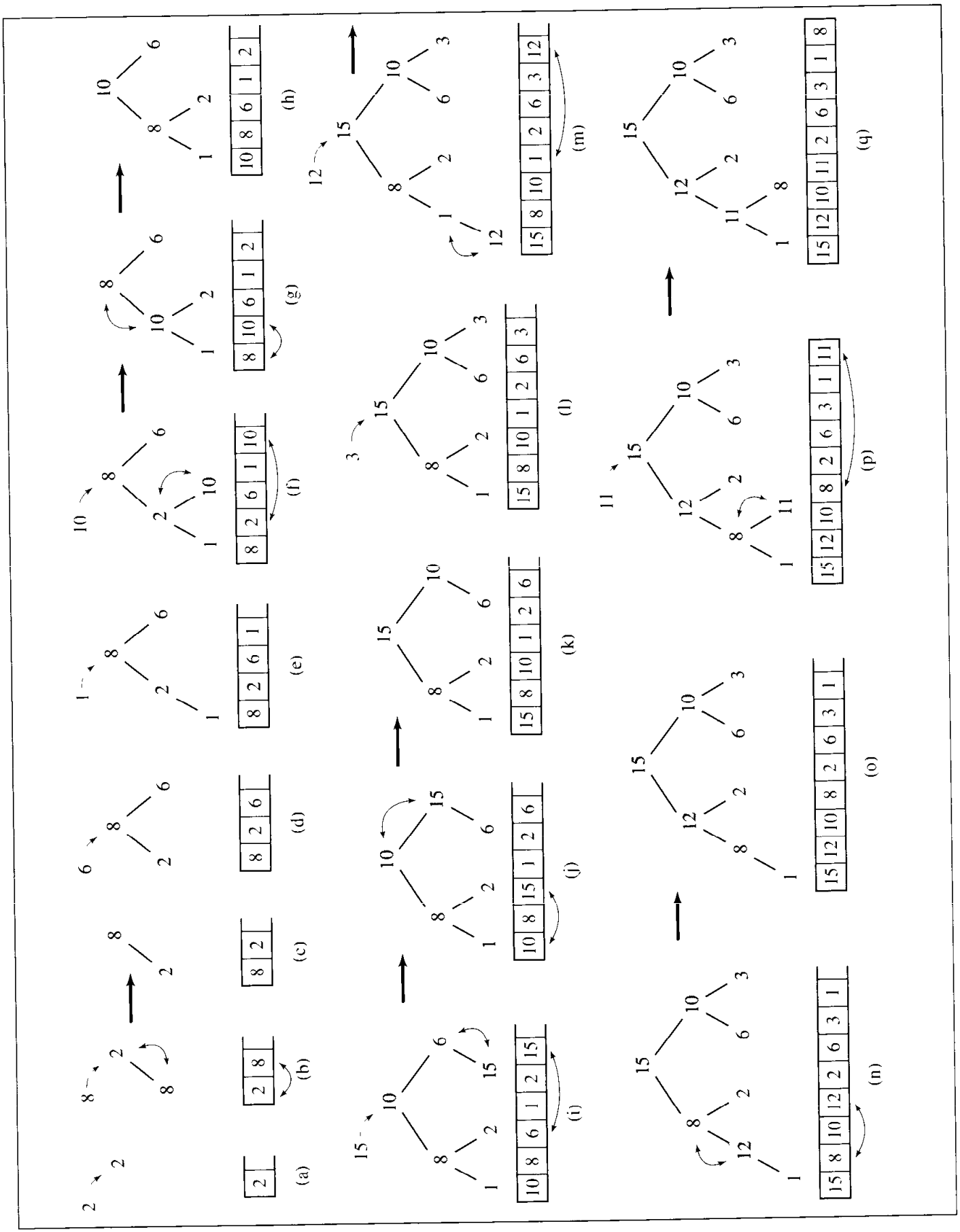
Heaps can be implemented as arrays, and in that sense, each heap is an array, but all arrays are not heaps. In some situations, however, most notably in heap sort (cf. Section 9.3.2), we need to convert an array into a heap (that is, reorganize the data in the array so that the resulting organization represents a heap). There are several ways to do this, but in light of the preceding section the simplest way is to start with an empty heap and sequentially include elements into a growing heap. This is a top-down method and it was proposed by John Williams; it extends the heap by enqueuing new elements in the heap.

Figure 6.57 contains a complete example of the top-down method. First, the number 2 is enqueued in the initially empty heap (6.57a). Next, 8 is enqueued by putting it at the end of the current heap (6.57b) and then swapping with its parent (6.57c). Enqueuing the third and fourth elements, 6 (6.57d) and then 1 (6.57e), necessitates no swaps. Enqueuing the fifth element, 10, amounts to putting it at the end of the heap (6.57f), then swapping it with its parent, 2 (6.57g), and then with its new parent, 8 (6.57h) so that eventually 10 percolates up to the root of the heap. All remaining steps can be traced in Figure 6.57.

To check the complexity of the algorithm, observe that in the worst case, when a newly added element has to be moved up to the root of the tree, $\lfloor \lg k \rfloor$ exchanges are made in a heap of k nodes. Therefore, if n elements are enqueued, then in the worst case

$$\sum_{k=1}^{n} \lfloor \lg k \rfloor \le \sum_{k=1}^{n} \lg k = \lg 1 + \cdots + \lg n = \lg(1 \cdot 2 \cdot \cdots \cdot n) = \lg(n!) = O(n \lg n)$$

FIGURE 6.57 Organizing an array as a heap with a top-down method.



exchanges are made during execution of the algorithm and the same number of comparisons. (For the last equality, $\lg(n!) = O(n \lg n)$, see Section A.2 in Appendix A.) It turns out, however, that we can do better than that.

In another algorithm, developed by Robert Floyd, a heap is built bottom-up. In this approach, small heaps are formed and repetitively merged into larger heaps in the following way:

```
FloydAlgorithm(data[])
    for (i = index of the last nonleaf; i >= 0; i--)
        restore the heap property for the tree whose root is data[i] by calling
        moveDown(data,i,n-1);
```

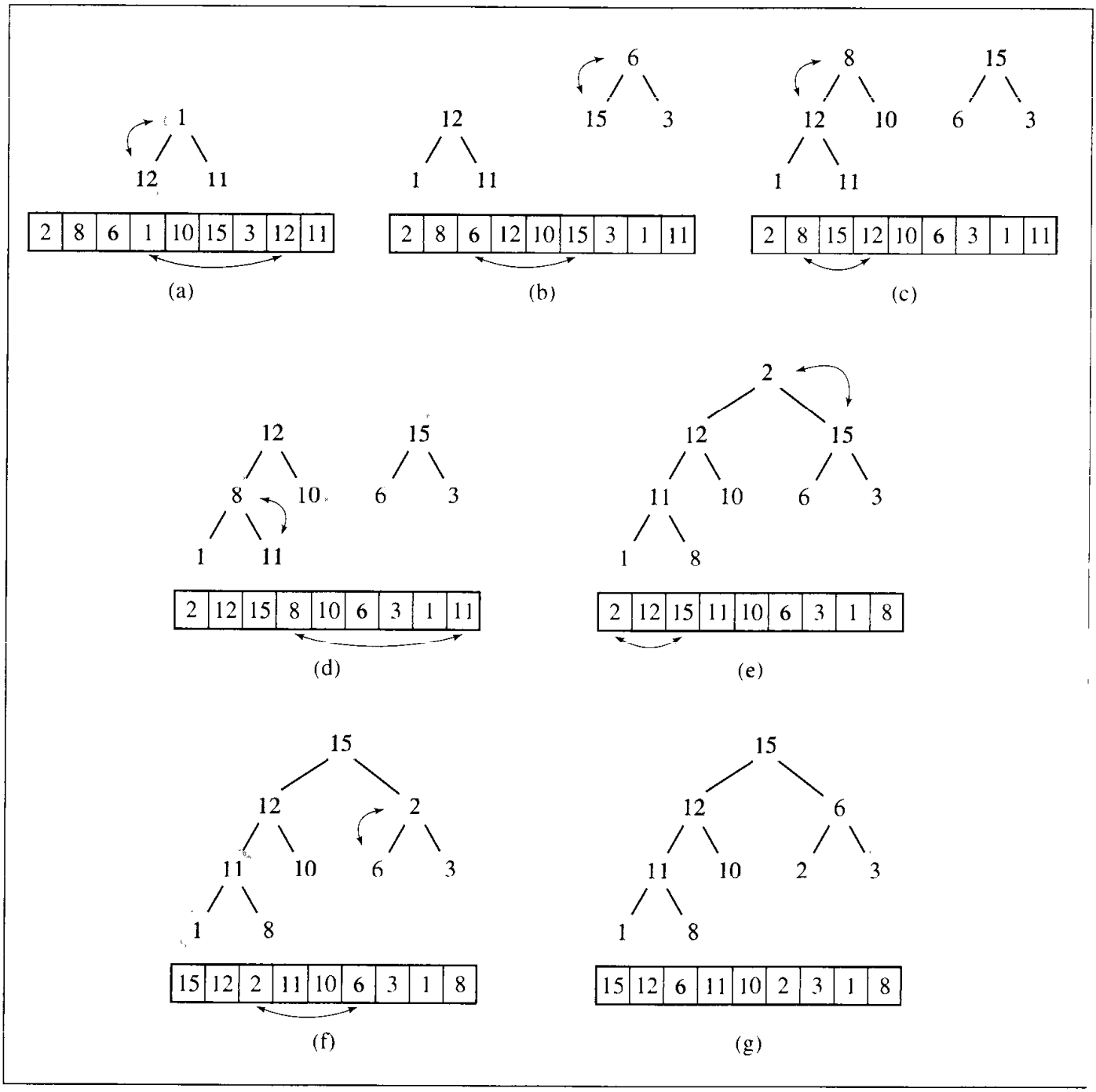
Figure 6.58 contains an example of transforming the array `data[]` = [2 8 6 1 10 15 3 12 11] into a heap.

We start from the last nonleaf node, which is `data[n/2-1]`, n being the array size. If `data[n/2-1]` is less than one of its children, it is swapped with the larger child. In the tree in Figure 6.58a, this is the case for `data[3]` = 1 and `data[7]` = 12. After exchanging the elements, a new tree is created, shown in Figure 6.58b. Next the element `data[n/2-2]` = `data[2]` = 6 is considered. Because it is smaller than its child `data[5]` = 15, it is swapped with that child and the tree is transformed to that in Figure 6.58c. Now `data[n/2-3]` = `data[1]` = 8 is considered. Since it is smaller than one of its children, which is `data[3]` = 12, an interchange occurs, leading to the tree in Figure 6.58d. But now it can be noticed that the order established in the subtree whose root was 12 (Figure 6.58c) has been somewhat disturbed since 8 is smaller than its new child 11. This simply means that it does not suffice to compare a node's value with its children's, but a similar comparison needs to be done with grandchildren's, great-grandchildren's, etc. until the node finds its proper position. Taking this into consideration, the next swap is made, after which the tree in Figure 6.58e is created. Only now is the element `data[n/2-4]` = `data[0]` = 2 compared with its children, which leads to two swaps (Figures 6.58f–g).

When a new element is analyzed, both its subtrees are already heaps, as is the case with number 2, and both its subtree with roots in nodes 12 and 15 are already heaps (Figure 6.58e). This observation is generally true: Before one element is considered, its subtrees have already been converted into heaps. Thus, a heap is created from the bottom up. If the heap property is disturbed by an interchange as in the transformation of the tree in Figure 6.58c to that in Figure 6.58d, it is immediately restored by shifting up elements that are larger than the element moved down. This is the case when 2 is exchanged with 15. The new tree is not a heap since node 2 has still larger children (Figure 6.58f). To remedy this problem, 6 is shifted up and 2 is moved down. Figure 6.58g is a heap.

We assume that a complete binary tree is created, that is, $n = 2^k - 1$ for some k. To create the heap, `moveDown()` is called $\frac{n+1}{2}$ times, once for each nonleaf. In the worst case, `moveDown()` moves data from the next to last level, consisting of $\frac{n+1}{4}$ nodes, down by one level to the level of leaves performing $\frac{n+1}{4}$ swaps. Therefore, all nodes from this level make $1 \cdot \frac{n+1}{4}$ moves. Data from the second to last level, which has $\frac{n+1}{8}$ nodes, are moved two levels down to reach the level of the leaves. Thus, nodes from this level perform $2 \cdot \frac{n+1}{8}$ moves and so on up to the root. The root of the tree as the tree becomes a heap is moved, again in the worst case, $\lg(n+1) - 1 = \lg \frac{n+1}{2}$ levels

FIGURE 6.58 Transforming the array [2 8 6 1 10 15 3 12 11] into a heap with a bottom-up method.



down the tree to end up in one of the leaves. Since there is only one root, this contributes $\lg \frac{n+1}{2} \cdot 1$ moves. The total number of movements can be given by this sum

$$\sum_{i=2}^{\lg(n+1)} \frac{n+1}{2^i}(i-1) = (n+1) \sum_{i=2}^{\lg(n+1)} \frac{i-1}{2^i}$$

which is $O(n)$ since the series $\sum_{i=2}^{\infty} \frac{i}{2^i}$ converges to 1.5 and $\sum_{i=2}^{\infty} \frac{i}{2^i}$ converges to 0.5. For an array which is not a complete binary tree, the complexity is all the more bounded

by $O(n)$. The worst case for comparisons is twice this value, which is also $O(n)$, since in `moveDown()`, for each node, both children of the node are compared to each other to choose the larger. That, in turn, is compared to the node. Therefore, for the worst case, Williams's method performs better than Floyd's.

The performance for the average case is much more difficult to establish. It has been found that Floyd's heap construction algorithm requires on average $1.88n$ comparisons (Knuth 1998; Doberkat 1984), and the number of comparisons required by Williams's algorithm in this case is between $1.75n$ and $2.76n$ and the number of swaps is $1.3n$ (Hayward and McDiarmid 1991; McDiarmid and Reed 1989). Thus, in the average case, the two algorithms perform at the same level.

6.10 Polish Notation and Expression Trees

One of the applications of binary trees is an unambiguous representation of arithmetical, relational, or logical expressions. In the early 1920s, a Polish logician, Jan Łukasiewicz (pronounced: wook-a-sie-vich), invented a special notation for propositional logic that allows us to eliminate all parentheses from formulas. However, Łukasiewicz's notation, called *Polish notation,* results in less readable formulas than the parenthesized originals and it was not widely used. However, it proved useful after the emergence of computers, especially for writing compilers and interpreters.

To maintain readability and prevent the ambiguity of formulas, extra symbols such as parentheses have to be used. However, if avoiding ambiguity is the only goal, then these symbols can be omitted at the cost of changing the order of symbols used in the formulas. This is exactly what the compiler does. It rejects everything that is not essential to retrieve the proper meaning of formulas, rejecting it as "syntactic sugar."

How does this notation work? Look first at the following example. What is the value of this algebraic expression?

$$2 - 3 \cdot 4 + 5$$

The result depends on the order in which the operations are performed. If we multiply first and then subtract and add, the result is –5 as expected. If subtraction is done first, then addition and multiplication, as in

$$(2 - 3) \cdot (4 + 5)$$

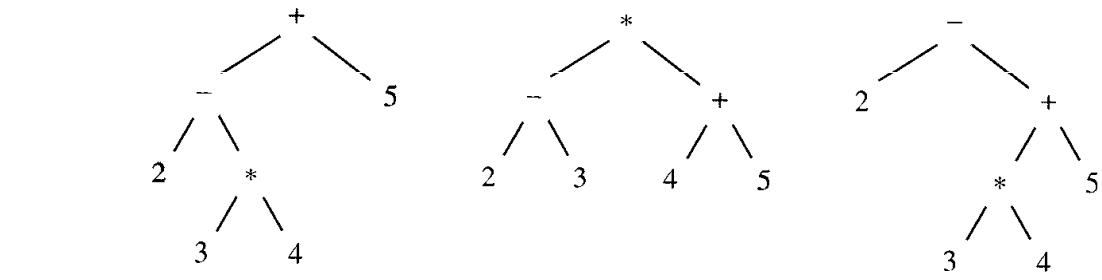
the result is –9. But if we subtract after we multiply and add, as in

$$2 - (3 \cdot 4 + 5)$$

then the result of evaluation is –15. If we see the first expression, then we know in what order to evaluate it. But the computer does not know that, in such a case, multiplication has precedence over addition and subtraction. If we want to override the precedence, then parentheses are needed.

Compilers need to generate assembly code in which one operation is executed at a time and the result is retained for other operations. Therefore, all expressions have

FIGURE 6.59 Examples of three expression trees and results of their traversals.



	(a)	(b)	(c)
	$2 - 3 * 4 + 5$	$(2 - 3) * (4 + 5)$	$2 - (3 * 4 + 5)$
Preorder	$+ - 2 * 3\ 4\ 5$	$* - 2\ 3 + 4\ 5$	$- 2 + * 3\ 4\ 5$
Inorder	$2 - 3 * 4 + 5$	$2 - 3 * 4 + 5$	$2 - 3 * 4 + 5$
Postorder	$2\ 3\ 4 * - 5 +$	$2\ 3 - 4\ 5 + *$	$2\ 3\ 4 * 5 + -$

to be broken down unambiguously into separate operations and put into their proper order. That is where Polish notation is useful. It allows us to create an *expression tree*, which imposes an order on the execution of operations. For example, the first expression, $2 - 3 \cdot 4 + 5$, which is the same as $2 - (3 \cdot 4) + 5$, is represented by the tree in Figure 6.59a. The second and the third expressions correspond to the trees in Figures 6.59b and 6.59c. It is obvious now that in both Figures 6.59a and 6.59c we have to first multiply 3 by 4 to obtain 12. But 12 is subtracted from 2, according to the tree in Figure 6.59a, and added to 5, according to Figure 6.59c. There is no ambiguity involved in this tree representation. The final result can be computed only if intermediate results are calculated first.

Notice also that trees do not use parentheses and yet no ambiguity arises. We can maintain this parentheses-free situation if the expression tree is linearized (that is, if the tree is transformed into an expression using a tree traversal method). The three traversal methods relevant in this context are preorder, inorder, and postorder tree traversals. Using these traversals, nine outputs are generated, as shown in Figure 6.59. Interestingly, inorder traversal of all three trees results in the same output, which is the initial expression that caused all the trouble. What it means is that inorder tree traversal is not suitable for generating unambiguous output. But the other two traversals are. They are different for different trees and are therefore useful for the purpose of creating unambiguous expressions and sentences.

Because of the importance of these different conventions, special terminology is used. Preorder traversal generates *prefix notation*, inorder traversal generates *infix notation*, and postorder traversal generates *postfix notation*. Note that we are accustomed to infix notation. In infix notation, an operator is surrounded by its two operands. In prefix notation, the operator precedes the operands, and in postfix notation the operator follows the operands. Some hand calculators operate on expressions in postfix

notation. Also, some programming languages are using Polish notation. For example, Forth uses postfix notation. LISP and, to a large degree, LOGO use prefix notation.

6.10.1 Operations on Expression Trees

Binary trees can be created in two different ways: top-down or bottom-up. In the implementation of insertion, the first approach was used. This section applies the second approach by creating expression trees bottom-up while scanning infix expressions from left to right.

The most important part of this construction process is retaining the same precedence of operations as in the expression being scanned, as exemplified in Figure 6.59. If parentheses are not allowed, the task is simple, as parentheses allow for many levels of nesting. Therefore, an algorithm should be powerful enough to process any number of nesting levels in an expression. A natural approach is a recursive implementation. We modify the recursive descent interpreter discussed in Chapter 5's case study and outline a recursive descent expression tree constructor.

As Figure 6.59 indicates, a node contains either an operator or an operand, the latter being either an identifier or a number. To simplify the task, all of them can be represented as strings in an instance of the class defined as

```
class ExprTreeNode {
public:
    ExprTreeNode(char *k, ExprTreeNode *l, ExprTreeNode *r){
        key = new char[strlen(k)+1];
        strcpy(key,k);
        left = l; right = r;
    }
    . . . . . .
private:
    char *key;
    ExprTreeNode *left, *right;
}
```

Expressions that are converted to trees use the same syntax as expressions in the case study in Chapter 5. Therefore, the same syntax diagrams can be used. Using these diagrams, a class `ExprTree` can be created in which member functions for processing a factor and term have the following pseudocode (a function for processing an expression has the same structure as the function processing a term):

```
factor()
    if (token is a number, id or operator)
        return new ExprTreeNode(token);
    else if (token is '(')
        ExprTreeNode *p = expr();
        if (token is ')')
            return p;
        else error;
```

```
term()
    ExprTreeNode *p1, *p2;
    p1 = factor();
    while (token is '*' or '/')
        oper = token;
        p2 = factor();
        p1 = new ExprTreeNode(oper,p1,p2);
    return p1;
```

The tree structure of expressions is very suitable for generating assembly code or intermediate code in compilers, as shown in this pseudocode of a function from `ExprTree` class:

```
void generateCode() {
     generateCode(root);
}
generateCode(ExprTreeNode *p) {
    if (p->key is a number or id)
         return p->key;
    else if (p->key is an addition operator)
         result = newTemporaryVar();
         output << "add\t" << generateCode(p->left) << "\t"
                << generateCode(p->right) << "\t"
                <<result<<endl;
         return result;
    . . . . . .
}
```

With these member functions, an expression

$$(\text{var2} + \text{n}) * (\text{var2} + \text{var1})/5$$

is transformed into an expression tree shown in Figure 6.60, and from this tree, `generateCode()` generates the following intermediate code:

```
add     var2        n           _tmp_3
add     var2        var1        _tmp_4
mul     _tmp_3      _tmp_4      _tmp_2
div     _tmp_2      5           _tmp_1
```

Expression trees are also very convenient for performing other symbolic operations, such as differentiation. Rules for differentiation (given in the programming assignments in Chapter 5) are shown in the form of tree transformations in Figure 6.61 and in the following pseudocode:

```
differentiate(p,x) {
    if (p == 0)
        return 0;
    if (p->key is the id x)
        return new ExprTreeNode("1");
```

Figure 6.60 An expression tree.

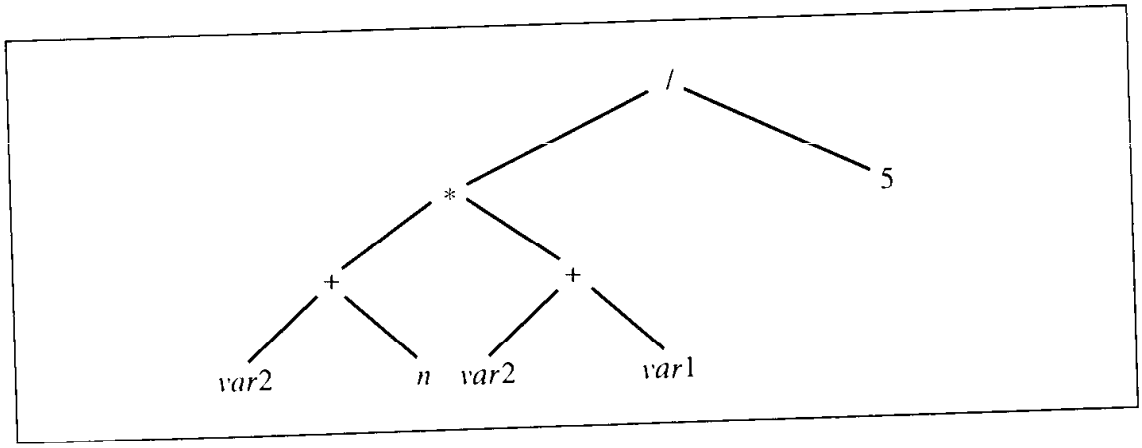
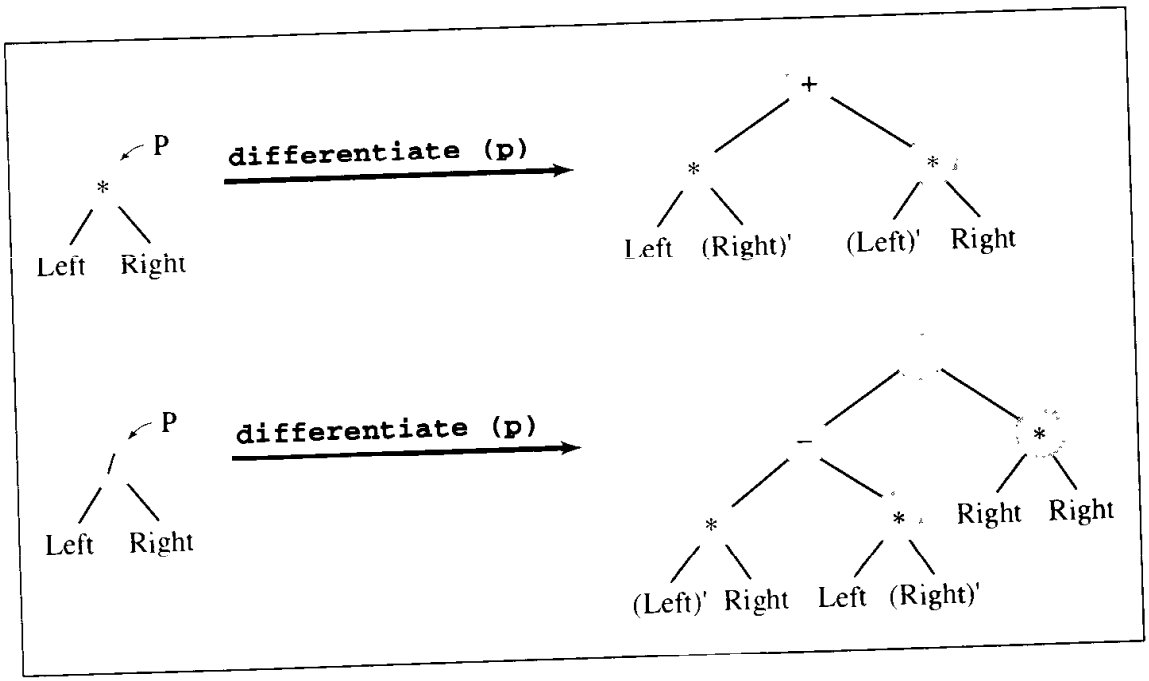


Figure 6.61 Tree transformations for differentiation of multiplication and division.




```
if (p->key is another id or a number)
    return new ExprTreeNode("0");
if (p->key is '+' or '-') {
    return new ExprTreeNode(p->key,differentiate(p->left,x),
                                   differentiate(p->right,x));

if (p->key is '*')
    ExprTreeNode *q = new ExprTreeNode("+");
    q->left = new ExprTreeNode("*",p->left,new ExprTreeNode(*p->right));
    q->left->right = differentiate(q->left->right,x);
    q->right = new ExprTreeNode("*",new ExprTreeNode(*p->left),p->right);
```

```
        q->right->left = differentiate(q->right->left,x);
        return q;
    . . . . . .
}
```

Here `p` is a pointer to the expression to be differentiated with respect to `x`.

The rule for division is left as an exercise.

6.11 Case Study: Computing Word Frequencies

One tool in establishing authorship of text in cases when the text is not signed, or it is attributed to someone else, is using word frequencies. If it is known that an author A wrote text T_1 and the distribution of word frequencies in a text T_2 under scrutiny is very close to the frequencies in T_1, then it is likely that T_2 was written by author A.

Regardless of how reliable this method is for literary studies, our interest lies in writing a program that scans a text file and computes the frequency of the occurrence of words in this file. For the sake of simplification, punctuation marks are disregarded and case sensitivity is disabled. Therefore, the word *man's* is counted as two words, *man* and *s*, although in fact it may be one word (for possessive) and not two words (contraction for *man is* or *man has*). But contractions are counted separately; for example, *s* from *man's* is considered a separate word. Similarly, separators in the middle of words such as hyphens cause portions of the same words to be considered separate words. For example, *pre-existence* is split into *pre* and *existence*. Also, by disabling case sensitivity, *Good* in the phrase *Mr. Good* is considered as another occurrence of the word *good*. On the other hand, *Good* used in its normal sense at the beginning of a sentence is properly included as another occurrence of *good*.

This program focuses not so much on linguistics as on building a self-adjusting binary search tree using the semisplaying technique. If a word is encountered in the file for the first time, it is inserted in the tree. Otherwise, the semisplaying is started from the node corresponding to this word.

Another concern is storing all predecessors when scanning the tree. It is achieved by using a pointer to the parent. In this way, from each node we can access any predecessor of this node up to the root of the tree.

Figure 6.62 shows the structure of the tree using the content of a short file, and Figure 6.63 contains the complete code. The program reads a word, which is any sequence of alphanumeric characters that starts with a letter (spaces, punctuation marks, etc. are discarded) and checks whether the word is in the tree. If so, the semisplaying technique is used to reorganize the tree and then the word's frequency count is incremented. Note that this movement toward the root is accomplished by changing links of the nodes involved, not by physically transferring information from one node to its parent and then to its grandparent and so on. If a word is not found in the tree, it is inserted in the tree by creating a new leaf for it. After all words are processed, an inorder tree traversal goes through the tree to count all the nodes and add all frequency counts to print as the final result the number of words in the tree and the number of words in the file.

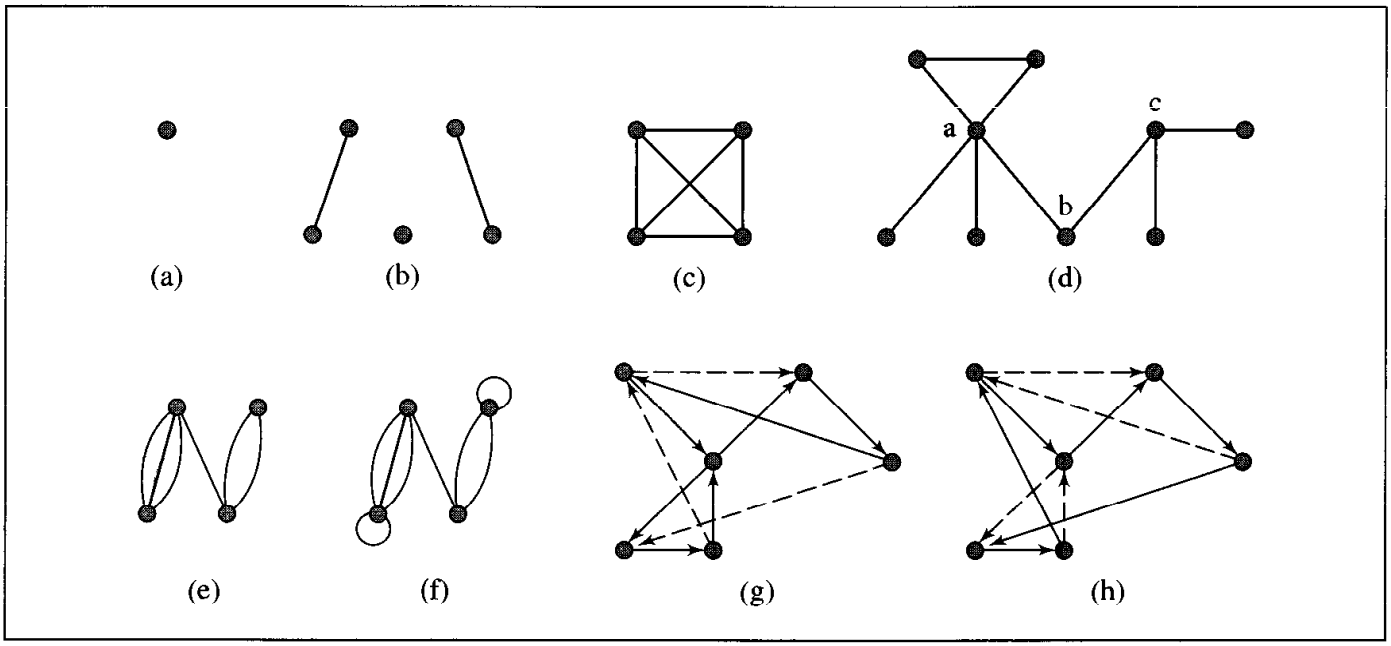
Graphs

In spite of the flexibility of trees and the many different tree applications, trees, by their nature, have one limitation, namely, they can only represent relations of a hierarchical type, such as relations between parent and child. Other relations are only represented indirectly, such as the relation of being a sibling. A generalization of a tree, a *graph,* is a data structure in which this limitation is lifted. Intuitively, a graph is a collection of vertices (or nodes) and the connections between them. Generally, no restriction is imposed on the number of vertices in the graph or on the number of connections one vertex can have to other vertices. Figure 8.1 contains examples of graphs. Graphs are versatile data structures that can represent a large number of different situations and events from diverse domains. Graph theory has grown into a sophisticated area of mathematics and computer science in the last 200 years since it was first studied. Many results are of theoretical interest, but in this chapter, some selected results of interest to computer scientists are presented. Before discussing different algorithms and their applications, several definitions need to be introduced.

A *simple graph* $G = (V, E)$ consists of a nonempty set V of *vertices* and a possibly empty set E of *edges,* each edge being a set of two vertices from V. The number of vertices and edges is denoted by $|V|$ and $|E|$, respectively. A *directed graph,* or a *digraph,* $G = (V, E)$ consists of a nonempty set V of vertices and a set E of edges (also called *arcs*), where each edge is a pair of vertices from V. The difference is that one edge of a simple graph is of the form $\{v_i, v_j\}$, and for such an edge, $\{v_i,v_j\} = \{v_j,v_i\}$. In a digraph, each edge is of the form (v_i,v_j), and in this case, $(v_i,v_j) \neq (v_j,v_i)$. Unless necessary, this distinction in notation will be disregarded, and an edge between vertices v_i and v_j will be referred to as $edge(v_i,v_j)$.

These definitions are restrictive in that they do not allow for two vertices to have more than one edge. A *multigraph* is a graph in which two vertices can be joined by multiple edges. Geometric interpretation is very simple (see Figure 8.1e). Formally,

FIGURE 8.1 Examples of graphs: (a–d) simple graphs; (c) a complete graph K_4; (e) a multigraph; (f) a pseudograph; (g) a circuit in a digraph; (h) a cycle in the digraph.



the definition is as follows: A multigraph $G = (V,E,f)$ is composed of a set of vertices V, a set of edges E, and a function $f: E \rightarrow \{\{v_i,v_j\} : v_i,v_j \in V \text{ and } v_i \neq v_j\}$. A *pseudograph* is a multigraph with the condition $v_i \neq v_j$ removed, which allows for *loops* to occur; in a pseudograph, a vertex can be joined with itself by an edge (Figure 8.1f).

A *path* from v_1 to v_n is a sequence of edges $edge(v_1v_2)$, $edge(v_2v_3)$, . . . , $edge(v_{n-1}v_n)$ and is denoted as path $v_1, v_2, v_3, \ldots, v_{n-1}, v_n$. If $v_1 = v_n$ and no edge is repeated, then the path is called a *circuit* (Figure 8.1g). If all vertices in a circuit are different, then it is called a *cycle* (Figure 8.1h).

A graph is called a *weighted graph* if each edge has an assigned number. Depending on the context in which such graphs are used, the number assigned to an edge is called its weight, cost, distance, length, or some other name.

A graph with n vertices is called *complete* and is denoted K_n if for each pair of distinct vertices there is exactly one edge connecting them; that is, each vertex can be connected to any other vertex (Figure 8.1c). The number of edges in such a graph $|E| = \binom{|V|}{2} = \frac{|V|!}{2!(|V|-2)!} = \frac{|V|(|V|-1)}{2} = O(|V|^2)$.

A *subgraph* G' of graph $G = (V,E)$ is a graph (V',E') such that $V' \subseteq V$ and $E' \subseteq E$. A subgraph *induced* by vertices V' is a graph (V',E') such that an edge $e \in E$ if $e \in E'$.

Two vertices v_i and v_j are called *adjacent* if the $edge(v_iv_j)$ is in E. Such an edge is called *incident with* the vertices v_i and v_j. The *degree* of a vertex v, $deg(v)$, is the number of edges incident with v. If $deg(v) = 0$, then v is called an *isolated vertex*. The definition of a graph indicating that the set of edges E can be empty allows for a graph consisting only of isolated vertices.

8.1 Graph Representation

There are variety of ways to represent a graph. A simple representation is given by an *adjacency list* which specifies all vertices adjacent to each vertex of the graph. This list can be implemented as a table, in which case it is called a *star representation*, which can be forward or reverse, as illustrated in Figure 8.2b, or as a linked list (Figure 8.2c).

Another representation is a matrix which comes in two forms: an adjacency matrix and an incidence matrix. An *adjacency matrix* of graph $G = (V,E)$ is a binary $|V| \times |V|$ matrix such that each entry of this matrix

$$a_{ij} = \begin{cases} 1 & \text{if there exists an } edge(v_i v_j) \\ 0 & \text{otherwise} \end{cases}$$

An example is shown in Figure 8.2d. Note that the order of vertices $v_1, \ldots, v_{|V|}$ used for generating this matrix is arbitrary; therefore, there are $n!$ possible adjacency matrices for the same graph G. Generalization of this definition to also cover multigraphs can be easily accomplished by transforming the definition into the following form:

$$a_{ij} = \text{number of edges between } v_i \text{ and } v_j$$

Another matrix representation of a graph is based on the incidence of vertices and edges and is called an *incidence matrix*. An incidence matrix of graph $G = (V,E)$ is a $|V| \times |E|$ matrix such that

$$a_{ij} = \begin{cases} 1 & \text{if edge } e_j \text{ is incident with vertex } v_i \\ 0 & \text{otherwise} \end{cases}$$

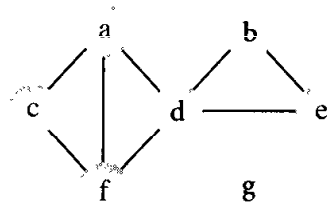
Figure 8.2e contains an example of an incidence matrix. In an incidence matrix for a multigraph, some columns are the same, and a column with only one 1 indicates a loop.

Which representation is best? It depends on the problem at hand. If our task is to process vertices adjacent to a vertex v, then the adjacency list requires only $deg(v)$ steps, whereas the adjacency matrix requires $|V|$ steps. On the other hand, inserting or deleting a vertex adjacent to v requires linked list maintenance for an adjacency list (if such an implementation is used); for a matrix, it requires only changing 0 to 1 for insertion, or 1 to 0 for deletion, in one cell of the matrix.

8.2 Graph Traversals

As in trees, traversing a graph consists of visiting each vertex only one time. The simple traversal algorithms used for trees cannot be applied here because graphs may include cycles; hence, the tree traversal algorithms would result in infinite loops. To prevent that from happening, each visited vertex can be marked to avoid revisiting it. However, graphs can have isolated vertices, which means that some parts of the graph are left out if unmodified tree traversal methods are applied.

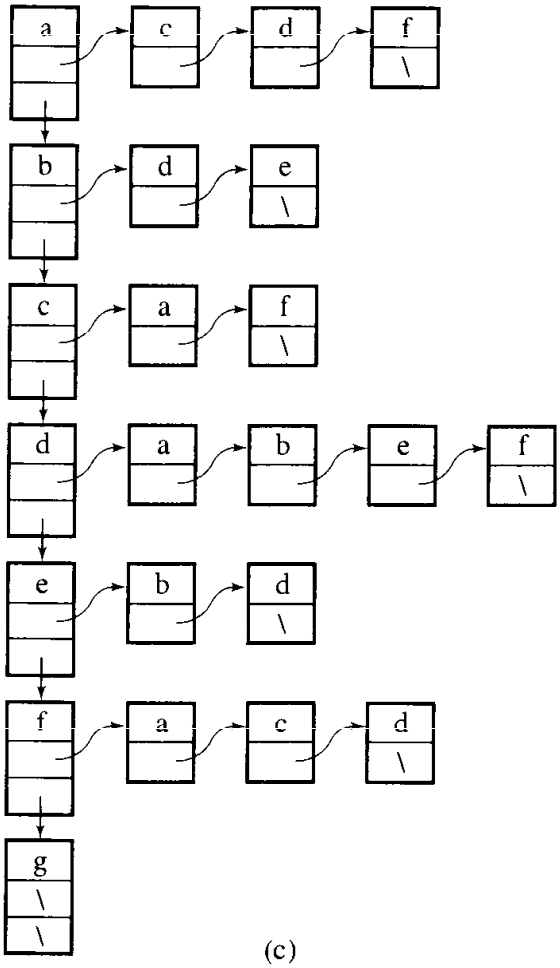
FIGURE **8.2** Graph representations. (a) A graph represented as (b–c) an adjacency list, (d) an adjacency matrix, and (e) an incidence matrix.



(a)

a	c	d	f	
b	d	e		
c	a	f		
d	a	b	e	f
e	b	d		
f	a	c	d	
g				

(b)



(c)

	a	**b**	**c**	**d**	**e**	**f**	**g**
a	0	0	1	1	0	1	0
b	0	0	0	1	1	0	0
c	1	0	0	0	0	1	0
d	1	1	0	0	1	1	0
e	0	1	0	1	0	0	0
f	1	0	1	1	0	0	0
g	0	0	0	0	0	0	0

(d)

	ac	**ad**	**af**	**bd**	**be**	**cf**	**de**	**df**
a	1	1	1	0	0	0	0	0
b	0	0	0	1	1	0	0	0
c	1	0	0	0	0	1	0	0
d	0	1	0	1	0	0	1	1
e	0	0	0	0	1	0	1	0
f	0	0	1	0	0	1	1	0
g	0	0	0	0	0	0	0	0

(e)

An algorithm for traversing a graph, known as the depth-first search algorithm, was developed by John Hopcroft and Robert Tarjan. In this algorithm, each vertex v is visited and then each unvisited vertex adjacent to v is visited. If a vertex v has no adjacent vertices or all of its adjacent vertices have been visited, we backtrack to the predecessor of v. The traversal is finished if this visiting and backtracking process leads to

the first vertex where the traversal started. If there are still some unvisited vertices in the graph, the traversal continues restarting for one of the unvisited vertices.

Although it is not necessary for the proper outcome of this method, the algorithm assigns a unique number to each accessed vertex so that vertices are now renumbered. This will prove useful in later applications of this algorithm.

```
DFS(v)
   num(v)= i++;
   for all vertices u adjacent to v
      if num(u) is 0
         attach edge(uv) to edges;
         DFS(u);

depthFirstSearch()
   for all vertices v
      num(v) = 0;
   edges = null;
   i = 1;
   while there is a vertex v such that num(v) is 0
      DFS(v);
   output edges;
```

Figure 8.3 contains an example with the numbers *num*(*v*) assigned to each vertex *v* shown in parentheses. Having made all necessary initializations, `depthFirstSearch()` calls `DFS(a)`. `DFS()` is first invoked for vertex *a; num*(*a*) is assigned number 1. *a* has four adjacent vertices, and vertex *e* is chosen for the next invocation, `DFS(e)`, which assigns number 2 to this vertex, that is, *num*(*e*) = 2, and puts the *edge*(*ae*) in `edges`. Vertex *e* has two unvisited adjacent vertices, and `DFS()` is called for the first of them, the vertex *f*. The call `DFS(f)` leads to the assignment *num*(*f*) = 3 and puts the *edge*(*ef*) in `edges`. Vertex *f* has only one unvisited adjacent vertex, *i;* thus, the fourth call, `DFS(i)`, leads to the assignment *num*(*i*) = 4 and to the attaching of *edge*(*fi*) to `edges`. Vertex *i* has only visited adjacent vertices; hence, we return to call `DFS(f)` and then to `DFS(e)` in which vertex *i* is accessed only to learn that *num*(*i*) is not 0, whereby the *edge*(*ei*) is not included in `edges`. The rest of the execution can be seen easily in Figure 8.3b. Solid lines indicate edges included in the set `edges`.

Note that this algorithm guarantees generating a tree (or a forest, a set of trees) which includes or spans over all vertices of the original graph. A tree that meets this condition is called a *spanning tree.* The fact that a tree is generated is ascertained by the fact that the algorithm does not include in the resulting tree any edge which leads from the currently analyzed vertex to a vertex already analyzed. An edge is added to `edges` only if the condition in "`if` *num*(u) *is* 0" is true, that is, if vertex *u* reachable from vertex *v* has not been processed. As a result, certain edges in the original graph do not appear in the resulting tree. The edges included in this tree are called *forward edges* (or *tree edges*), and the edges not included in this tree are called *back edges* and are shown as dashed lines.

Figure 8.4 illustrates the execution of this algorithm for a digraph. Notice that the original graph results in three spanning trees, although we started with only two isolated subgraphs.

FIGURE 8.3 An example of application of `depthFirstSearch()` algorithm to a graph.

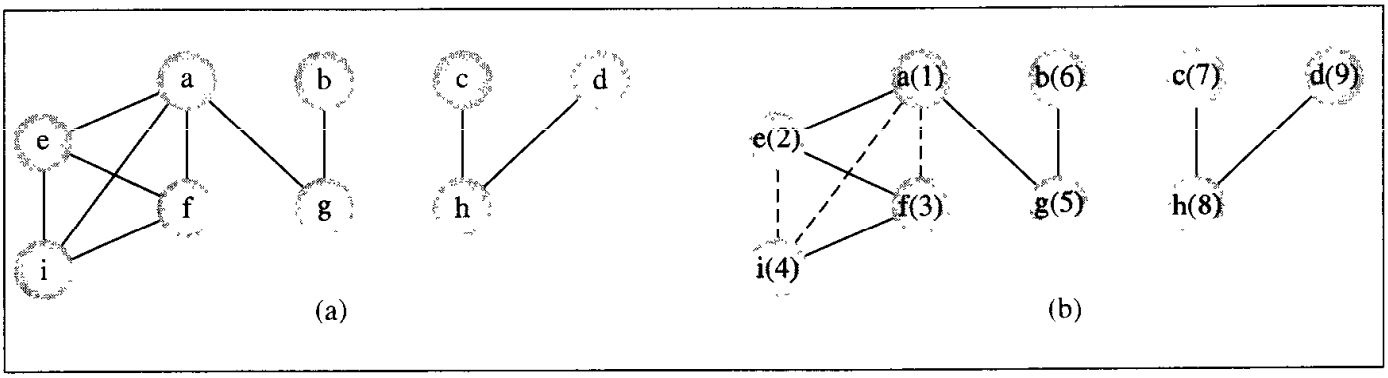
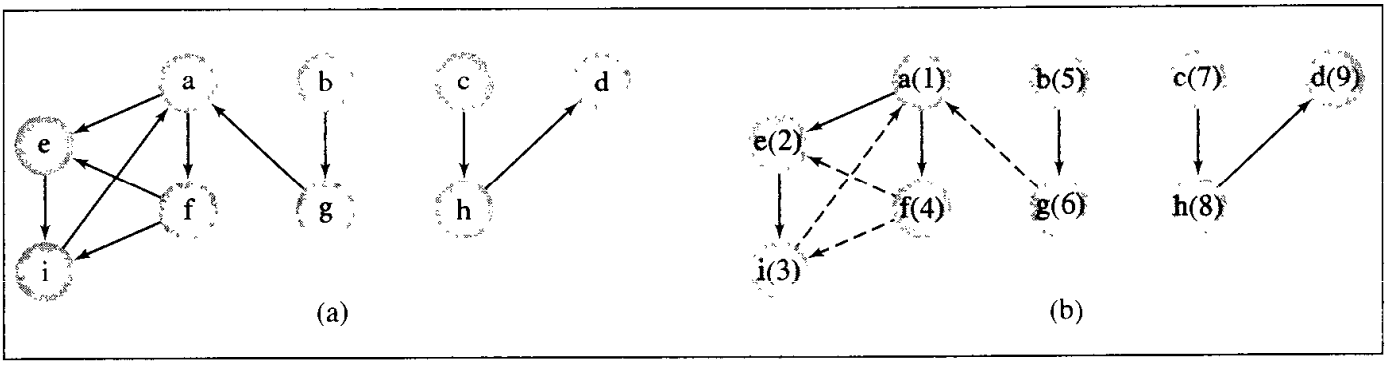


FIGURE 8.4 `depthFirstSearch()` algorithm applied to a digraph.



The complexity of `depthFirstSearch()` is $O(|V| + |E|)$ because (a) initializing $num(v)$ for each vertex v requires $|V|$ steps; (b) `DFS(v)` is called $deg(v)$ times for each v, that is, once for each edge of v (to spawn into more calls or to finish the chain of recursive calls), hence, the total number of calls is $2|E|$; (c) searching for vertices as required by the statement

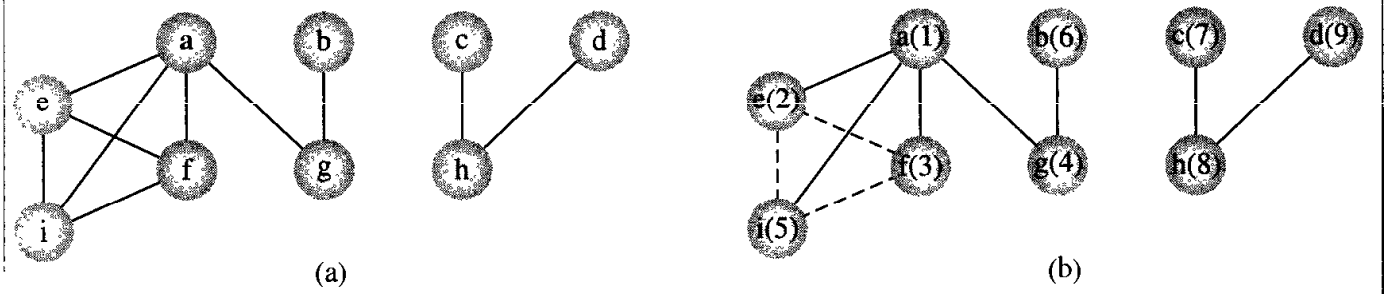
`while` *there is a vertex* `v` *such that* num`(v)` *is* `0`

can be assumed to require $|V|$ steps. For a graph with no isolated parts, the loop makes only one iteration, and an initial vertex can be found in one step, although it may take $|V|$ steps. For a graph with all isolated vertices, the loop iterates $|V|$ times and each time a vertex can also be chosen in one step, although in an unfavorable implementation, the ith iteration may require i steps, whereby the loop would require $O(|V|^2)$ steps in total. For example, if an adjacency list is used, then for each v, the condition in the loop,

`for` *all vertices* `u` *adjacent to* `v`

is checked deg(v) times. However, if an adjacency matrix is used, then the same condition is used $|V|$ times, whereby the algorithm's complexity becomes $O(|V|^2)$.

FIGURE 8.5 An example of application of `breadthFirstSearch()` algorithm to a graph.

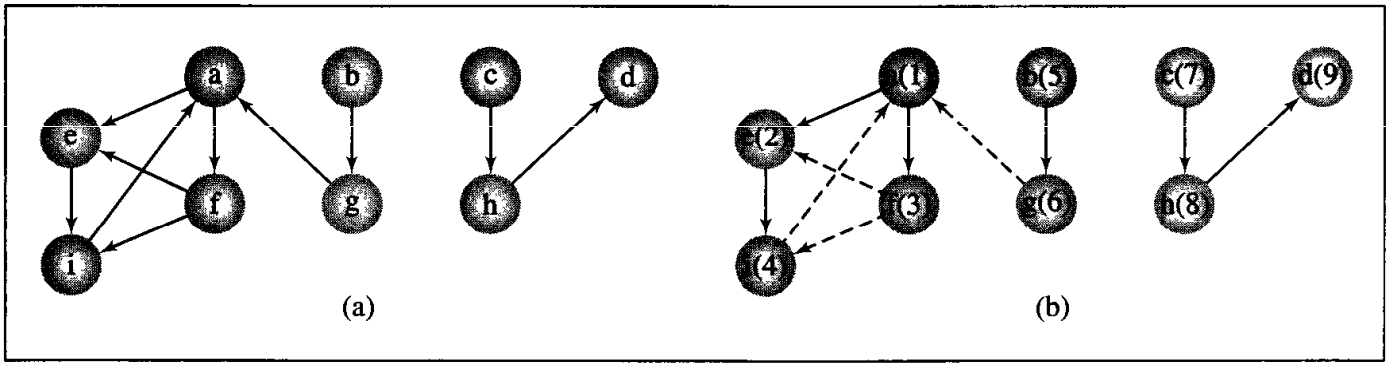


As we shall see, many different algorithms are based on `DFS()`; however, some algorithms are more efficient if the underlying graph traversal is not depth first but breadth first. We have already encountered these two types of traversals in Chapter 6; recall that the depth-first algorithms rely on the use of a stack (explicitly, or implicitly, in recursion), and breadth-first traversal uses a queue as the basic data structure. Not surprisingly, this idea can also be extended to graphs, as shown in the following pseudocode:

```
breadthFirstSearch()
   for all vertices u
      num(u) = 0;
   edges = null;
   i = 1;
   while there is a vertex v such that num(v) == 0
      num(v)=i++;
      enqueue(v);
      while queue is not empty
         v = dequeue();
         for all vertices u adjacent to v
            if num(u) is 0
               num(u) = i++;
               enqueue(u);
               attach edge(vu) to edges;
   output edges;
```

Examples of processing a simple graph and a digraph are shown in Figures 8.5 and 8.6. `breadthFirstSearch()` first tries to mark all neighbors of a vertex v before proceeding to other vertices, whereas `DFS()` picks one neighbor of a v and then proceeds to a neighbor of this neighbor before processing any other neighbors of v.

FIGURE 8.6 `breadthFirstSearch()` algorithm applied to a digraph.



8.3 SHORTEST PATHS

Finding the shortest path is a classical problem in graph theory, and a large number of different solutions have been proposed. Edges are assigned certain weights representing, for example, distances between cities, times separating the execution of certain tasks, costs of transmitting information between locations, amounts of some substance transported from one place to another, etc. When determining the shortest path from vertex v to vertex u, information about distances between intermediate vertices w has to be recorded. This information can be recorded as a label associated with these vertices, where the label is only the distance from v to w or the distance along with the predecessor of w in this path. The methods of finding the shortest path rely on these labels. Depending on how many times these labels are updated, the methods solving the shortest path problem are divided in two classes: label-setting methods and label-correcting methods.

For *label-setting methods*, in each pass through the vertices still to be processed, one vertex is set to a value which remains unchanged to the end of the execution. This, however, limits such methods to processing graphs with only positive weights. The second category includes *label-correcting methods* which allow for the changing of *any* label during application of the method. These two methods can be applied to graphs with negative weights and with no *negative cycle*—a cycle composed of edges with weights adding up to a negative number—but they guarantee that, for all vertices, the current distances indicate the shortest path only after the processing of the graph is finished. Most of the label-setting and label-correcting methods, however, can be subsumed to the same form which allows finding the shortest paths from one vertex to all other vertices (Gallo and Pallottino 1986):

```
genericShortestPathAlgorithm(weighted simple digraph, vertex first)
   for all vertices v
      currDist(v) = ∞;
   currDist(first) = 0;
   initialize toBeChecked;
```

```
while toBeChecked is not empty
   v = a vertex in toBeChecked;
   remove v from toBeChecked;
   for all vertices u adjacent to v
      if currDist(u) > currDist(v) + weight(edge(vu))
         currDist(u) = currDist(v) + weight(edge(vu));
         predecessor(u) = v;
         add u to toBeChecked if it is not there;
```

In this generic algorithm, a label consists of two elements:

$$label(v) = (currDist(v), predecessor(v))$$

This algorithm leaves two things open: the organization of the set `toBeChecked` and the order of assigning new values to v in the assignment statement

```
v = a vertex in toBeChecked;
```

It should be clear that the organization of `toBeChecked` can determine the order of choosing new values for v, but it also determines the efficiency of the algorithm.

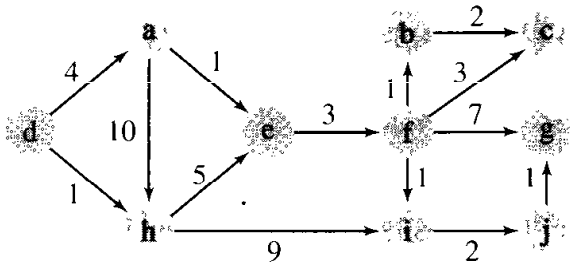
What distinguishes label-setting methods from label-correcting methods is the method of choosing the value for v, which is always a vertex in `toBeChecked` with the smallest current distance. One of the first label-setting algorithms was developed by Dijkstra.

In Dijkstra's algorithm, a number of paths $p_1, \ldots, p_n$ from a vertex v are tried, and each time, the shortest path is chosen among them, which may mean that the same path p_i can be continued by adding one more edge to it. But if p_i turns out to be longer than any other path that can be tried, p_i is abandoned and this other path is tried by resuming from where it was left and by adding one more edge to it. Since paths can lead to vertices with more than one outgoing edge, new paths for possible exploration are added for each outgoing edge. Each vertex is tried once, all paths leading from it are opened, and the vertex itself is put away and not used anymore. After all vertices are visited, the algorithm is finished. Dijkstra's algorithm is as follows:

```
DijkstraAlgorithm(weighted simple digraph, vertex first)
   for all vertices v
      currDist(v) = ∞;
   currDist(first) = 0;
   toBeChecked = all vertices;
   while toBeChecked is not empty
      v = a vertex in toBeChecked with minimal currDist(v);
      remove v from toBeChecked;
      for all vertices u adjacent to v and in toBeChecked
         if currDist(u) > currDist(v)+ weight(edge(vu))
            currDist(u) = currDist(v)+ weight(edge(vu));
            predecessor(u) = v;
```

Dijkstra's algorithm is obtained from the generic method by being more specific about which vertex is to be taken from `toBeChecked` so that the line

FIGURE 8.7 An execution of `DijkstraAlgorithm()`.



(a)

iteration:	init	1	2	3	4	5	6	7	8	9	10
active vertex:		d	h	a	e	f	b	i	c	j	g
a	∞	4	4								
b	∞	∞	∞	∞	∞	9					
c	∞	∞	∞	∞	∞	11	11	11			
d	0										
e	∞	∞	6	5							
f	∞	∞	∞	∞	8						
g	∞	∞	∞	∞	∞	15	15	15	15	12	
h	∞	1									
i	∞	∞	10	10	10	9	9				
j	∞	∞	∞	∞	∞	∞	∞	11	11		

(b)

`v =` *a vertex in* `toBeChecked;`

is replaced by the line

`v =` *a vertex in* `toBeChecked` *with minimal currDist*`(v);`

and by extending the condition in the `if` statement whereby the current distance of vertices eliminated from `toBeChecked` is set permanently.[1] Note that the structure of `toBeChecked` is not specified, and the efficiency of the algorithms depends on the data type of `toBeChecked`, which determines how quickly a vertex with minimal distance can be retrieved.

Figure 8.7 contains an example. The table in this figure shows all iterations of the `while` loop. There are ten iterations because there are ten vertices. The table indicates the current distances determined up until the current iteration.

The list `toBeChecked` is initialized to $\{a\ b \ldots j\}$, the current distances of all vertices are initialized to a very large value, marked here as ∞, and in the first iteration,

[1] Dijkstra used six sets to ascertain this condition, three for vertices and three for edges.

the current distances of d's neighbors are set to numbers equal to the weights of the edges from d. Now, there are two candidates for the next try, a and h, since d was excluded from `toBeChecked`. In the second iteration, h is chosen, since its current distance is minimal, and then the two vertices accessible from h, namely, e and i, acquire the current distances 6 and 10. Now, there are three candidates in `toBeChecked` for the next try, a, e, and i. Since a has the smallest current distance, it is chosen in the third iteration. Eventually, in the tenth iteration, `toBeChecked` becomes empty and the execution of the algorithm completes.

The complexity of Dijkstra's algorithm is $O(|V|^2)$. The first `for` loop and the `while` loop are executed $|V|$ times. For each iteration of the `while` loop, (a) a vertex v in `toBeChecked` with minimal current distance has to be found, which requires $O(|V|)$ steps, and (b) the `for` loop iterates $deg(v)$ times, which is also $O(|V|)$. The efficiency can be improved by using a heap to store and order vertices and adjacency lists (Johnson 1977). Using a heap turns the complexity of this algorithm into $O((|E| + |V|) \lg |V|)$; each time through the `while` loop, the cost of restoring the heap after removing a vertex is proportional to $O(\lg|V|)$. Also, in each iteration, only adjacent vertices are updated on an adjacency list so that the total updates for all vertices considered in all iterations are proportional to $|E|$, and each list update corresponds to the cost of $\lg|V|$ of the heap update.

Dijkstra's algorithm is not general enough in that it fails when negative weights are used in graphs. To see why, change the weight of *edge*(*ah*) from 10 to –10. Note that the path *d*, *a*, *h*, *e* is now –1, whereas the path *d*, *a*, *e* as determined by the algorithm is 5. The reason for overlooking this less costly path is that the vertices with the current distance set from ∞ to a value are not checked anymore: First, successors of vertex d are scrutinized and d is removed from `toBeChecked`, then the vertex h is removed from `toBeChecked`, and only afterward is the vertex a considered as a candidate to be included in the path from d to other vertices. But now, the *edge*(*ah*) is not taken into consideration because the condition in the `for` loop prevents the algorithm from doing this. To overcome this limitation, a label-correcting method is needed.

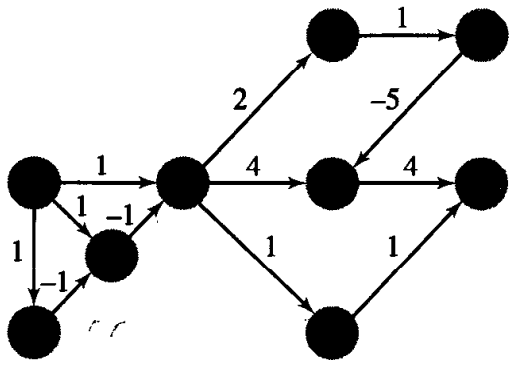
One of the first label-correcting algorithms was devised by Lester Ford. Like Dijkstra's algorithm, it uses the same method of setting current distances, but Ford's method does not permanently determine the shortest distance for any vertex until it processes the entire graph. It is more powerful than Dijkstra's method in that it can process graphs with negative weights (but not graphs with negative cycles).

As required by the original form of the algorithm, all edges are monitored to find a possibility for an improvement of the current distance of vertices so that the algorithm can be presented in this pseudocode:

```
FordAlgorithm(weighted simple digraph, vertex first)
   for all vertices v
      currDist(v) = ∞;
   currDist(first) = 0;
   while there is an edge(vu) such that currDist(u) > currDist(v)+ weight(edge(vu))
      currDist(u) = currDist(v)+ weight(edge(vu));
```

To impose a certain order on monitoring the edges, an alphabetically ordered sequence of edges can be used so that the algorithm can repeatedly go through the entire

FIGURE 8.8 `FordAlgorithm()` applied to a digraph with negative weights.



(a)

the order of edges: *ab be cd cg ch da de di ef gd hg if*

	iteration					
	init	**1**		**2**	**3**	**4**
a	∞	3	3	2	1	
b	∞	∞	∞	4	3	2
c	0					
d	∞	1	0	–1		
e	∞	5	5	–1	–2	–3
f	∞	9	3	2	1	
g	∞	1	0			
h	∞	1				
i	∞	2	2	1	0	

(b)

sequence and adjust the current distance of any vertex if needed. Figure 8.8 contains an example. The graph includes edges with negative weights. The table indicates iterations of the `while` loop and current distances updated in each iteration, where one iteration is defined as one pass through the edges. Note that a vertex can change its current distance during the same iteration. However, at the end, each vertex of the graph can be reached through the shortest path from the starting vertex (vertex *c* in the example in Figure 8.8).

The computational complexity of this algorithm is $O(|V||E|)$. There will be at most $|V| - 1$ passes through the sequence of $|E|$ edges, since $|V| - 1$ is the largest number of edges in any path. In the first pass, at least all one-edge paths are determined, in the second pass, all two-edge paths are determined, and so on. However, for graphs with irrational weights, this complexity is $O(2^{|V|})$ (Gallo and Pallottino 1986).

We have seen in the case of Dijkstra's algorithm that the efficiency of an algorithm can be improved by scanning edges and vertices in a certain order, which in turn depends on the data structure used to store them. The same holds true for label-correcting methods. In particular, `FordAlgorithm()` does not specify the order of checking edges. In the example illustrated in Figure 8.8, a simple solution is used in that all adjacency lists of all vertices were visited in each iteration. However, in this approach, all the edges are checked every time, which is not necessary, and more judicious organization of the list of vertices can limit the number of visits per vertex. Such an improvement is based on the `genericShortestPathAlgorithm()` by explicitly referring to the `toBeChecked` list which in `FordAlgorithm()` is used only implicitly: It simply is the set of all vertices V and remains such for the entire run of the

algorithm. This leads us to a general form of a label-correcting algorithm as expressed in this pseudocode:

```
labelCorrectingAlgorithm(weighted simple digraph, vertex first)
   for all vertices v
      currDist(v) = ∞;
   currDist(first) = 0;
   toBeChecked = {first};
   while toBeChecked is not empty
      v = a vertex in toBeChecked;
      remove v from toBeChecked;
      for all vertices u adjacent to v
         if currDist(u) > currDist(v)+ weight(edge(vu))
            currDist(u) = currDist(v)+ weight(edge(vu));
            predecessor(u) = v;
            add u to toBeChecked if it is not there;
```

The efficiency of particular instantiations of this algorithm hinges on the data structure used for the `toBeChecked` list and on operations for extracting elements from this list and including them into it.

One possible organization of this list is a queue: Vertex v is dequeued from `toBeChecked` and if the current distance of any of its neighbors, u, is updated, u is enqueued onto `toBeChecked`. It seems like a natural choice, and in fact, it was one of the earliest, used in 1968 by C. Witzgall (Deo and Pang 1984). However, it is not without flaws, as it sometimes reevaluates the same labels more times than necessary. Figure 8.9 contains an example of an excessive reevaluation. The table in this figure shows all changes on `toBeChecked` implemented as a queue when `labelCorrectingAlgorithm()` is applied to the graph in Figure 8.8a. The vertex d is updated three times. These updates cause three changes to its successors, a and i, and two changes to another successor, e. The change of a translates into two changes to b and these into two more changes to e. To avoid such repetitious updates, a doubly ended queue, or deque, can be used.

The choice of a deque as a solution to this problem is attributed to D. D'Esopo (Pollack and Wiebenson 1960) and was implemented by Pape. In this method, the vertices included in `toBeChecked` for the first time are put at the end of the list; otherwise, they are added at the front. The rationale for this procedure is that if a vertex v is included for the first time, then there is a good chance that the vertices accessible from v have not been processed yet, so they will be processed after processing v. On the other hand, if v has been processed at least once, then it is likely that the vertices reachable from v are still on the list waiting for processing; by putting v at the end of the list, these vertices may very likely be reprocessed due to the update of *currDist*(v). Therefore, it is better to put v in front of their successors to avoid an unnecessary round of updates. Figure 8.10 shows changes in the deque during the execution of `labelCorrectingAlgorithm()` applied to the graph in Figure 8.8a. This time, the number of iterations is dramatically reduced. Although d is again evaluated three times, these evaluations are performed before processing its successors so that a and i are processed once and e twice. However, this algorithm has a problem of its own

FIGURE 8.9 An execution of `labelCorrectingAlgorithm()`, which uses a queue.

		active vertex																				
		c	d	g	h	a	e	i	d	g	b	f	a	e	i	d	b	f	a	i	e	b	f	e
queue		d	g	h	a	e	i	d	g	b	f	a	e	i	d	b	f	a	i	e	b	f	e	
		g	h	a	e	i	d	g	b	f	a	e	i	d	b	f	a	i	e	b	f	e		
		h	a	e	i	d	g	b	f	a	e	i	d	b	f	a	i	e	b	f				
			e	i	d	g	b	f	a	e	i	d	b			i	e							
			i	d	g	b	f		e	i	d													
									i	d														
a	∞	∞	3	3	3	3	3	3	2	2	2	2	2	2	2	1								
b	∞	∞	∞	∞	∞	4	4	4	4	4	4	4	3	3	3	3	3	3	2					
c	0																							
d	∞	1	1	0	0	0	0	0	0	−1														
e	∞	∞	5	5	5	5	5	5	4	4	−1	−1	−1	−1	−1	−1	−2	−2	−2	−2	−2	−3		
f	∞	∞	∞	∞	∞	∞	9	3	3	3	3	3	3	3	2	2	2	2	2	1				
g	∞	1	1	1	0																			
h	∞	1																						
i	∞	∞	2	2	2	2	2	2	1	1	1	1	1	1	1	0								

FIGURE 8.10 An execution of `labelCorrectingAlgorithm()`, which applies a deque.

		active vertex												
		c	d	g	d	h	g	d	a	e	i	b	e	f
deque		d	g	d	h	g	d	a	e	i	b	e	f	
		g	h	h	a	a	a	e	i	b	f	f		
		h	a	a	e	e	e	i	b	f				
			e	e	i	i	i							
			i	i										
a	∞	∞	3	3	2	2	2	1						
b	∞	∞	∞	∞	∞	∞	∞	∞	2					
c	0													
d	∞	1	1	0	0	0	−1							
e	∞	∞	5	5	4	4	4	3	3	3	3	−3		
f	∞	∞	∞	∞	∞	∞	∞	∞	∞	7	1			
g	∞	1	1	1	1	0								
h	∞	1												
i	∞	∞	2	2	1	1	1	0						

because in the worst case its performance is an exponential function of the number of vertices. (See Exercise 13 at the end of this chapter.) But in the average case, as Pape's experimental runs indicate, this implementation fares at least 60% better than the previous queue solution.

Instead of using a deque, which combines two queues, the two queues can be used separately. In this version of the algorithm, vertices stored for the first time are enqueued on $queue_1$ and on $queue_2$ otherwise. Vertices are dequeued from $queue_1$ if it is not empty and from $queue_2$ otherwise (Gallo and Pallottino 1988).

Another version of the label-correcting method is the *threshold algorithm,* which also uses two lists. Vertices are taken for processing from $list_1$. A vertex is added to the end of $list_1$ if its label is below the current threshold level and to $list_2$ otherwise. If $list_1$ is empty, then the threshold level is changed to a value greater than a minimum label among the labels of the vertices in $list_2$, and then the vertices with the label values below the threshold are moved to $list_1$ (Glover, Glover, and Klingman 1984).

Still another algorithm is a *small label first* method. In this method, a vertex is included at the front of a deque if its label is smaller than the label at the top of the deque; otherwise, it is put at the end of the deque (Bertsekas 1993). To some extent, this method includes the main criterion of label-setting methods. The latter methods always retrieve the minimal element from the list; the small label first method puts a vertex with the label smaller than the label of the front vertex at the top. The approach can be carried to its logical conclusion by requiring each vertex to be included in the list according to its rank so that the deque turns into a priority queue and the resulting method becomes a label-correcting version of Dijkstra's algorithm.

8.3.1 All-to-All Shortest Path Problem

Although the task of finding all shortest paths from any vertex to any other vertex seems to be more complicated than the task of dealing with one source only, a method designed by Stephen Warshall and implemented by Robert W. Floyd and P. Z. Ingerman does it in a surprisingly simple way provided an adjacency matrix is given that indicates all the edge weights of the graph (or digraph). The graph can include negative weights. The algorithm is as follows:

```
WFIalgorithm(matrix weight)
   for i = 1 to |V|
      for j = 1 to |V|
         for k = 1 to |V|
            if weight[j][k] > weight[j][i] + weight[i][k]
               weight[j][k] = weight[j][i] + weight[i][k];
```

The outermost loop refers to vertices which may be on a path between the vertex with index j and the vertex with index k. For example, in the first iteration, when $i = 1$, all paths $v_j \ldots v_1 \ldots v_k$ are considered, and if there is currently no path from v_j to v_k and v_k is reachable from v_j, the path is established, with its weight equal to $p = weight(path(v_j v_1)) + weight(path(v_1 v_k))$, or the current weight of this path, $weight(path(v_j v_k))$, is changed to p if p is less than $weight(path(v_j v_k))$. As an example,

consider the graph and the corresponding adjacency matrix in Figure 8.11. This figure also contains tables that show changes in the matrix for each value of i and the changes in paths as established by the algorithm. After the first iteration, the matrix and the graph remain the same, since a has no incoming edges (Figure 8.11a). They also remain the same in the last iteration, when $i = 5$; no change is introduced to the matrix because vertex e has no outgoing edges. A better path, one with a lower combined weight, is always chosen, if possible. For example, the direct one-edge path from b to e in Figure 8.11c is abandoned after a two-edge path from b to e is found with a lower weight, as in Figure 8.11d.

This algorithm also allows us to detect cycles if the diagonal is initialized to ∞ and not to zero. If any of the diagonal values are changed, then the graph contains a cycle. Also, if an initial value of ∞ between two vertices in the matrix is not changed to a finite value, it is an indication that one vertex cannot be reached from another.

The simplicity of the algorithm is reflected in the ease with which its complexity can be computed: Since all three `for` loops are executed $|V|$ times, its complexity is $|V|^3$. This is a good efficiency for dense, nearly complete graphs, but in sparse graphs, there is no need to check for all possible connections between vertices. For sparse graphs, it may be more beneficial to use a one-to-all method $|V|$ times, that is, apply it to each vertex separately. This should be a label-setting algorithm, which as a rule has better complexity than a label-correcting algorithm. However, a label-setting algorithm cannot work with graphs with negative weights. To solve this problem, we have to modify the graph so that it does not have negative weights and it guarantees to have the same shortest paths as the original graph. Fortunately, such a modification is possible (Edmonds and Karp 1972).

Observe first that, for any vertex v, the length of the shortest path to v is never greater than the length of the shortest path to any of its predecessors w plus the length of edge from w to v, or

$$dist(v) \leq dist(w) + weight(edge(wv))$$

for any vertices v and w. This inequality is equivalent to the inequality

$$0 \leq weight'(edge(wv)) = weight(edge(vw)) + dist(w) - dist(v)$$

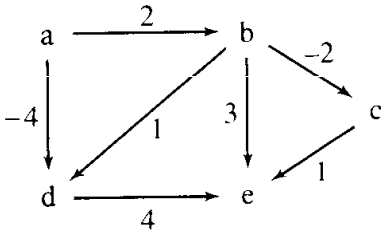
Hence, changing $weight(e)$ to $weight'(e)$ for all edges e renders a graph with nonnegative edge weights. Now note that the shortest path $v_1, v_2, \ldots, v_k$ is

$$\sum_{i=1}^{k-1} weight'\left(edge\left(v_i v_{i+1}\right)\right) = \left(\sum_{i=1}^{k-1} weight\left(edge\left(v_i v_{i+1}\right)\right)\right) + dist\left(v_1\right) - dist\left(v_k\right)$$

Therefore, if the length L' of the path from v_1 to v_k is found in terms of nonnegative weights, then the length L of the same path in the same graph using the original weights, some possibly negative, is $L = L' - dist(v_1) + dist(v_k)$.

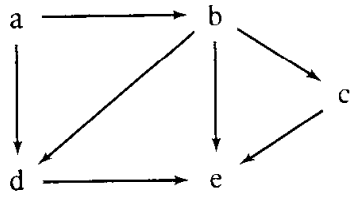
But because the shortest paths have to be known to make such a transformation, the graph has to be preprocessed by one application of a label-correcting method. Only afterward are the weights modified, and then a label-setting method is applied $|V|$ times.

FIGURE **8.11** An execution of `WFIalgorithm()`.



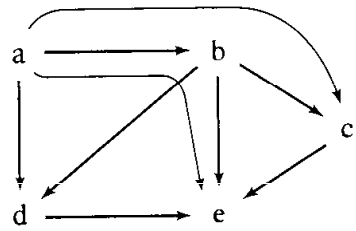
(a)

	a	b	c	d	e
[a]	0	2	∞	−3	∞
b	∞	0	−2	1	3
c	∞	∞	0	∞	1
d	∞	∞	∞	0	4
e	∞	∞	∞	∞	0



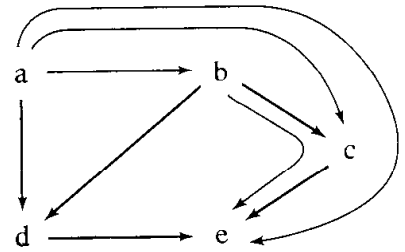
(b)

	a	b	c	d	e
a	0	2	0	−4	5
[b]	∞	0	−2	1	3
c	∞	∞	0	∞	1
d	∞	∞	∞	0	9
e	∞	∞	∞	∞	0



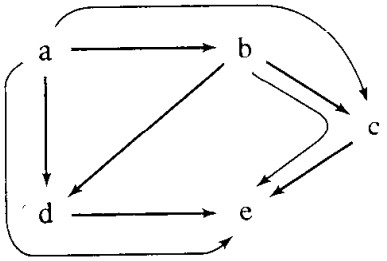
(c)

	a	b	c	d	e
a	0	2	0	−4	1
b	∞	0	−2	1	−1
[c]	∞	∞	0	∞	1
d	∞	∞	∞	0	4
e	∞	∞	∞	∞	0



(d)

	a	b	c	d	e
a	0	2	0	−4	0
b	∞	0	−2	1	−1
c	∞	∞	0	∞	1
[d]	∞	∞	∞	0	4
e	∞	∞	∞	∞	0



(e)

8.4 Cycle Detection

Many algorithms rely on detecting cycles in graphs. We have just seen that, as a side effect, `WFIalgorithm()` allows for detecting cycles in graphs. However, it is a cubic algorithm, which in many situations is too inefficient. Therefore, other cycle detection methods have to be explored.

One such algorithm is obtained directly from `depthFirstSearch()`. For undirected graphs, it is enough to add only one line in `DFS(v)` to detect cycles, which is an `else` statement as in

```
cycleDetectionDFS(v)
   num(v) = i++;
   for all vertices u adjacent to v
      if num(u) is 0
         attach edge(uv) to edges;
         cycleDetectionDFS(u);
      else if edge(vu) is not in edges
         cycle detected;
```

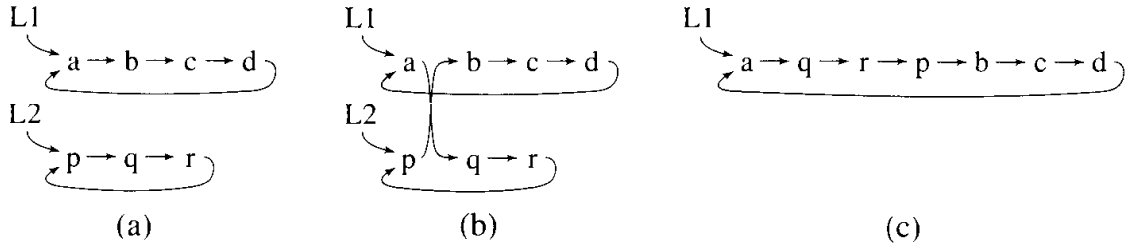
For digraphs, the situation is a bit more complicated, since there may be edges between different spanning subtrees, called *side edges* (see *edge*(*ga*) in Figure 8.4b). An edge (a back edge) indicates a cycle if it joins two vertices already included in the same spanning subtree. To consider only this case, a number higher than any number generated in subsequent searches is assigned to a vertex being currently visited after all its descendants have also been visited. In this way, if a vertex is about to be joined by an edge with a vertex having a lower number, we declare a cycle detection. The algorithm is now

```
digraphCycleDetectionDFS(v)
   num(v) = i++:
   for all vertices u adjacent to v
      if num(u) is 0
         attach edge(uv) to edges;
         digraphCycleDetectionDFS(u);
      else if num(u) is not ∞
         cycle detected;
   num(v) = ∞;
```

8.4.1 Union-Find Problem

Let us recall from a preceding section that depth-first search guaranteed generating a spanning tree in which no element of `edges` used by `depthFirstSearch()` led to a cycle with other elements of `edges`. This was due to the fact that if vertices v and u belonged to `edges`, then the *edge*(vu) was disregarded by `depthFirstSearch()`. A problem arises when `depthFirstSearch()` is modified so that it can detect whether a specific *edge*(vu) is part of a cycle (see Exercise 20). Should such a modified depth-first search be applied to each edge separately, then the total run would be

FIGURE 8.12 Concatenating two circular linked lists.



$O(|E|(|E| + |V|))$, which could turn into $O(|V|^4)$ for dense graphs. Hence, a better method needs to be found.

The task is to determine if two vertices are in the same set. Two operations are needed to implement this task: finding the set to which a vertex v belongs and uniting two sets into one if vertex v belongs to one of them and w to another. This is known as the *union-find problem.*

The sets used to solve the union-find problem are implemented with circular linked lists; each list is identified by a vertex that is the root of the three to which the vertices in the list belong. But first, all vertices are numbered with integers $0, \ldots, |V| - 1$, which are used as indexes in three arrays: `root[ ]` to store a vertex index identifying a set of vertices, `next[ ]` to indicate the next vertex on a list, and `length[ ]` to indicate the number of vertices in a list.

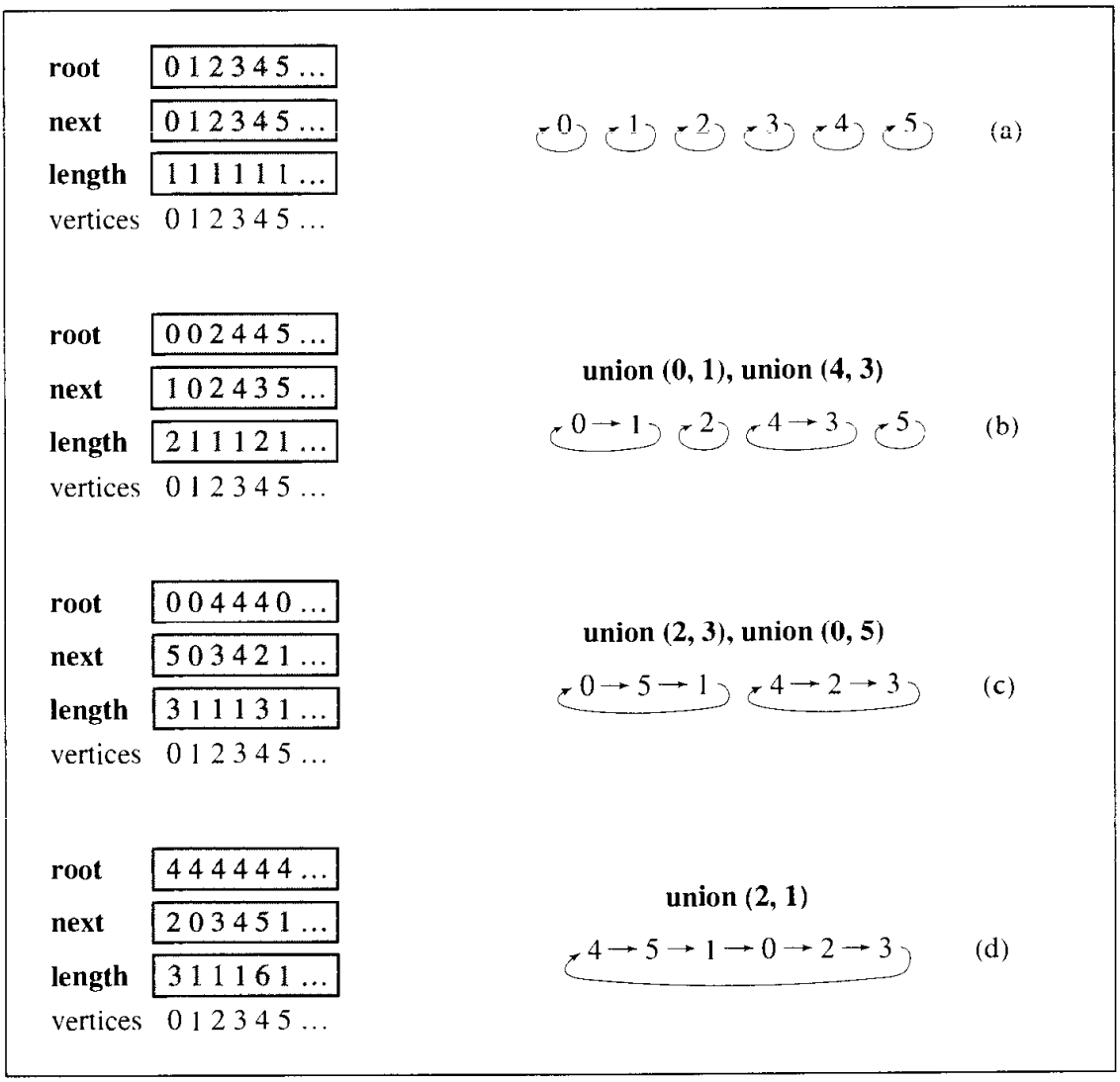
We use circular lists to be able to combine two lists right away, as illustrated in Figure 8.12. Lists L1 and L2 (Figure 8.12a) are merged into one by interchanging `next` pointers in both lists (Figure 8.12b or, the same list, Figure 8.12c). However, the vertices in L2 have to " know" to which list they belong; therefore, their root indicators have to be changed to the new root. Since it has to be done for all vertices of list L2, then L2 should be the shorter of the two lists. To determine the length of lists, the third array is used, `length[ ]`, but only lengths for the identifying nodes (roots) have to be updated. Therefore, the lengths indicated for other vertices that were roots (and at the beginning each of them was) are disregarded.

The union operation performs all the necessary tasks, so the find operation becomes trivial. By constantly updating the array `root[ ]`, the set, to which a vertex j belongs, can be immediately identified, since it is a set whose identifying vertex is `root[j]`. Now, after the necessary initializations,

```
initialize( )
   for i = 0 to |V| – 1
      root[i] = next[i] = i;
      length[i] = 1;
```

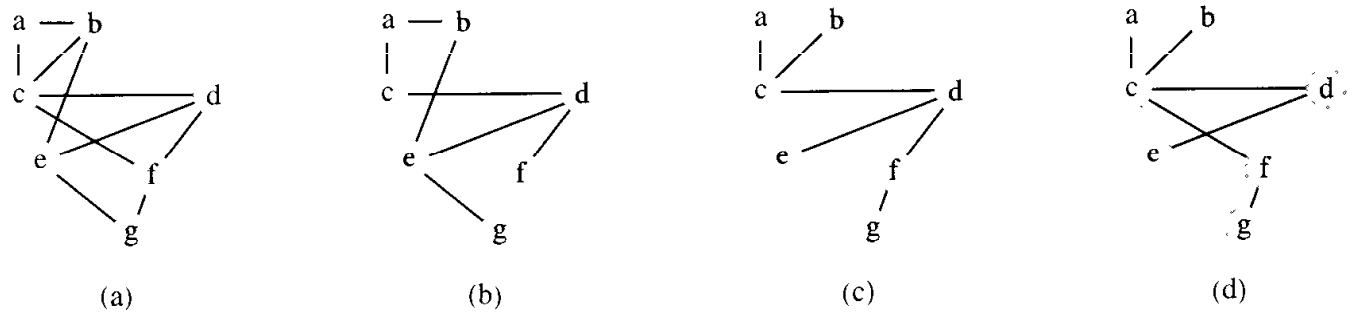
`union( )` can be defined as follows:

FIGURE 8.13 An example of application of `union()` to merge lists.




```
union(edge(vu))
   if (root[u] == root[v])                          // disregard this edge,
      return;                                       // since v and u are in
   else if (length[root[v]] < length[root[u]])      // the same set; combine
      rt = root[v];                                 // two sets into one;
      length[root[u]] += length[rt];
      root[rt] = root[u];                           // update root of root and
      for (j = next[rt]; j != rt; j = next[j])      // then other vertices
         root[j] = root[u];                         // in circular list;
      swap(next[rt],next[root[u]]);                 // merge two lists;
      add edge(vu) to spanningTree;
   else  // if length[root[v]] >= length[root[u]]
         // proceed as before, with v and u reversed;
```

FIGURE 8.14 A graph representing (a) the airline connections between seven cities and (b–d) three possible sets of connections.



An example of the application of `union()` to merge lists is shown in Figure 8.13. After initialization, there are $|V|$ unary sets or one-node linked lists, as in Figure 8.13a. After executing `union()` several times, smaller linked lists are merged into larger ones, and each time, the new situation is reflected in the three arrays, as shown in Figures 8.13b–c.

The complexity of `union()` depends on the number of vertices that have to be updated when merging two lists, specifically, on the number of vertices on the shorter list, since this number determines how many times the `for` loop in `union()` iterates. Since this number can be between 1 and $|V|/2$, the complexity of `union()` is given by $O(|V|)$.

8.5 SPANNING TREES

Consider the graph representing the airline connections between seven cities (Figure 8.14a). If the economic situation forces this airline to shut down as many connections as possible, which of them should be retained to make sure that it is still possible to reach any city from any other city, if only indirectly? One possibility is the graph in Figure 8.14b. City *a* can be reached from city *d* using the path *d, c, a,* but it is also possible to use the path *d, e, b, a.* Since the number of retained connections is the issue, there is still the possibility we can reduce this number. It should be clear that the minimum number of such connections form a tree because alternate paths arise as a result of cycles in the graph. Hence, to create the minimum number of connections, a spanning tree should be created, and such a spanning tree is the byproduct of `depthFirstSearch()`. Clearly, we can create different spanning trees (Figures 8.14c–d), that is, decide to retain different sets of connections, but all these trees have six edges and we cannot do any better than that.

The solution to this problem is not optimal in that the distances between cities have not been taken into account. Since there are alternative six-edge connections between cities, the airline uses the cost of these connections to choose the best, guaranteeing the

optimum cost. This can be achieved by having maximally short distances for the six connections. This problem can now be phrased as finding a *minimum spanning tree*, which is a spanning tree in which the sum of the weights of its edges is minimal. The previous problem of finding a spanning tree in a simple graph is a case of the minimum spanning tree problem in that the weights for each edge are assumed to equal one. Therefore, each spanning tree is a minimum tree in a simple graph.

The minimum spanning tree problem has many solutions and only a handful of them are presented here. (For a review of these methods, see Graham and Hell 1985. These algorithms can be divided in the following categories:

1. Creating and expanding at the same time many trees to be merged into larger trees (Borůvka's algorithm).
2. Expanding a set of trees to form one spanning tree (Kruskal's algorithm).
3. Creating and expanding only one tree by adding new branches to it (Jarník-Prim's algorithm).
4. Creating and expanding only one tree by adding new branches to it and possibly removing branches from it (Dijkstra's method).

8.5.1 Borůvka's Algorithm

Probably the first algorithm for finding the minimum spanning tree was devised in 1926 by Otakar Borůvka (pronounced: boh-roof-ka). In this method, we start with $|V|$ one-vertex trees, and for each vertex v, we look for an *edge*(vw) of minimum weight among all edges outgoing from v and create small trees by including these edges. Then, we look for edges of minimal weight that can connect the resulting trees to larger trees. The process is finished when one tree is created. Here is a pseudocode for this algorithm:

```
BorůvkaAlgorithnm( weighted connected undirected graph)
    make each vertex the root of a one-node tree;
    while there is more than one tree
        for each tree t
            e = minimum weight edge (vu) where v is included in t and u is not;
            create a tree by combining t and the tree that includes u
                if such a tree does not exist yet;
```

For example, for the graph in Figure 8.15a, out of seven one-vertex trees, two trees are created because, for vertices a and c, *edge*(ac) is chosen, for vertex b, *edge*$(ab$ is chosen, for vertex d, *edge*(df) is chosen, for vertex e, *edge*(eg) is chosen, and for vertices f and g, *edge*(fg) is chosen (Figure 8.15b). Afterward, for the tree(abc) and the tree$(defg)$, *edge*(cf) is selected, since it is the shortest edge that connects these two trees, resulting in one spanning tree.

How many iterations are required? In each iteration of the `while` loop, each of the existing r trees is joined with an edge to at least one tree. In the worst case, $r/2$ trees are generated; in the best case, one tree is generated. In the subsequent iteration, there are $r/4$ trees and so on. In the worst case, $\lg|V|$ iterations are needed, where $|V|$ is the initial number of one-vertex trees.

Borůvka's method lends itself very nicely to parallel processing, since for each tree, a minimal edge has to be found independently.

8.5.2 Kruskal's Algorithm

One popular algorithm was devised by Joseph Kruskal. In this method, all edges are ordered by weight, and then each edge in this ordered sequence is checked to see whether it can be considered part of the tree under construction. It is added to the tree if no cycle arises after its inclusion. This simple algorithm can be summarized as follows:

```
KruskalAlgorithm(weighted connected undirected graph)
   tree = null;
   edges = sequence of all edges of graph sorted by weight;
   for (i = 1; i ≤ |E| and |tree| < |V| − 1; i++)
      if e_i from edges does not form a cycle with edges in tree
         add e_i to tree;
```

Figures 8.15ca–cf contain a step-by-step example of Kruskal's algorithm.

The complexity of this algorithm is determined by the complexity of the sorting method applied, which for an efficient sorting is $O(|E| \lg |E|)$. It also depends on the complexity of the method used for cycle detection. If we use `union()` to implement Kruskal's algorithm, then the `for` loop of `KruskalAlgorithm()` becomes

```
for (i = 1; i≤|E| and |tree| < |V| − 1; i++)
   union(e_i = edge(vu));
```

Although `union()` can be called up to $|E|$ times, it is exited after one (the first) test if a cycle is detected and it performs a union, which is of complexity $O(|V|)$, only for $|V| - 1$ edges added to `tree`. Hence, the complexity of `KruskalAlgorithm()`'s `for` loop is $O(|E| + (|V| - 1)|V|)$, which is $O(|V|^2)$. Therefore, the complexity of `KruskalAlgorithm()` is determined by the complexity of a sorting algorithm which is $O(|E|\lg|E|)$, that is, $O(|E|\lg|V|)$.

8.5.3 Jarník-Prim's Algorithm

Another algorithm was discovered by Vojtech Jarník (pronounced: yar-neek) in 1936 and later rediscovered by Robert Prim. In this method, all of the edges are also initially ordered, but a candidate for inclusion in the spanning tree is an edge which not only does not lead to cycles in the tree, but also is incident to a vertex already in the tree:

```
JarnikPrimAlgorithm(weighted connected undirected graph)
   tree = null;
   edges = sequence of all edges of graph sorted by weight;
   for i = 1 to |V| − 1
      for j = 1 to |edges|
         if e_j from edges does not form a cycle with edges in tree and
            is incident to a vertex in tree
               add e_j to tree;
               break;
```

FIGURE 8.15 A spanning tree of graph (a) found (ba–bb) with Borůvka's algorithm, (ca–cf) with Kruskal's algorithm, (da–df) with Jarník-Prim's algorithm, (ea–el) and with Dijkstra's method.

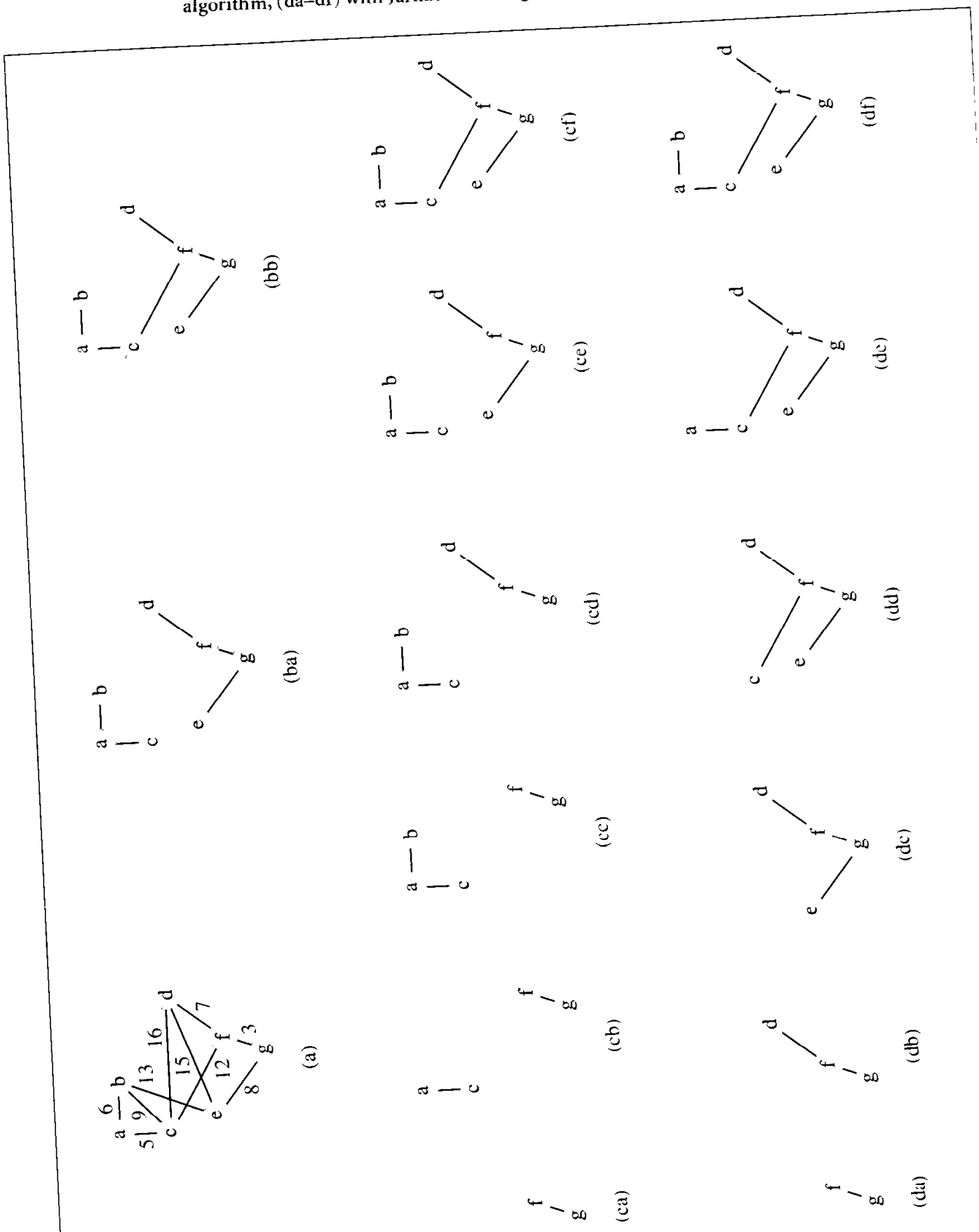
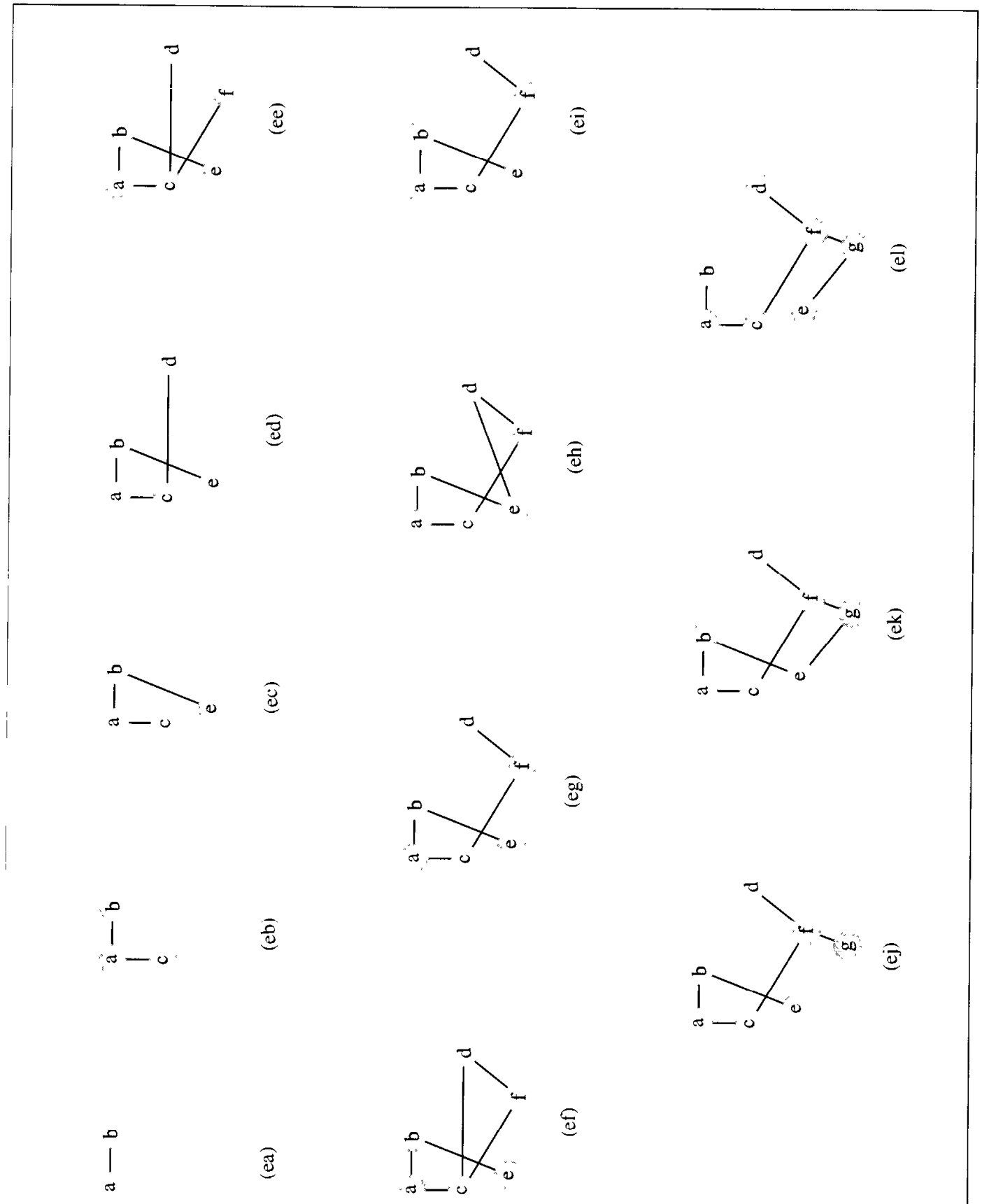


FIGURE 8.15 *(continued)*



Figures 8.15da–df contain a step-by-step example of Jarník-Prim's algorithm. The spanning tree resulting from this algorithm is the same as that given by the Kruskal algorithm; however, the order in which edges have been added to the tree is different. The inner loop of `JarnikPrimAlgorithm()` can be $O(E)$ in the worst case, and since the outer loop iterates $|V| - 1$ times, the inner loop may iterate $O(|V||E|)$ times in total. However, this complexity can be substantially improved by a careful implementation of `edges`.

The difference between the Kruskal algorithm and the Jarník-Prim algorithm is that the latter always keeps the tree being constructed in one piece so that it is a tree at all stages of application of this algorithm. The Kruskal algorithm is more concerned about the outcome, so it considers it irrelevant that in the middle of its execution the spanning tree may not be a tree at all, but at best a collection of trees. Nevertheless, the Kruskal algorithm guarantees that, at the end, there is only one spanning tree. Therefore, the Jarník-Prim algorithm may be considered more elegant, as we see a tree being expanded at all times. The price for this elegance is that the only edges that can be added to the tree are the ones which are not isolated from the tree built so far, so that certain edges may need to be reconsidered several times. In the Kruskal algorithm, each edge needs to be considered only once, since if it leads to a cycle at one stage, it all the more would lead to a cycle at a later stage, and hence, it does not have to be reconsidered anymore. Thus, the Kruskal algorithm is faster.

8.5.4 Dijkstra's Method

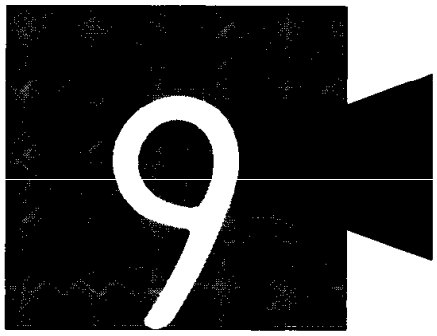
Kruskal's and Jarník-Prim's algorithms require that all the edges be ordered before beginning to build the spanning tree. This, however, is not necessary; it is possible to build a spanning tree by using any order of edges. A method was proposed by Dijkstra (1960) and independently by Robert Kalaba, and because no particular order of edges is required here, their method is more general than the other two.

```
DijkstraMethod(weighted connected undirected graph)
   tree = null;
   edges = an unsorted sequence of all edges of graph;
   for j = 1 to |E|
      add e_j to tree;
      if there is a cycle in tree
         remove an edge with maximum weight from this only cycle;
```

In this algorithm, the tree is being expanded by adding to it edges one by one, and if a cycle is detected, then an edge in this cycle with maximum weight is discarded. An example of building the minimum spanning tree with this method is shown in Figures 8.15ea–el.

8.6 Connectivity

In many problems, we are interested in finding a path in the graph from one vertex to any other vertex. For undirected graphs, this means that there are no separate pieces,



Sorting

The efficiency of data handling can often be substantially increased if the data are sorted according to some criteria of order. For example, it would be practically impossible to find a name in the telephone directory if the names were not alphabetically ordered. The same can be said about dictionaries, book indexes, payrolls, bank accounts, student lists, and other alphabetically organized materials. The convenience of using sorted data is unquestionable and must be addressed in computer science as well. Although a computer can grapple with an unordered telephone book more easily and quickly than a human, it is extremely inefficient to have the computer process such an unordered data set. It is often necessary to sort data before processing.

The first step is to choose the criteria which will be used to order data. This choice will vary from application to application and must be defined by the user. Very often, the sorting criteria are natural, as in the case of numbers. A set of numbers can be sorted in ascending or descending order. The set of five positive integers (5, 8, 1, 2, 20) can be sorted in ascending order resulting in the set (1, 2, 5, 8, 20) or in descending order resulting in the set (20, 8, 5, 2, 1). Names in the phone book are ordered alphabetically by last name, which is the natural order. For alphabetic and nonalphabetic characters, the ASCII code is commonly used, although other choices such as EBCDIC are possible. Once a criterion is selected, the second step is how to put a set of data in order using that criterion.

The final ordering of data can be obtained in a variety of ways, and only some of them can be considered meaningful and efficient. To decide which method is best, certain criteria of efficiency have to be established and a method for quantitatively comparing different algorithms must be chosen.

To make the comparison machine-independent, certain critical properties of sorting algorithms should be defined when comparing alternative methods. Two such properties are the number of comparisons and the number of data movements. The choice of these two properties should not be surprising. To sort a set of data, the data

have to be compared and moved as necessary; the efficiency of these two operations depends on the size of the data set.

Since determining the precise number of comparisons is not always necessary or possible, an approximate value can be computed. For this reason, the number of comparisons and movements is approximated with big-O notation by giving the order of magnitude of these numbers. But the order of magnitude can vary depending on the initial ordering of data. How much time, for example, does the machine spend on data ordering if the data are already ordered? Does it recognize this initial ordering immediately or is it completely unaware of that fact? Hence, the efficiency measure also indicates the "intelligence" of the algorithm. For this reason, the number of comparisons and movements is computed (if possible) for the following three cases: best case (often, data already in order), worst case (usually, data in reverse order), and average case (data in random order). Some sorting methods perform the same operations regardless of the initial ordering of data. It is easy to measure the performance of such algorithms, but the performance itself is usually not very good. Many other methods are more flexible and their performance measures for all three cases differ.

The number of comparisons and the number of movements do not have to coincide. An algorithm can be very efficient on the former and perform poorly on the latter, or vice versa. Therefore, practical reasons must aid in the choice of which algorithm to use. For example, if only simple keys are compared, such as integers or characters, then the comparisons are relatively fast and inexpensive. If strings or arrays of numbers are compared, then the cost of comparisons goes up substantially, and the weight of the comparison measure becomes more important. If, on the other hand, the data items moved are large, such as structures, then the movement measure may stand out as the determining factor in efficiency considerations. All theoretically established measures have to be used with discretion, and theoretical considerations should be balanced with practical applications. After all, the practical applications serve as a rubber stamp for theory decisions.

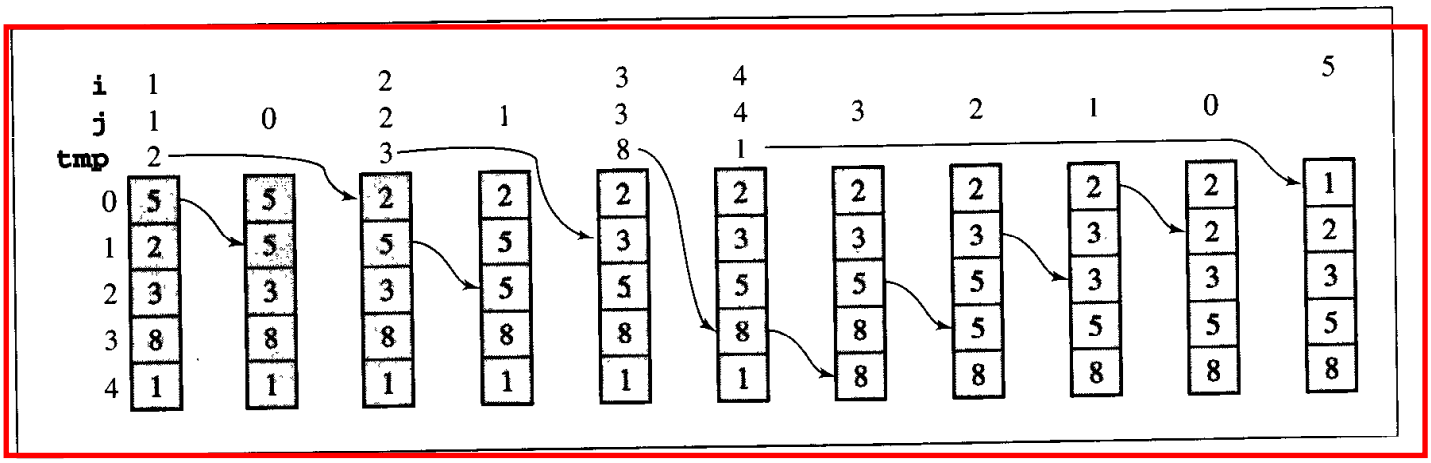
Sorting algorithms, whose number can be counted in the dozens, are of different levels of complexity. A simple method can be only 20% less efficient than a more elaborate one. If sorting is used in the program once in a while and only for small sets of data, then using a sophisticated and slightly more efficient algorithm may not be desirable; the same operation can be performed using a simpler method and simpler code. But if thousands of items are to be sorted, then a gain of 20% must not be neglected. Simple algorithms often perform better with a small amount of data than their more complex counterparts whose effectiveness may only become obvious when data samples become very large.

9.1 Elementary Sorting Algorithms

9.1.1 Insertion Sort

An *insertion sort* starts by considering the two first elements of the array `data`, which are `data[0]` and `data[1]`. If they are out of order, an interchange takes place. Then,

FIGURE 9.1 The array [5 2 3 8 1] sorted by insertion sort.



the third element, `data[2]`, is considered and inserted into its proper place. If `data[2]` is less than `data[0]` and `data[1]`, these two elements are shifted by one position; `data[0]` is placed at position 1, `data[1]` at position 2, and `data[2]` at position 0. If `data[2]` is less than `data[1]` and not less than `data[0]`, then only `data[1]` is moved to position 2 and its place is taken by `data[2]`. If, finally, `data[2]` is not less than both its predecessors, it stays in its current position. Each element `data[i]` is inserted into its proper location j such that $0 \le j \le i$, and all elements greater than `data[i]` are moved by one position.

An outline of the insertion sort algorithm is as follows:

```
insertionsort(data[],n)
   for(i = 1; i < n; i++)
      move all elements data[j] greater than data[i] by one position;
      place data[i] in its proper position;
```

Note that sorting is restricted only to a fraction of the array in each iteration, and only in the last pass is the whole array considered. Figure 9.1 shows what changes are made to the array [5 2 3 8 1] when `insertionsort()` executes.

Since an array having only one element is already ordered, the algorithm starts sorting from the second position, position 1. Then for each element `tmp = data[i]`, all elements greater than `tmp` are copied to the next position, and `tmp` is put in its proper place.

An implementation of insertion sort is:

```
template<class T>
void insertionsort(T data[], int n) {
    for (int i = 1,j; i < n; i++) {
        T tmp = data[i];
        for (j = i; j > 0 && tmp < data[j-1]; j--)
            data[j] = data[j-1];
        data[j] = tmp;
    }
}
```

An advantage of using insertion sort is that it sorts the array only when it is really necessary. If the array is already in order, no substantial moves are performed; only the variable `tmp` is initialized, and the value stored in it is moved back to the same position. The algorithm recognizes that part of the array is already sorted and stops execution accordingly. But it recognizes only this, and the fact that elements may already be in their proper positions is overlooked. Therefore, they can be moved from these positions and then later moved back. This happens to numbers 2 and 3 in the example in Figure 9.1. Another disadvantage is that if an item is being inserted, all elements greater than the one being inserted have to be moved. Insertion is not localized and may require moving a significant number of elements. Considering that an element can be moved from its final position only to be placed there again later, the number of redundant moves can slow down execution substantially.

To find the number of movements and comparisons performed by `insertionsort()`, observe first that the outer `for` loop always performs $n-1$ iterations. However, the number of elements greater than `data[i]` to be moved by one position is not always the same.

The best case is when the data are already in order. Only one comparison is made for each position i, so there are $n-1$ comparisons, which is $O(n)$, and $2(n-1)$ moves, all of them redundant.

The worst case is when the data are in reverse order. In this case, for each i, the item `data[i]` is less than every item `data[0]`, . . . , `data[i-1]`, and each of them is moved by one position. For each iteration i of the outer `for` loop, there are i comparisons and, the total number of comparisons for all iterations of this loop is

$$\sum_{i=1}^{n-1} i = 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

The number of times the assignment in the inner `for` loop is executed can be computed using the same formula. The number of times `tmp` is loaded and unloaded in the outer `for` loop is added to that, resulting in the total number of moves:

$$\frac{n(n-1)}{2} + 2(n-1) = \frac{n^2+3n-4}{2} = O(n^2)$$

Only extreme cases have been taken into consideration. What happens if the data are in random order? Is the sorting time closer to the time of the best case, $O(n)$, or to the worst case, $O(n^2)$? Or is it somewhere in between? The answer is not immediately evident and requires certain introductory computations. The outer `for` loop always executes $n-1$ times, but it is also necessary to determine the number of iterations for the inner loop.

For every iteration i of the outer `for` loop, the number of comparisons depends on how far away the item `data[i]` is from its proper position in the currently sorted subarray `data[0 . . . i]`. If it is already in this position, only one test is performed that compares `data[i]`and `data[i-1]`. If it is one position away from its proper place, two comparisons are performed: `data[i]` is compared with `data[i-1]` and then with `data[i-2]`. Generally, if it is j positions away from its proper location, `data[i]` is compared with $j+1$ other elements. This means that, in iteration i of the outer `for` loop, there are either 1, 2, . . . or i comparisons.

Under the assumption of equal probability of occupying array cells, the average number of comparisons of `data[i]` with other elements during the iteration i of the outer `for` loop can be computed by adding all the possible numbers of times such tests are performed and dividing the sum by the number of such possibilities. The result is

$$\frac{1+2+\ldots+i}{i}=\frac{\frac{1}{2}i(i+1)}{i}=\frac{i+1}{2}$$

To obtain the average number of all comparisons, the computed figure has to be added for all i's (for all iterations of the outer `for` loop) from 1 to $n-1$. The result is

$$\sum_{i=1}^{n-1}\frac{i+1}{2}=\frac{1}{2}\sum_{i=1}^{n-1}i+\sum_{i=1}^{n-1}\frac{1}{2}=\frac{\frac{1}{2}n(n-1)}{2}+\frac{1}{2}(n-1)=\frac{n^2+n-2}{4}$$

which is $O(n^2)$ and approximately one-half of the number of comparisons in the worst case.

By similar reasoning, we can establish that, in iteration i of the outer `for` loop, `data[i]` can be moved either 0, 1, . . . , or $i-1$ times; that is

$$\frac{0+1+\ldots+(i-1)}{i}=\frac{\frac{1}{2}i(i-1)}{i}=\frac{i-1}{2}$$

times plus two unconditional movements (to `tmp` and from `tmp`). Hence, in all the iterations of the outer `for` loop we have, on the average,

$$\sum_{i=1}^{n-1}\left(\frac{i-1}{2}+2\right)=\frac{1}{2}\sum_{i=1}^{n-1}i+\sum_{i=1}^{n-1}\frac{3}{2}=\frac{\frac{1}{2}n(n-1)}{2}+\frac{3}{2}(n-1)=\frac{n^2+5n-6}{4}$$

movements, which is also $O(n^2)$.

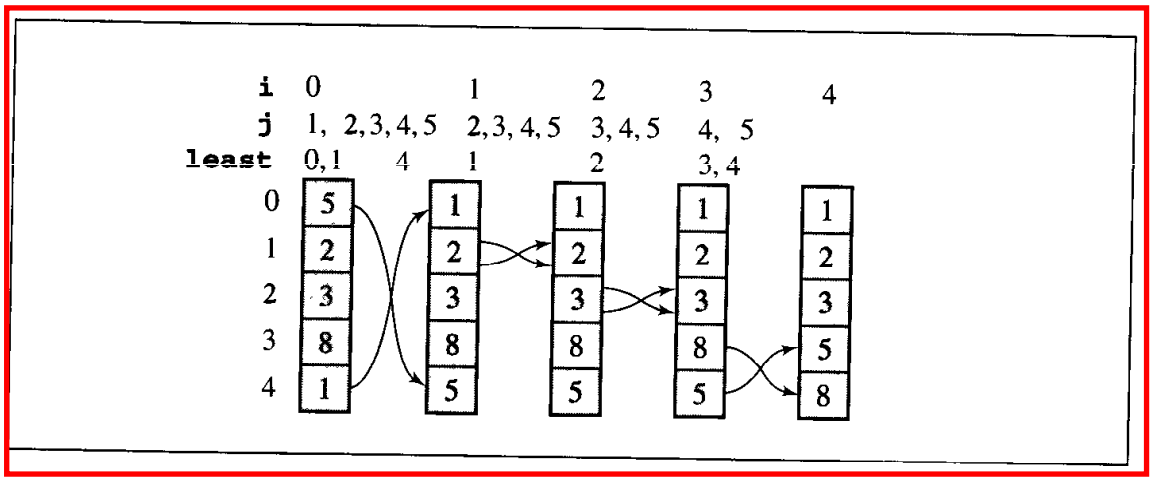
This answers the question: Is the number of movements and comparisons for a randomly ordered array closer to the best or to the worst case? Unfortunately, it is closer to the latter, which means that, on the average, when the size of an array is doubled, the sorting effort quadruples.

9.1.2 Selection Sort

Selection sort is an attempt to localize the exchanges of array elements by finding a misplaced element first and putting it in its final place. The element with the lowest value is selected and exchanged with the element in the first position. Then, the smallest value among the remaining elements `data[1],...,data[n-1]` is found and put in the second position. This selection and placement by finding, in each pass `i`, the lowest value among the elements `data[i],...,data[n-1]` and swapping it with `data[i]` are continued until all elements are in their proper positions. The following pseudocode reflects the simplicity of the algorithm:

```
selectionsort(data[],n)
   for(i = 0; i < n-1; i++)
      select the smallest element among data[i],...,data[n-1];
      swap it with data[i];
```

FIGURE 9.2. The array [5 2 3 8 1] sorted by selection sort.



It is rather obvious that `n-2` should be the last value for `i`, since if all elements but the last have been already considered and placed in their proper position, then the nth element (occupying position `n-1`) has to be the largest. An example is shown in Figure 9.2. Here is a C++ implementation of selection sort:

```
template<class T>
void selectionsort(T data[], int n) {
    for (int i = 0,j,least; i < n-1; i++) {
        for (j = i+1, least = i; j < n; j++)
            if (data[j] < data[least])
                least = j;
        swap(data[least],data[i]);
    }
}
```

where the function `swap()` exchanges elements `data[least]` and `data[i]` (see the end of Section 1.2). Note that `least` is not the smallest element but its position.

The analysis of the performance of the function selectionsort() is simplified by the presence of two for loops with lower and upper bounds. The outer loop executes n – 1 times, and for each i between 0 and n – 2, the inner loop iterates j = (n - 1) - i times. Because comparisons of keys are done in the inner loop, there are

$$\sum_{i=0}^{n-2}(n-1-i) = (n-1) + \cdots + 1 = \frac{n(n-1)}{2} = O(n^2)$$

comparisons. This number stays the same for all cases. There can be some savings only in the number of swaps. Note that if the assignment in the `if` statement is executed, only the index `j` is moved, not the item located currently at position `j`. Array elements are swapped unconditionally in the outer loop as many times as this loop executes, which is n – 1. Thus, in all cases, items are moved the same number of times, 3 · (n – 1).

The best thing about this sort is the required number of assignments, which can hardly be beaten by any other algorithm. However, it might seem somewhat unsatisfactory

that the total number of exchanges, $3 \cdot (n-1)$, is the same for all cases. Obviously, no exchange is needed if an item is in its final position. The algorithm disregards that and swaps such an item with itself making three redundant moves. The problem can be alleviated by making `swap()` a conditional operation. The condition preceding the `swap()` should indicate that no item less than `data[least]` has been found among elements `data[i+1]`, ...,`data[n-1]`. The last line of `selectionsort()` might be replaced by the lines:

```
if (data[i] != data[least])
   swap (data[least], data[i]);
```

This increases the number of array element comparisons by `n` – 1, but this increase can be avoided by noting that there is no need to compare items. We proceed as we did in the case of the `if` statement of `selectionsort()` by comparing the indexes and not the items. The last line of `selectionsort()` can be replaced by:

```
if (i != least)
   swap (data[least], data[i]);
```

Is such an improvement worth the price of introducing a new condition in the procedure and adding $n-1$ index comparisons as a consequence? It depends on what types of elements are being sorted. If the elements are numbers or characters, then interposing a new condition to avoid execution of redundant swaps gains little in efficiency. But if the elements in `data` are large compound entities such as arrays or structures, then one swap (which requires three assignments) may take the same amount of time as, say, 100 index comparisons, and using a conditional `swap()` is recommended.

9.1.3 Bubble Sort

A bubble sort can be best understood if the array to be sorted is envisaged as a vertical column whose smallest elements are at the top and whose largest elements are at the bottom. The array is scanned from the bottom up, and two adjacent elements are interchanged if they are found to be out of order with respect to each other. First, items `data[n-1]` and `data[n-2]` are compared and swapped if they are out of order. Next, `data[n-2]` and `data[n-3]` are compared, and their order is changed if necessary and so on up to `data[1]` and `data[0]`. In this way, the smallest element is bubbled up to the top of the array.

However, this is only the first pass through the array. The array is scanned again comparing consecutive items and interchanging them when needed, but this time, the last comparison is done for `data[2]` and `data[1]` because the smallest element is already in its proper position, namely, position 0. The second pass bubbles the second smallest element of the array up to the second position, position 1. The procedure continues until the last pass when only one comparison, `data[n-1]` with `data[n-2]`, and possibly one interchange are performed.

A pseudocode of the algorithm is as follows:

```
bubblesort(data[],n)
    for (i = 0; i < n-1; i++)
        for (j = n-1; j > i; --j)
            swap elements in position j and j-1 if they are out of order;
```

FIGURE 9.3 The array [5 2 3 8 1] sorted by bubble sort.

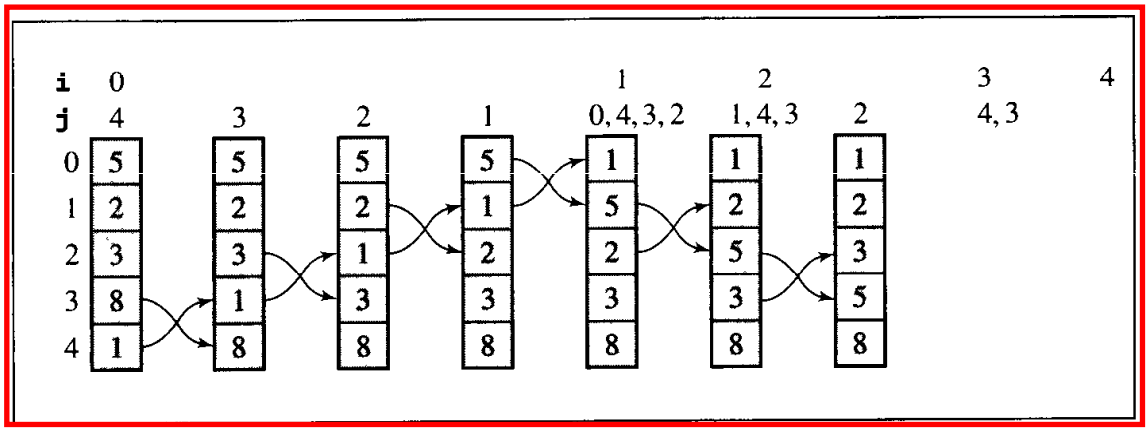


Figure 9.3. illustrates the changes performed in the integer array [5 2 3 8 1] during the execution of `bubblesort()`. Here is an implementation of bubble sort:

```
template<class T>
void bubblesort(T data[], int n) {
    for (int i = 0; i < n-1; i++)
        for (int j = n-1; j > i; --j)
            if (data[j] < data[j-1])
                swap(data[j],data[j-1]);
}
```

The number of comparisons is the same in each case (best, average, and worst) and equals the total number of iterations of the inner `for` loop:

$$\sum_{i=0}^{n-2}(n-1-i)=\frac{n(n-1)}{2}=O(n^2)$$

comparisons. This formula also computes the number of swaps in the worst case when the array is in reverse order. In this case, 3 $\frac{n(n-1)}{2}$ moves have to be made.

The best case, when all elements are already ordered, requires no swaps. If an i-cell array is in random order, then the number of swaps can be any number between zero and $i-1$; that is, there can be either no swap at all (all items are in ascending order), one swap, two swaps, . . . or $i-1$ swaps. The array processed by the inner `for` loop is `data[i]`, . . . , `data[n-1]`, and the number of swaps in this subarray—if its elements are randomly ordered—is either zero, one, two, . . . or $n-1-i$. After averaging the sum of all these possible numbers of swaps by the number of these possibilities, the average number of swaps is obtained, which is

$$\frac{0+1+2+\cdots+(n-1-i)}{n-i}=\frac{n-i-1}{2}$$

If all these averages for all the subarrays processed by `bubblesort()` are added (that is, if such figures are summed over all iterations i of the outer `for` loop), the result is

$$\sum_{i=0}^{n-2} \frac{n-i-1}{2} = \frac{1}{2}\sum_{i=0}^{n-2}(n-1) - \frac{1}{2}\sum_{i=0}^{n-2} i$$

$$= \frac{(n-1)^2}{2} - \frac{(n-1)(n-2)}{4} = \frac{n(n-1)}{4}$$

swaps, which is equal to $\frac{3}{4}n(n-1)$ moves.

The main disadvantage of bubble sort is that it still painstakingly bubbles items step by step up toward the top of the array. It looks at two adjacent array elements at a time and swaps them if they are not in order. If an element has to be moved from the bottom to the top, it is exchanged with every element in the array. It does not skip them as selection sort did. In addition, the algorithm concentrates only on the item that is being bubbled up. Therefore, all elements that distort the order are moved, even those that are already in their final positions (see numbers 2 and 3 in Figure 9.3, the situation analogous to that in insertion sort).

What is bubble sort's performance in comparison to insertion and selection sort? In the average case, bubble sort makes approximately twice as many comparisons and the same number of moves as insertion sort, as many comparisons as selection sort, and n times more moves than selection sort.

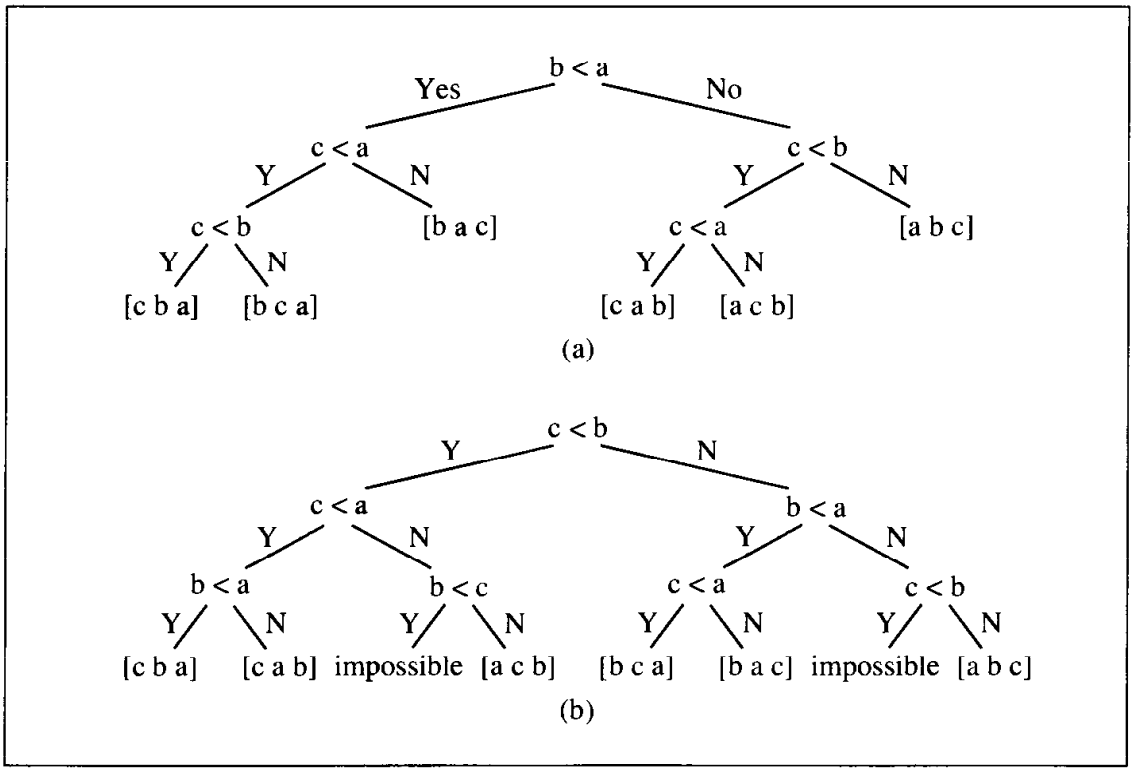
It could be said that insertion sort is twice as fast as bubble sort. In fact it is, but this fact does not immediately follow from the performance estimates. The point is that when determining a formula for the number of comparisons, only comparisons of data items have been included. The actual implementation for each algorithm involves more than just that. In `bubblesort()`, for example, there are two loops, both of which compare indexes: `i` and `n-1` in the first loop, `j` and `i` in the second. All in all, there are $\frac{n(n-1)}{2}$ such comparisons, and this number should not be treated too lightly. It becomes negligible if the data items are large structures. But if `data` consists of integers, then comparing the data takes a similar amount of time as comparing indexes. A more thorough treatment of the problem of efficiency should focus on more than just data comparison and exchange. It should also include the overhead necessary for implementation of the algorithm.

9.2 Decision Trees

The three sorting methods analyzed in previous sections were not very efficient. This leads to several questions: Can any better level of efficiency for a sorting algorithm be expected? Can algorithms, at least theoretically, be more efficient by executing faster? If so, when can we be satisfied with an algorithm and be sure that the sorting speed is unlikely to be increased? We need a quantitative measurement to estimate a *lower bound* of sorting speed.

This section focuses on the comparisons of two elements and not the element interchange. The questions are: On the average, how many comparisons have to be made to sort n elements? Or what is the best estimate of the number of item comparisons if an array is assumed to be ordered randomly?

FIGURE 9.4 Decision trees for (a) insertion sort and (b) bubble sort as applied to the array [a b c].



Every sorting algorithm can be expressed in terms of a binary tree in which the arcs carry the labels Y(es) or N(o). Nonterminal nodes of the tree contain conditions or queries for labels, and the leaves have all possible orderings of the array to which the algorithm is applied. This type of tree is called a *decision tree.* Since the initial ordering cannot be predicted, all possibilities have to be listed in the tree in order for the sorting procedure to grapple with any array and any possible initial order of data. This initial order determines which path is taken by the algorithm and what sequence of comparisons is actually chosen. Note that different trees have to be drawn for arrays of different length.

Figure 9.4 illustrates decision trees for insertion sort and bubble sort for an array [a b c]. The tree for insertion sort has six leaves, and the tree for bubble sort has eight leaves. How many leaves does a tree for an n-element array have? Such an array can be ordered in $n!$ different ways, as many ways as the possible permutations of the array elements, and all of these orderings have to be stored in the leaves of the decision tree. Thus, the tree for insertion sort has six leaves because $n = 3$, and $3! = 6$.

But as the example of the decision tree for bubble sort indicates, the number of leaves does not have to equal $n!$. In fact, it is never less than $n!$, which means that it can be greater than $n!$. This is a consequence of the fact that a decision tree can have leaves corresponding to failures, not only to possible orderings. The failure nodes are reached by an inconsistent sequence of operations. Also, the total number of leaves can be greater than $n!$ because some orderings (permutations) can occur in more than one leaf, since the comparisons may be repeated.

One of the interesting properties of decision trees is the average number of arcs traversed from the root to reach a leaf. Because one arc represents one comparison, the average number of arcs reflects the average number of key comparisons when executing a sorting algorithm.

As already established in Chapter 6, an i-level complete decision tree has 2^{i-1} leaves, $2^{i-1} - 1$ nonterminal nodes (for $i \geq 1$) and $2^i - 1$ total nodes. Because all noncomplete trees with the same number of i levels have fewer nodes than that, $k + m \leq 2^i - 1$, where m is the number of leaves and k the number of nonleaves. Also, $k \leq 2^{i-1} - 1$ and $m \leq 2^{i-1}$ (Section 6.1 and Figure 6.5). The latter inequality is used as an approximation for m. Hence, in an i-level decision tree, there are at most 2^{i-1} leaves.

Now, a question arises: What is a relationship between the number of leaves of a decision tree and the number of all possible orderings of an n-element array? There are $n!$ possible orderings, and each one of them is represented by a leaf in a decision tree. But the tree also has some extra nodes due to repetitions and failures. Therefore, $n! \leq m \leq 2^{i-1}$, or $2^{i-1} \geq n!$. This inequality answers the following question: How many comparisons are performed when using a certain sorting algorithm for an n-element array in the worst case? Or rather, what is the lowest or the best figure expected in the worst case? Note that this analysis pertains to the worst case. We assume that i is a level of a tree regardless of whether or not it is complete; i always refers to the longest path leading from the root of the tree to the lowest tree level, which is also the largest number of comparisons needed to reach an ordered configuration of array stored in the root. First, the inequality $2^{i-1} \geq n!$ is transformed into $i - 1 \geq \lg(n!)$ which means that the path length in a decision tree with at least $n!$ leaves must be at least $\lg(n!)$, or rather, it must be $\lceil \lg(n!) \rceil$, where $\lceil x \rceil$ is an integer not less than x. See the example in Figure 9.5.

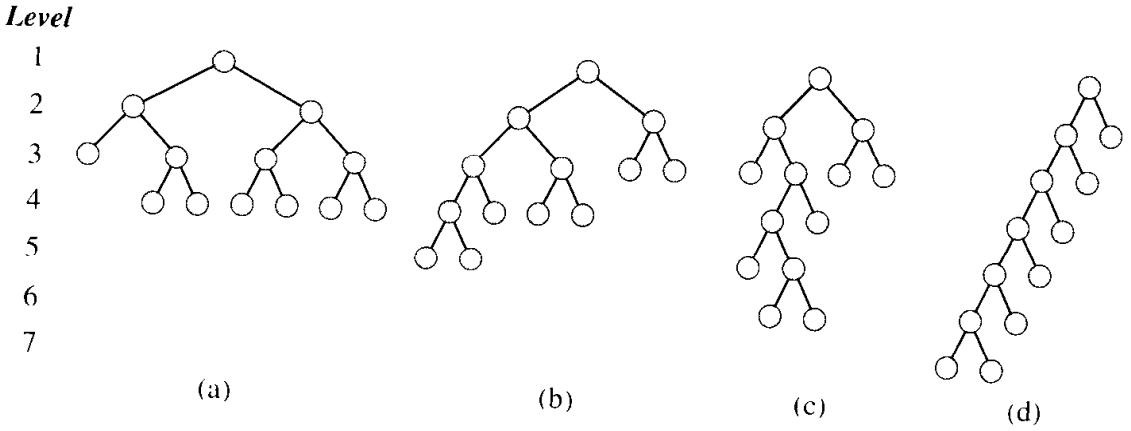
It can be proven that, for a randomly chosen leaf of an m-leaf decision tree, the length of the path from the root to the leaf is not less than $\lg m$ and that, both in the average case and the worst case, the required number of comparisons, $\lg(n!)$, is big-O of $n \lg n$ (see Section A.2 in Appendix A). That is, $O(n \lg n)$ is also the best that can be expected in average cases.

It is interesting to compare this approximation to some of the numbers computed for sorting methods, especially for the average and worst cases. For example, insertion sort requires only $n - 1$ comparisons in the best case, but in the average and the worst cases, this sort turns into an n^2 algorithm since the functions relating the number of comparisons to the number of elements are, for these cases, the big-Os of n^2. This is much greater than $n \lg n$, especially for large numbers. Consequently, insertion sort is not an ideal algorithm. The quest for better methods can be continued with at least the expectation that the number of comparisons should be approximated by $n \lg n$ rather than by n^2.

The difference between these two functions is best seen in Figure 9.6 if the performance of the algorithms analyzed so far is compared with the expected performance $n \lg n$ in the average case. The numbers in the table in Figure 9.6 show that if 100 items are sorted, the desired algorithm is four times faster than insertion sort and eight times faster than selection sort and bubble sort. For 1000 items, it is 25 and 50 times faster. For 10,000, the difference in performance differs by factors of 188 and 376, respectively. This can only serve to encourage the search for an algorithm embodying the performance of the function $n \lg n$.

FIGURE 9.5 Examples of decision trees for an array of three elements.

These are some possible decision trees for an array of three elements. These trees must have at least $3! = 6$ leaves. For the sake of the example, it is assumed that each tree has one extra leaf (a repetition or a failure). In the worst and average cases, the number of comparisons is $i - 1 \geq \lceil \lg(n!) \rceil$. In this example, $n = 3$, so $i - 1 \geq \lceil \lg 3! \rceil = \lceil \lg 6 \rceil \approx \lceil 2.59 \rceil = 3$. And, in fact, only for the best balanced tree (a), the nonrounded length of the average path is less than three.



These are the sums of the lengths of the paths from the root to all leaves in trees (a) – (d) and the average path lengths:

(a) $2 + 3 + 3 + 3 + 3 + 3 + 3 = 20$; average $= \frac{20}{7} \approx 2.86$

(b) $4 + 4 + 3 + 3 + 3 + 2 + 2 = 21$; average $= \frac{21}{7} = 3$

(c) $2 + 4 + 5 + 5 + 3 + 2 + 2 = 23$; average $= \frac{23}{7} \approx 3.29$

(d) $6 + 6 + 5 + 4 + 3 + 2 + 1 = 27$; average $= \frac{27}{7} \approx 3.86$

9.3 Efficient Sorting Algorithms

9.3.1 Shell Sort

The $O(n^2)$ limit for a sorting method is much too large and must be broken to improve efficiency and decrease run time. How can this be done? The problem is that the time required for ordering an array by the three sorting algorithms usually grows faster than the size of the array. In fact, it is customarily a quadratic function of that

FIGURE 9.6 Number of comparisons performed by the simple sorting method and by an algorithm whose efficiency is estimated by the function $n \lg n$.

sort type	n	100	1,000	10,000
insertion	$\frac{n(n-1)}{4}$	2,475	249,750	24,997,500
selection, bubble	$\frac{n(n-1)}{2}$	4,950	499,500	49,995,000
expected	$n \lg n$	664	9,966	132,877

size. It may turn out to be more efficient to sort parts of the original array first and then, if they are at least partially ordered, to sort the entire array. If the subarrays are already sorted, we are that much closer to the best case of an ordered array than initially. A general outline of such a procedure is as follows:

```
divide data into h subarrays;
for (i = 1; i <= h; i++)
    sort subarray data_i;
sort array data;
```

If h is too small, then the subarrays $\texttt{data}_i$ of array `data` could be too large, and sorting algorithms might prove inefficient as well. On the other hand, if h is too large, then too many small subarrays are created, and although they are sorted, it does not substantially change the overall order of `data`. Lastly, if only one such partition of `data` is done, the gain on the execution time may be rather modest. To solve that problem, several different subdivisions are used, and for every subdivision, the same procedure is applied separately, as in:

```
determine numbers h_t ... h_1 of ways of dividing array data into subarrays;
for (h=h_t; t > 1; t--, h=h_t)
    divide data into h subarrays;
    for (i = 1; i <= h; i++)
        sort subarray data_i;
sort array data;
```

This idea is the basis of the *diminishing increment sort*, also known as *Shell sort* and named after Donald L. Shell who designed this technique. Note that this pseudocode does not identify a specific sorting method for ordering the subarrays; it can be any simple method. Usually, however, Shell sort uses insertion sort.

The heart of Shell sort is an ingenious division of the array `data` into several subarrays. The trick is that elements spaced further apart are compared first, then the elements closer to each other are compared, and so on, until adjacent elements are compared on the last pass. The original array is logically subdivided into subarrays by

picking every h_tth element as part of one subarray. Therefore, there are h_t subarrays, and for every $h = 1, \ldots, h_t$,

$$\texttt{data}_{h_t h}[i] = \texttt{data}[h_t \cdot i + (h-1)]$$

For example, if $h_t = 3$, the array `data` is subdivided into three subarrays $\texttt{data}_1$,$\texttt{data}_2$, and $\texttt{data}_3$ so that

$\texttt{data}_{31}[0] = \texttt{data}[0], \texttt{data}_{31}[1] = \texttt{data}[3], \ldots, \texttt{data}_{31}[i] = \texttt{data}[3*i], \ldots$
$\texttt{data}_{32}[0] = \texttt{data}[1], \texttt{data}_{32}[1] = \texttt{data}[4], \ldots, \texttt{data}_{32}[i] = \texttt{data}[3*i+1], \ldots$
$\texttt{data}_{33}[0] = \texttt{data}[2], \texttt{data}_{33}[1] = \texttt{data}[5], \ldots, \texttt{data}_{33}[i] = \texttt{data}[3*i+2], \ldots$

and these subarrays are sorted separately. After that, new subarrays are created with an $h_{t-1} < h_t$, and insertion sort is applied to them. The process is repeated until no subdivisions can be made. If $h_t = 5$, the process of extracting subarrays and sorting them is called a 5-sort.

Figure 9.7 shows the elements of the array `data` that are five positions apart and are logically inserted into a separate array, "logically" since physically they still occupy the same positions in `data`. For each value of increment h_t, there are h_t subarrays, and each of them is sorted separately. As the value of the increment decreases, the number of subarrays decreases accordingly, and their sizes grow. Since much of `data`'s disorder has been removed in the earlier iterations, on the last pass, the array is much closer to its final form than before all the intermediate h-sorts.

There is still one problem that has to be addressed, namely, choosing the optimal value of the increment. In the example in Figure 9.7, the value of 5 is chosen to begin with, then 3, and 1 is used for the final sort. But why these values? Unfortunately, no convincing answer can be given. In fact, any decreasing sequence of increments can be used as long as the last one, h_1, is equal to 1. Donald Knuth has shown that even if there are only two increments, $(\frac{16n}{\pi})^{\frac{1}{5}}$ and 1, Shell sort is more efficient than insertion sort because it takes $O(n^{\frac{5}{3}})$ time instead of $O(n^2)$. But the efficiency of Shell sort can be improved by using a larger number of increments. It is imprudent, however, to use sequences of increments such as 1, 2, 4, 8, . . . or 1, 3, 6, 9, . . . since the mixing effect of data is lost.

For example, when using 4-sort and 2-sort, a subarray, $\texttt{data}_{2,i}$, for $i = 1, 2$, consists of elements of two arrays, $\texttt{data}_{4,i}$ and $\texttt{data}_{4,j}$, where $j = i + 2$, and only those. It is much better if elements of $\texttt{data}_{4,i}$ do not meet together again in the same array since a faster reduction in the number of exchange inversions is achieved if they are sent to different arrays when performing the 2-sort. Using only powers of 2 for the increments, as in Shell's original algorithm, the items in the even and odd positions of the array do not interact until the last pass, when the increment equals 1. This is where the mixing effect (or lack thereof) comes into play. But there is no formal proof indicating which sequence of increments is optimal. Extensive empirical studies along with some theoretical considerations suggest that it is a good idea to choose increments satisfying the conditions

$$h_1 = 1$$

$$h_{i+1} = 3h_i + 1$$

FIGURE 9.7 The array [10 8 6 20 4 3 22 1 0 15 16] sorted by Shell sort.

`data` before 5-sort	10	8	6	20	4	3	22	1	0	15	16
Five subarrays before sorting	10	—	—	—	—	3	—	—	—	—	16
		8	—	—	—	—	22				
			6	—	—	—	—	1			
				20	—	—	—	—	0		
					4	—	—	—	—	15	
Five subarrays after sorting	3	—	—	—	—	10	—	—	—	—	16
		8	—	—	—	—	22				
			1	—	—	—	—	6			
				0	—	—	—	—	20		
					4	—	—	—	—	15	
`data` after 5-sort and before 3-sort	3	8	1	0	4	10	22	6	20	15	16
Three subarrays before sorting	3	—	—	0	—	—	22	—	—	15	
		8	—	—	4	—	—	6	—	—	16
			1	—	—	10	—	—	20		
Three subarrays after sorting	0	—	—	3	—	—	15	—	—	22	
		4	—	—	6	—	—	8	—	—	16
			1	—	—	10	—	—	20		
`data` after 3-sort and before 1-sort	0	4	1	3	6	10	15	8	20	22	16
`data` after 1-sort	0	1	3	4	6	8	10	15	16	20	22

and stop with h_t for which $h_{t+2} \geq n$. For $n = 10{,}000$, this gives the sequence

$$1, 4, 13, 40, 121, 364, 1093, 3280$$

Experimental data have been approximated by the exponential function, the estimate, $1.21n^{\frac{5}{4}}$, and the logarithmic function $.39n \ln^2 n - 2.33n \ln n = O(n \ln^2 n)$. The first form fits the results of the tests better. $1.21n^{1.25} = O(n^{1.25})$ is much better than $O(n^2)$ for insertion sort, but it is still much greater than the expected $O(n \lg n)$ performance.

Figure 9.8 contains a function to sort the array `data` using Shell sort. Note that before sorting starts, increments are computed and stored in the array `increments`.

The core of Shell sort is to divide an array into subarrays by taking elements h positions apart. There are three features of this algorithm that vary from one implementation to another:

1. The sequence of increments
2. A simple sorting algorithm applied in all passes except the last
3. A simple sorting algorithm applied only in the last pass, for 1-sort

FIGURE 9.8 Implementation of Shell sort.

```
template<class T>
void ShellSort(T data[], int arrSize) {
    register int i, j, hCnt, h;
    int increments[20], k;
//  create an appropriate number of increments h
    for (h = 1, i = 0; h < arrSize; i++) {
        increments[i] = h;
        h = 3*h + 1;
    }
 // loop on the number of different increments h
    for (i--; i >= 0; i--) {
        h = increments[i];
     // loop on the number of subarrays h-sorted in ith pass
        for (hCnt = h; hCnt<2*h; hCnt++) {
         // insertion sort for subarray containing every hth element of
            for (j = hCnt; j < arrSize; ) {   // array data
                T tmp = data[j];
                k = j;
                while (k-h >= 0 && tmp < data[k-h]) {
                    data[k] = data[k-h];
                    k -= h;
                }
                data[k] = tmp;
                j += h;
            }
        }
    }
}
```

In our implementation as in Shell's, insertion sort is applied in all h-sorts, but other sorting algorithms can be used. For example, Dobosiewicz uses bubble sort for the last pass and insertion sort for other passes. Incerpi and Sedgewick use two iterations of cocktail shaker sort and a version of bubble sort in each h-sort and finish with insertion sort obtaining what they call a *shakersort*. All these versions perform better than simple sorting methods, although there are some differences in performance among versions. Analytical results concerning the complexity of these sorts are not available. All results regarding complexity are of an empirical nature.

9.3.2 Heap Sort

Selection sort makes $O(n^2)$ comparisons and is very inefficient, especially for large n. But it performs relatively few moves. If the comparison part of the algorithm is improved, the end results can be promising.

Heap sort was invented by John Williams and uses the approach inherent to selection sort. Selection sort finds among the n elements the one that precedes all other $n-1$ elements, then the least element among those $n-1$ items, and so forth, until the array is sorted. To have the array sorted in ascending order, heap sort puts the largest element at the end of the array, then the second largest in front of it, and so on. Heap sort starts from the end of the array by finding the largest elements, whereas selection sort starts from the beginning using the smallest elements. The final order in both cases is indeed the same.

Heap sort uses a heap as described in Section 6.9. A heap is a binary tree with the following two properties:

1. The value of each node is not less than the values stored in each of its children.
2. The tree is perfectly balanced and the leaves in the last level are all in the leftmost positions.

A tree has the heap property if it satisfies condition 1. Both conditions are useful for sorting, although this is not immediately apparent for the second condition. The goal is to use only the array being sorted without using additional storage for the array elements; by condition 2, all elements are located in consecutive positions in the array starting from position 0, with no unused position inside the array. In other words, condition 2 reflects the packing of an array with no gaps.

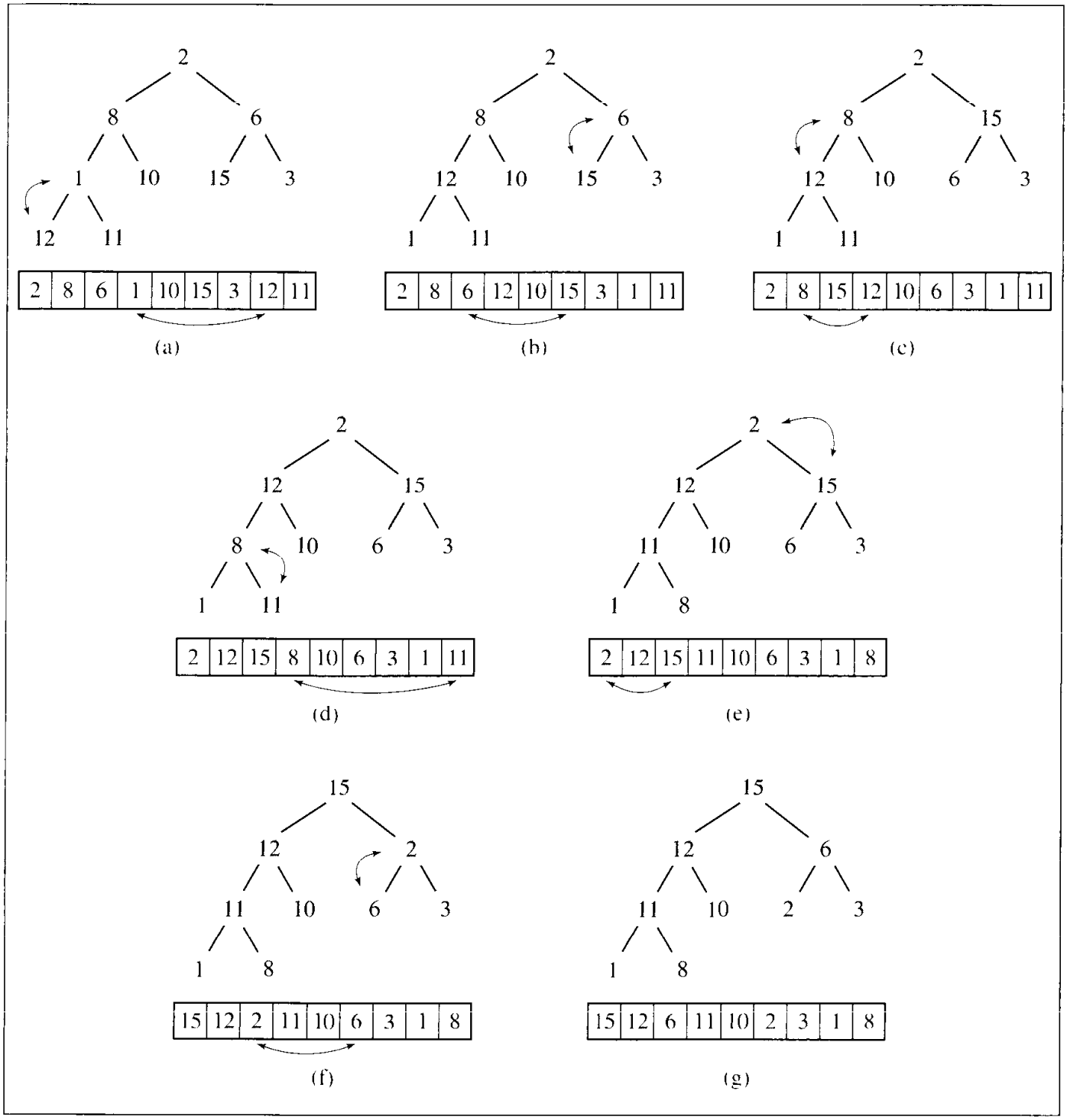
Elements in a heap are not perfectly ordered. It is only known that the largest element is in the root node and that, for each other node, all its descendants are not greater than the element in this node. Heap sort thus starts from the heap, puts the largest element at the end of the array, and restores the heap that now has one less element. From the new heap, the largest element is removed and put in its final position, and then the heap property is restored for the remaining elements. Thus, in each round, one element of the array ends up in its final position, and the heap becomes smaller by this one element. The process ends with exhausting all elements from the heap and is summarized in the following pseudocode:

```
heapsort(data[],n)
    transform data into a heap;
    for (i = n-1; i > 1; i--)
        swap the root with the element in position i;
        restore the heap property for the tree data[0],...,data[i-1];
```

In the first phase of heap sort, an array is transformed into a heap. In this process, we use a bottom-up method devised by Floyd and described in Section 6.9.2. All steps leading to the transformation of the array [2 8 6 1 10 15 3 12 11] into a heap are illustrated in Figure 9.9 (cf. Figure 6.58).

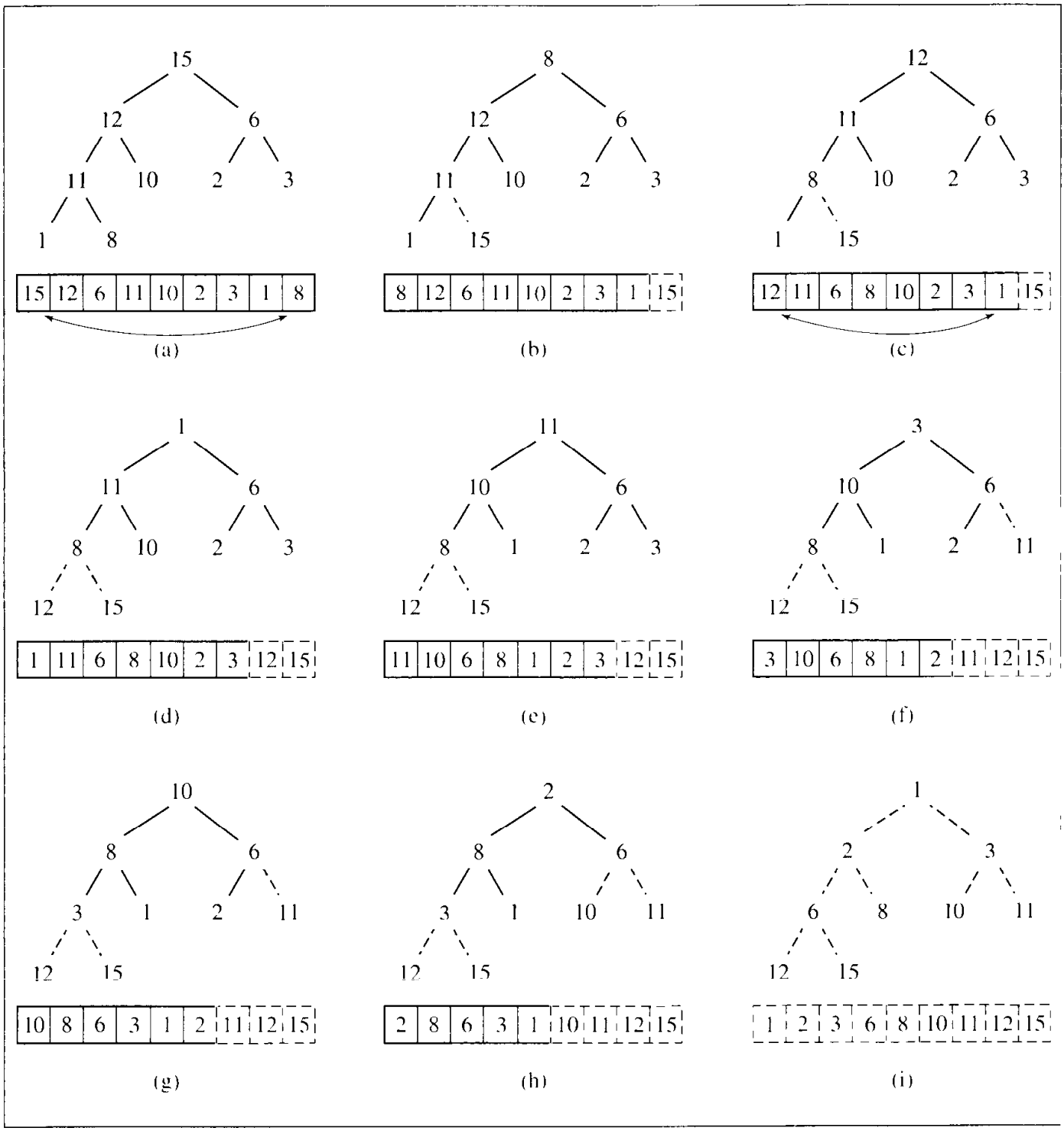
The second phase begins after the heap has been built (Figures 9.9g and 9.10a). At that point, the largest element, number 15, is moved to the end of the array. Its place is

FIGURE 9.9 Transforming the array [2 8 6 1 10 15 3 12 11] into a heap.



taken by 8, thus violating the heap property. The property has to be restored, but this time it is done for the tree without the largest element, 15. Because it is already in its proper position, it does not need to be considered anymore and is removed (pruned) from the tree (indicated by the dashed lines in Figure 9.10). Now, the largest element among `data[0]`, . . . , `data[n-2]` is looked for. To that end, the function

FIGURE **9.10** Execution of heap sort on the array [15 12 6 11 10 2 3 1 8], which is the heap constructed in Figure 9.9.



`moveDown()` from Section 6.9 (Figure 6.56) is called to construct a heap out of all the elements of `data` except the last, `data[n-1]`, which results in the heap in Figure 9.10c. Number 12 is sifted up and then swapped with 1, giving the tree in Figure 9.10d. The function `moveDown()` is called again to select 11 (Figure 9.10e), and the

element is swapped with the last element of the current subarray, which is 3 (Figure 9.10f). Now 10 is selected (Figure 9.10g) and exchanged with 2 (Figure 9.10h). The reader can easily construct trees and heaps for the next passes through the loop of `heapsort()`. After the last pass, the array is in ascending order and the tree is ordered accordingly. An implementation of `heapsort()` is as follows:

```
template<class T>
void heapsort(T data[], int size) {
    for (int i = size/2 - 1; i >= 0; --i)   // create the heap;
        moveDown (data,i,size-1);
    for (i = size-1; i >= 1; --i) {
        swap(data[0],data[i]); // move the largest item to data[i];
        moveDown(data,0,i-1);  // restore the heap property;
    }
}
```

Heap sort might be considered inefficient because the movement of data seems to be very extensive. First, all effort is applied to moving the largest element to the leftmost side of the array in order to move it to the furthest right. But therein lies its efficiency. In the first phase, to create the heap, `heapsort()` uses `moveDown()`, which performs $O(n)$ steps (see Section 6.9.2).

In the second phase, `heapsort()` exchanges $n-1$ times the root with the element in position i and also restores the heap $n-1$ times which in the worst case causes `moveDown()` to iterate $\lg i$ times to bring the root down to the level of the leaves. Thus, the total number of moves in all executions of `moveDown()` in the second phase of `heapsort()` is $\sum_{i=1}^{n-1} \lg i$, which is $O(n \lg n)$. In the worst case, `heapsort()` requires $O(n)$ steps in the first phase, and in the second phase, $n-1$ swaps and $O(n \lg n)$ operations to restore the heap property, which gives $O(n) + O(n \lg n) + (n-1) = O(n \lg n)$ exchanges for the whole process in the worst case.

For the best case, when the array contains identical elements, `moveDown()` is called $\frac{n}{2}$ times in the first phase, but no moves are performed. In the second phase, `heapsort()` makes one swap to move the root element to the end of the array, resulting in only $n-1$ moves. Also, in the best case, n comparisons are made in the first phase and $2(n-1)$ in the second. Hence, the total number of comparisons in the best case is $O(n)$. However, if the array has distinct elements, then in the best case the number of comparisons equals $n \lg n - O(n)$ (Ding and Weiss 1991).

9.3.3 Quicksort

Shell sort approached the problem of sorting by dividing the original array into subarrays, sorting them separately, and then dividing them again to sort the new subarrays until the whole array is sorted. The goal was to reduce the original problem to subproblems that can be solved more easily and quickly. The same reasoning was a guiding principle for C. A. R. Hoare, who invented an algorithm appropriately called *quicksort*.

The original array is divided into two subarrays, the first of which contains elements less than or equal to a chosen key called the *bound* or *pivot*. The second array includes elements equal to or greater than the bound. The two subarrays can be sorted

separately, but before this is done, the partition process is repeated for both subarrays. As a result, two new bounds are chosen, one for each subarray. The four subarrays are created because each subarray obtained in the first phase is now divided into two segments. This process of partitioning is carried down until there are only one-cell arrays that do not need to be sorted at all. By dividing the task of sorting a large array into two simpler tasks and then dividing those tasks into even simpler tasks, it turns out that in the process of getting prepared to sort, the data have already been sorted. Since the sorting has been somewhat dissipated in the preparation process, this process is the core of quicksort.

Quicksort is recursive in nature because it is applied to both subarrays of an array at each level of partitioning. This technique is summarized in the following pseudocode:

```
quicksort(array[])
   if length(array) > 1
      choose bound; // partition array into subarray1 and subarray2
      while there are elements left in array
            include element either in subarray1 = {el: el ≤ bound};
            or in subarray2 = {el: el ≥ bound};
      quicksort(subarray1);
      quicksort(subarray2);
```

To partition an array, two operations have to be performed: A bound has to be found and the array has to be scanned to place the elements in the proper subarrays. However, choosing a good bound is not a trivial task. The problem is that the subarrays should be approximately the same length. If an array contains the numbers 1 through 100 (in any order) and 2 is chosen as a bound, then an imbalance results: The first subarray contains only one number after partitioning, whereas the second has 99 numbers.

A number of different strategies for selecting a bound have been developed. One of the simplest consists of choosing the first element of an array. That approach can suffice for some applications. However, since many arrays to be sorted already have many elements in their proper positions, a more cautious approach is to choose the element located in the middle of the array. This approach is incorporated in the implementation in Figure 9.11.

Another task is scanning the array and dividing the elements between its two subarrays. The pseudocode is vague about how this can be accomplished. In particular, it does not decide where to place an element equal to the bound. It only says that elements are placed in the first subarray if they are less than or the same as the bound and in the second if they are greater than or the same as the bound. The reason is that the difference between the lengths of the two subarrays should be minimal. Therefore, elements equal to the bound should be so divided between the two subarrays to make this difference in size minimal. The details of handling this depend on a particular implementation, and one such implementation is given in Figure 9.11. In this implementation, `quicksort(data[],n)` preprocesses the array to be sorted by locating the largest element in the array and exchanging it with the last element of the array. Having the largest element at the end of the array prevents the index `lower` from running off the end of the array. This could happen in the first inner `while` loop if the bound were the largest element in the array. The index `lower` would be constantly

FIGURE **9.11** Implementation of quicksort.

```
template<class T>
void quicksort(T data[], int first, int last) {
    int lower = first+1, upper = last;
    swap(data[first],data[(first+last)/2]);
    T bound = data[first];
    while (lower <= upper) {
        while (data[lower] < bound)
            lower++;
        while (bound < data[upper])
            upper--;
        if (lower < upper)
            swap(data[lower++],data[upper--]);
        else lower++;
    }
    swap(data[upper],data[first]);
    if (first < upper-1)
        quicksort (data,first,upper-1);
    if (upper+1 < last)
        quicksort (data,upper+1,last);
}

template<class T>
void quicksort(T data[], int n) {
    if (n < 2)
        return;
    for (int i = 1, max = 0; i < n; i++)// find the largest
        if (data[max] < data[i])        // element and put it
            max = i;                    // at the end of data[];
    swap(data[n-1],data[max]); // largest el is now in its
    quicksort(data,0,n-2);     // final position;
}
```

incremented eventually causing an abnormal program termination. Without this preprocessing, the first inner `while` loop would have to be

```
while (lower < last && data[lower] < bound)
```

The first test, however, would be necessary only in extreme cases, but it would be executed in each iteration of this `while` loop.

In this implementation, the main property of the bound is used, namely, that it is a boundary item. Hence, as befits the boundary item, it is placed on the borderline between

the two subarrays obtained as a result of one call to `quicksort()`. In this way, the bound is located in its final position and can be excluded from further processing. To ensure that the bound is not moved around, it is stashed in the first position, and after partitioning is done, it is moved to its proper position, which is the rightmost position of the first subarray.

Figure 9.12 contains an example of partitioning the array [8 5 4 7 6 1 6 3 8 12 10]. In the first partitioning, the largest element in the array is located and exchanged with the last element resulting in the array [8 5 4 7 6 1 6 3 8 10 12]. Because the last element is already in its final position, it does not have to be processed anymore. Therefore, in the first partitioning, `lower` = 1, `upper` = 9, and the first element of the array, 8, is exchanged with the bound, 6 in position 4, so that the array is [6 5 4 7 8 1 6 3 8 10 12] (Figure 9.12b). In the first iteration of the outer `while` loop, the inner `while` loop moves `lower` to position 3 with 7, which is greater than the bound. The second inner `while` loop moves `upper` to position 7 with 3, which is less than the bound (Figure 9.12c). Next the elements in these two cells are exchanged, giving the array [6 5 4 3 8 1 6 7 8 10 12] (Figure 9.12d). Then `lower` is incremented to 4 and `upper` is decremented to 6 (Figure 9.12e). This concludes the first iteration of the outer `while` loop.

In its second iteration, neither of the two inner `while` loops modifies any of the two indexes because `lower` indicates a position occupied by 8, which is greater than the bound, and `upper` indicates a position occupied by 6, which is equal to the bound. The two numbers are exchanged (Figure 9.12f), and then both indexes are updated to 5 (Figure 9.12g).

In the third iteration of the outer `while` loop, `lower` is moved to the next position containing 8, which is greater than the bound, and `upper` stays in the same position because 1 in this position is less than the bound (Figure 9.12h). But at that point, `lower` and `upper` cross each other, so no swapping takes place, and after a redundant increment of `lower` to 7, the outer `while` loop is exited. At that point, `upper` is the index of the rightmost element of the first subarray (with the element not exceeding the bound), so the element in this position is exchanged with the bound (Figure 9.12i). In this way, the bound is placed in its final position and can be excluded from subsequent processing. Therefore, the two subarrays that are processed next are the left subarray, with elements to the left of the bound, and the right subarray, with elements to its right (Figure 9.12j). Then partitioning is performed for these two subarrays separately, and then for subarrays of these subarrays, until subarrays have less than two elements. The entire process is summarized in Figure 9.13, in which all the changes in all current arrays are indicated.

The worst case occurs if in each invocation of `quicksort()`, the smallest (or largest) element of the array is chosen for the bound. This is the case if we try to sort the array [5 3 1 2 4 6 8]. The first bound is 1, and the array is broken into an empty array and the array [3 5 2 4 6] (the largest number, 8, does not participate in partitioning). The new bound is 2, and again only one nonempty array, [5 3 4 6], is obtained as the result of partitioning. The next bound and array returned by partition are 3 and [5 4 6], then 4 and [5 6], and finally 5 and [6]. The algorithm thus operates on arrays of size $n-1, n-2, \ldots, 2$. The partitions require $n-2+n-3+\cdots+1$ comparisons, and for each partition, only the bound is placed in the proper position. This results in a run time equal to $O(n^2)$, which is hardly a desirable result, especially for large arrays or files.

FIGURE **9.12** Partitioning the array [8 5 4 7 6 1 6 3 8 12 10] with `quicksort( )`.

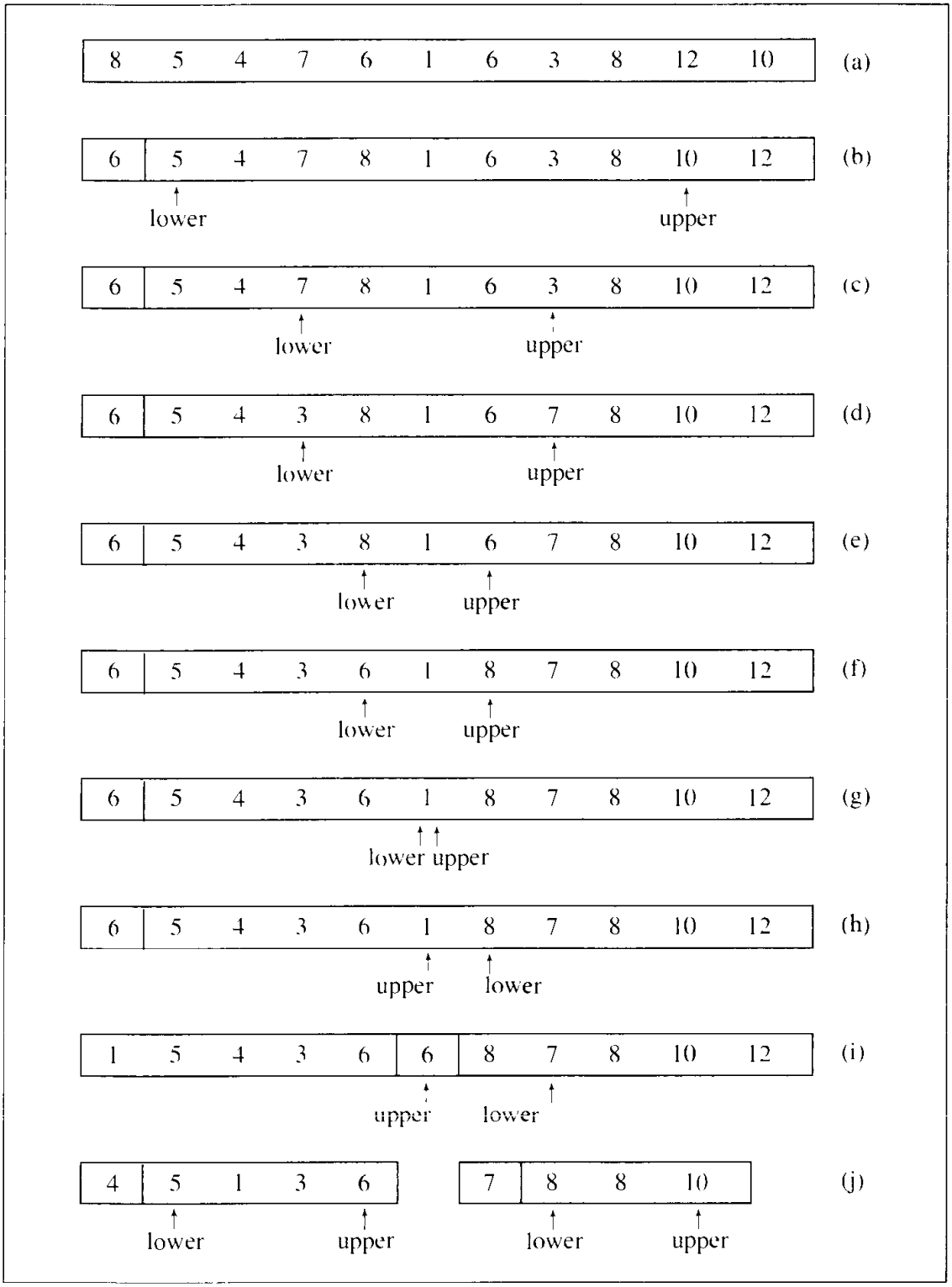
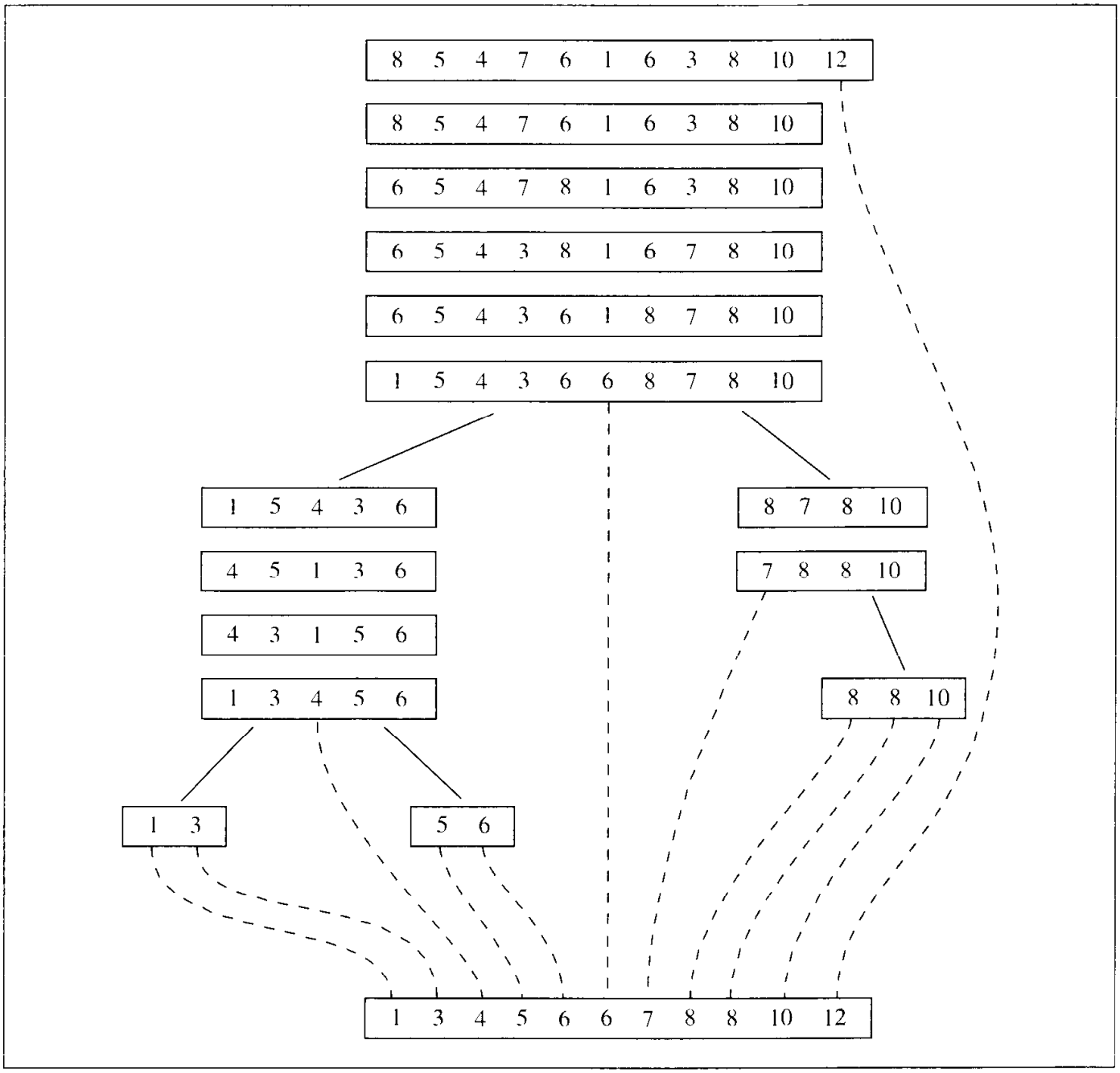


FIGURE 9.13 Sorting the array [8 5 4 7 6 1 6 3 8 12 10] with `quicksort()`.



The best case is when the bound divides an array into two subarrays of approximately length $\frac{n}{2}$. If the bounds for both subarrays are well chosen, the partitions produce four new subarrays, each of them with approximately $\frac{n}{4}$ cells. If, again, the bounds for all four subarrays divide them evenly, the partitions give eight subarrays, each with approximately $\frac{n}{8}$ elements. Therefore, the number of comparisons performed for all partitions is approximately equal to

$$n + 2\frac{n}{2} + 4\frac{n}{4} + 8\frac{n}{8} + \ldots + n\frac{n}{n} = n(\lg n + 1)$$

which is $O(n \lg n)$. This is due to the fact that parameters in the terms of this sum (and also the denominators) form a geometric sequence so that $n = 2^k$ for $k = \lg n$ (assuming that n is a power of 2).

To answer the question asked before: Is the average case, when the array is ordered randomly, closer to the best case, $n \lg n$, or to the worst, $O(n^2)$? Some calculations show that the average case requires only $O(n \lg n)$ comparisons (see Appendix A3), which is the desired result. The validity of this figure can be strengthened by referring to the tree obtained after disregarding the bottom rectangle in Figure 9.13. This tree indicates how important it is to keep the tree balanced, for the smaller the number of levels, the quicker the sorting process. In the extreme case, the tree can be turned into a linked list in which every nonleaf node has only one child. That rather rare phenomenon is possible and prevents us from calling quicksort the ideal sort. But quicksort seems to be closest to such an ideal because, as analytic studies indicate, it outperforms other efficient sorting methods by at least a factor of 2.

How can the worst case be avoided? The partition procedure should produce arrays of approximately the same size, which can be achieved if a good bound is chosen. This is the crux of the matter: How can the best bound be found? Only two methods will be mentioned. The first method randomly generates a number between `first` and `last`. This number is used as an index of the bound, which is then interchanged with the first element of the array. In this method, the partition process proceeds as before. Good random number generators may slow down the execution time as they themselves often use sophisticated and time-consuming techniques. Thus, this method is not highly recommended.

The second method chooses a median of three elements: the first, middle, and last. For the array [1 5 4 7 8 6 6 3 8 12 10], 6 is chosen from the set [1 6 10], and for the first generated subarray, the bound 4 is chosen from the set [1 4 6]. Obviously, there is the possibility that all three elements are always the smallest (or the largest) in the array, but it does not seem very likely.

Is quicksort the best sorting algorithm? It certainly is—usually. It is not bulletproof, however, and some problems have already been addressed in this section. First, everything hinges on which element of the file or array is chosen for the bound. Ideally, it should be the median element of the array. An algorithm to choose a bound should be flexible enough to handle all possible orderings of the data to be sorted. Because some cases always slip by these algorithms, from time to time quicksort can be expected to be anything but quick.

Second, it is inappropriate to use quicksort for small arrays. For arrays with fewer than ten items, insertion sort is more efficient than quicksort (Cook and Kim 1980). In this case the initial pseudocode can be changed to

```
quicksort2 (array)
   if length(array) > 10
      partition array into subarray1 and subarray2;
      quicksort2(subarray1);
      quicksort2(subarray2);
   else insertionsort(array);
```

and the implementations changed accordingly. However, the table in Figure 9.18 indicates that the improvement is not significant.

9.3.4 Mergesort

The problem with quicksort is that its complexity in the worst case is $O(n^2)$ because it is difficult to control the partitioning process. Different methods of choosing a bound attempt to make the behavior of this process fairly regular. However, there is no guarantee that partitioning results in arrays of approximately the same size. Another strategy is to make partitioning as simple as possible and concentrate on merging the two sorted arrays. This strategy is characteristic of *mergesort*. It was one of the first sorting algorithms used on a computer and was developed by John von Neumann.

The key process in mergesort is merging sorted halves of an array into one sorted array. However, these halves have to be sorted first, which is accomplished by merging the already sorted halves of these halves. This process of dividing arrays into two halves stops when the array has fewer than two elements. The algorithm is recursive in nature and can be summarized in the following pseudocode:

```
mergesort(data)
   if data have at least two elements
      mergesort(left half of data);
      mergesort(right half of data);
      merge(both halves into a sorted list);
```

Merging two subarrays into one is a relatively simple task, as indicated in this pseudocode:

```
merge(array1, array2, array3)
   i1, i2, i3 are properly initialized;
   while both array2 and array3 contain elements
      if array2[i2] < array3[i3]
           array1[i1++] = array2[i2++];
      else array1[i1++] = array3[i3++];
   load into array1 the remaining elements of either array2 or array3;
```

For example, if `array2` = [1 4 6 8 10] and `array3` = [2 3 5 22], then the resulting `array1` = [1 2 3 4 5 6 8 10 22].

The pseudocode for `merge()` suggests that `array1`, `array2`, and `array3` are physically separate entities. However, for the proper execution of `mergesort()`, `array1` is a concatenation of `array2` and `array3` so that `array1` before the execution of `merge()` is [1 4 6 8 10 2 3 5 22]. In this situation, `merge()` leads to erroneous results, since after the second iteration of the `while` loop, `array2` is [1 2 6 8 10] and `array1` is [1 2 6 8 10 2 3 5 22]. Therefore, a temporary array has to be used during the merging process. At the end of the merging process, the contents of this temporary array are transferred to `array1`. Because `array2` and `array3` are subarrays of `array1`, they do not need to be passed as parameters to `merge()`. Instead, indexes for the beginning and the end of `array1` are passed, since `array1` can be a part of another array. The new pseudocode is

```
merge (array1, first, last)
   mid = (first + last) / 2;
   i1 = 0;
```

```
i2 = first;
i3 = mid + 1;
while both left and right subarrays of array1 contain elements
   if array1[i2] < array1[i3]
        temp[i1++] = array1[i2++];
   else temp[i1++] = array1[i3++];
load into temp the remaining elements of array1;
load to array1 the content of temp;
```

Since the entire `array1` is copied to `temp` and then `temp` is copied back to `array1`, the number of movements in each execution of `merge()` is always the same and is equal to $2 \cdot$ (`last` $-$ `first` $+ 1$). The number of comparisons depends on the ordering in `array1`. If `array1` is in order or if the elements in the right half precede the elements in the left half, the number of comparisons is (`first` + `last` -1). The worst case is when the last element of one half precedes only the last element of the other half, as in [1 6 10 12] and [5 9 11 13]. In this case, the number of comparisons is `last` $-$ `first`. For an n-element array, the number of comparisons is $n-1$.

The pseudocode for `mergesort()` is now

```
mergesort (data, first, last)
   if first < last
      mid = (first + last) / 2;
      mergesort(data, first, mid);
      mergesort(data, mid+1, last);
      merge(data, first, last);
```

Figure 9.14 illustrates an example using this sorting algorithm. This pseudocode can be used to analyze the computing time for mergesort. For an n-element array, the number of movements is computed by the following recurrence relation:

$$M(1) = 0$$
$$M(n) = 2M\left(\frac{n}{2}\right) + 2n$$

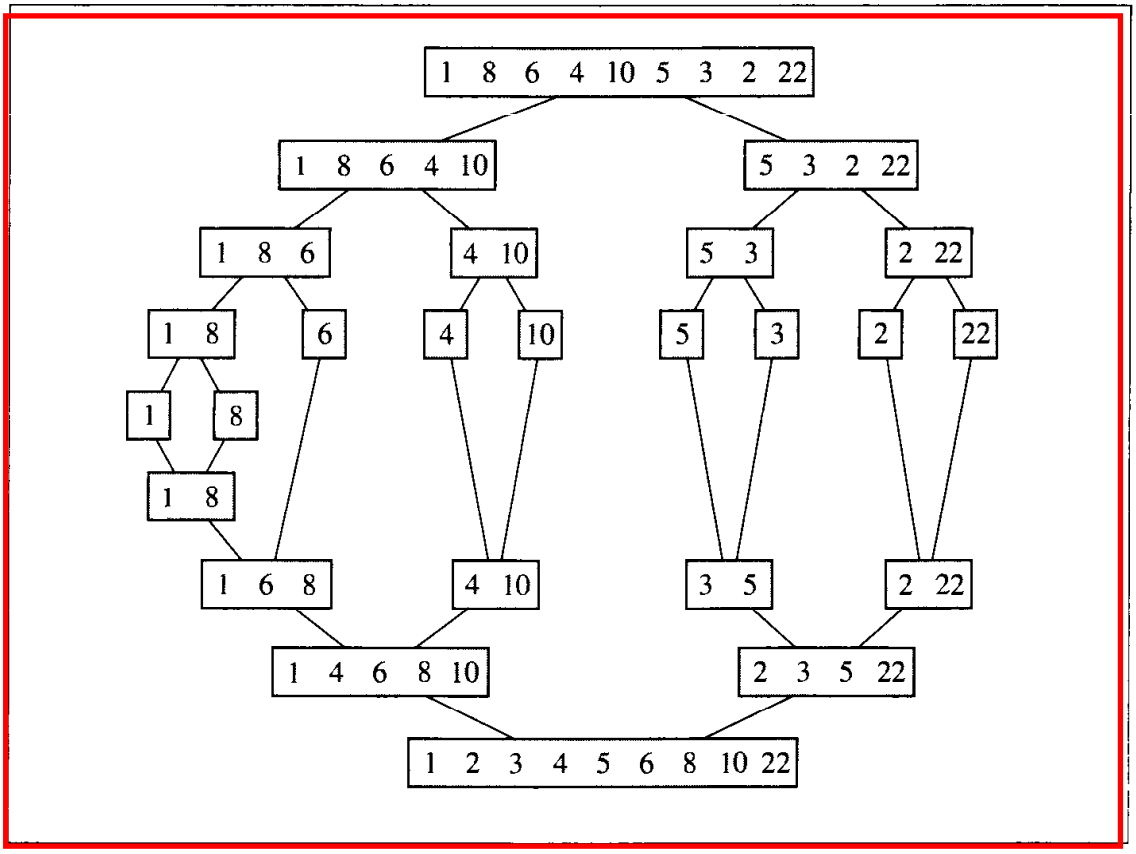
$M(n)$ can be computed in the following way:

$$\begin{aligned} M(n) &= 2\left(2M\left(\frac{n}{4}\right) + 2\left(\frac{n}{2}\right)\right) + 2n = 4M\left(\frac{n}{4}\right) + 4n \\ &= 4\left(2M\left(\frac{n}{8}\right) + 2\left(\frac{n}{4}\right)\right) + 4n - 8M\left(\frac{n}{8}\right) + 6n \\ &\vdots \\ &= 2^i M\left(\frac{n}{2^i}\right) + 2in \end{aligned}$$

Choosing $i = \lg n$ so that $n = 2^i$ allows us to infer

$$M(n) = 2^i M\left(\frac{n}{2^i}\right) + 2in = nM(1) + 2n \lg n = 2n \lg n = O(n \lg n)$$

FIGURE 9.14 The array [1 8 6 4 10 5 3 2 22] sorted by mergesort.



The number of comparisons in the worst case is given by a similar relation:

$$C(1) = 0$$

$$C(n) = 2C\left(\frac{n}{2}\right) + n - 1$$

which also results in $C(n)$ being $O(n \lg n)$.

Mergesort can be made more efficient by replacing recursion with iteration (see the exercises at the end of this chapter) or by applying insertion sort to small portions of an array, a technique that was suggested for quicksort. However, mergesort has one serious drawback: the need for additional storage for merging arrays, which for large amounts of data could be an insurmountable obstacle. One solution to this drawback uses a linked list; analysis of this method is left as an exercise.

9.3.5 Radix Sort

Radix sort is a popular way of sorting used in everyday life. To sort library cards, we may create as many piles of cards as letters in the alphabet, each pile containing authors whose names start with the same letter. Then, each pile is sorted separately using

the same method; namely, piles are created according to the second letter of the authors' names. This process continues until the number of times the piles are divided into smaller piles equals the number of letters of the longest name. This method is actually used when sorting mail in the post office, and it was used to sort 80 column cards of coding information in the early days of computers.

When sorting library cards, we proceed from left to right. This method can also be used for sorting mail since all zip codes have the same length. However, it may be inconvenient for sorting lists of integers because they may have an unequal number of digits. If applied, this method would sort the list [23 123 234 567 3] into the list [123 23 234 3 567]. To get around this problem, zeros can be added in front of each number to make them of equal length so that the list [023 123 234 567 003] is sorted into the list [003 023 123 234 567]. Another technique looks at each number as a string of bits so that all integers are of equal length. This approach will be discussed shortly. Still another way to sort integers is by proceeding right to left, and this method is discussed now.

When sorting integers, ten piles numbered 0 through 9 are created, and initially, integers are put in a given pile according to their rightmost digit so that 93 is put in pile 3. Then, piles are combined and the process is repeated, this time with the second rightmost digit; in this case, 93 ends up on pile 9. The process ends after the leftmost digit of the longest number is processed. The algorithm can be summarized in the following pseudocode:

```
radixsort()
   for (d = 1; d <= the position of the leftmost digit of longest number; d++)
      distribute all numbers among piles 0 through 9 according to the dth digit;
      put all integers on one list;
```

The key to obtaining a proper outcome is the way the ten piles are implemented and then combined. For example, if these piles are implemented as stacks, then the integers 93, 63, 64, 94 are put on piles 3 and 4 (other piles being empty):

```
pile 3: 63 93
pile 4: 94 64
```

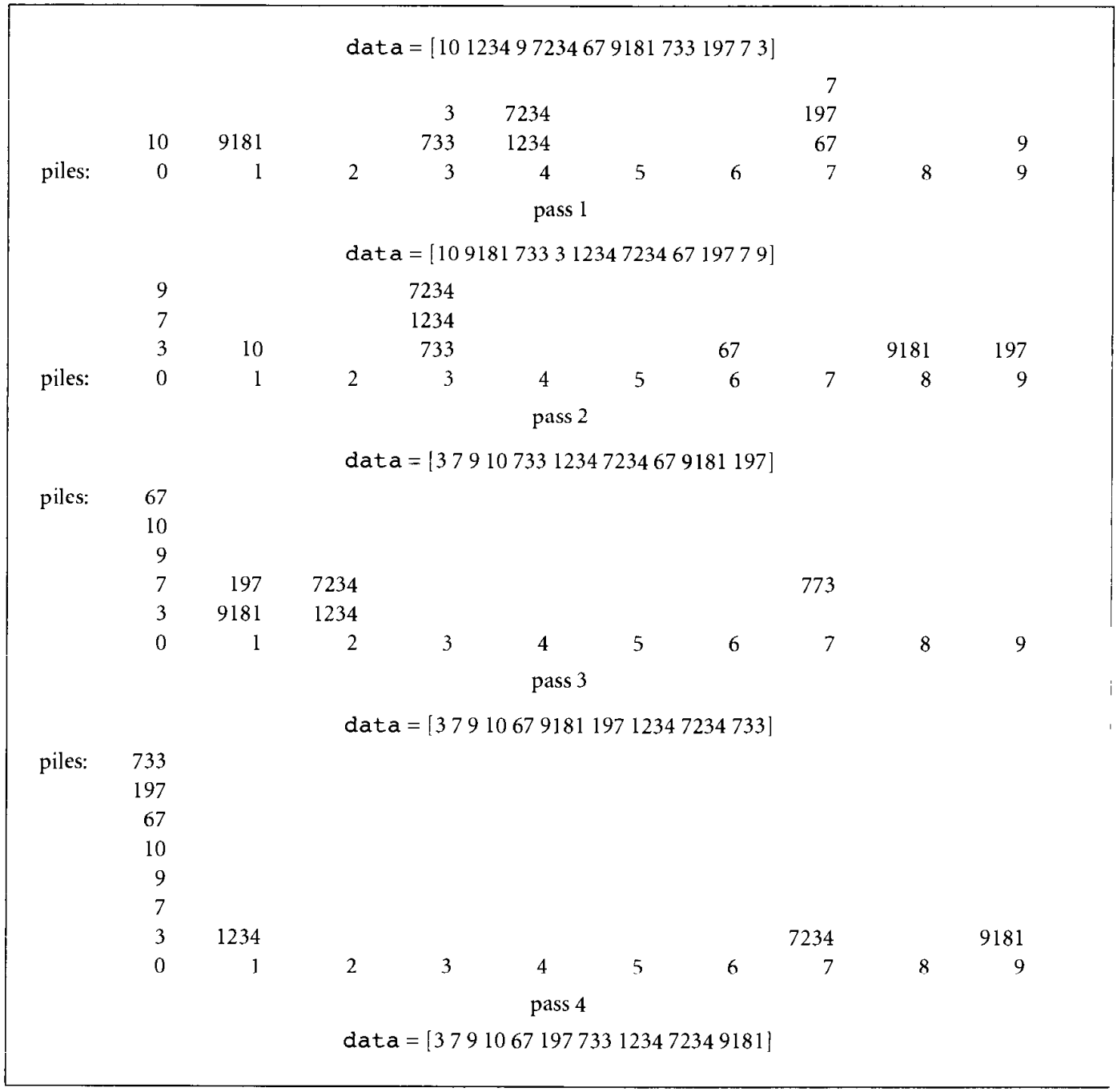
These piles are then combined into the list 63, 93, 94, 64. When sorting them according to the second rightmost digit, the piles are as follows:

```
pile 6: 64 63
pile 9: 94 93
```

and the resulting list is 64, 63, 94, 93. The processing is finished, but the result is an improperly sorted list.

However, if piles are organized as queues, the relative order of elements on the list is retained. When integers are sorted according to the digit in position d, then within each pile, integers are sorted with regard to the part of the integer extending from digit 1 to $d-1$. For example, if after the third pass, pile 5 contains the integers 12534, 554, 3590, then this pile is ordered with respect to the two rightmost digits of each number. Figure 9.15 illustrates another example of radix sort.

FIGURE 9.15 Sorting the list 10, 1234, 9, 7234, 67, 9181, 733, 197, 7, 3 with radix sort.



Here is an implementation of radix sort:

```
void radixsort(long data[], int n) {
    register int i, j, k, factor;
    const int radix = 10;
    const int digits = 10; // the maximum number of digits for a long
    Queue<long> queues[radix];   // integer;
    for (i = 0, factor = 1; i < digits; factor *= radix, i++) {
```

```
        for (j = 0; j < n; j++)
            queues[(data[j] / factor) % radix].enqueue(data[j]);
        for (j = k = 0; j < radix; j++)
            while (!queues[j].empty())
                data[k++] = queues[j].dequeue();
    }
}
```

This algorithm does not rely on data comparison as did the previous sorting methods. For each integer from `data`, two operations are performed: division by a `factor` to disregard digits following digit d being processed in the current pass and division modulo `radix` (equal to 10) to disregard all digits preceding d for a total of $2n$`digits` = $O(n)$ operations. The operation `div` can be used which combines both `/` and `%`. In each pass, all integers are moved to piles and then back to `data` for a total of $2n$`digits` = $O(n)$ moves. The algorithm requires additional space for piles, which if implemented as linked lists, is equal to $2n$ or $3n$ words depending on the size of the pointers. Our implementation uses only `for` loops with counters; therefore, it requires the same amount of passes for each case: best, average, and worst. The body of the only `while` loop is always executed n times to dequeue integers from all queues.

The foregoing discussion treated integers as combinations of digits. But as already mentioned, they can be regarded as combinations of bits. This time, division and division modulo are not appropriate, since for each pass, one bit for each number has to be extracted. In this case, only two queues are required.

An implementation is as follows:

```
void bitRadixsort(long data[], int n) {
    register int i, j, k, mask = 1;
    const int bits = 31;
    Queue<long> queues[2];
    for (i = 0; i < bits; i++) {
        for (j = 0; j < n; j++)
            queues[data[j] & mask ? 1 : 0].enqueue(data[j]);
        mask <<= 1;
        k = 0;
        while (!queues[0].empty())
            data[k++] = queues[0].dequeue();
        while (!queues[1].empty())
            data[k++] = queues[1].dequeue();
    }
}
```

Division is replaced here by the bitwise and-operation `&`. The variable `mask` has one bit set to 1 and the rest are set to 0. After each iteration, this 1 is shifted to the left. If `data[j] & mask` has a nonzero value, then `data[j]` is put in `queues[1]`; otherwise, it is put in `queues[0]`. Bitwise and is much faster than integer division, but in this example, 31 passes are needed, instead of 10 when the largest integer is 10 digits long. This means $62n$ data movements as opposed to $20n$.

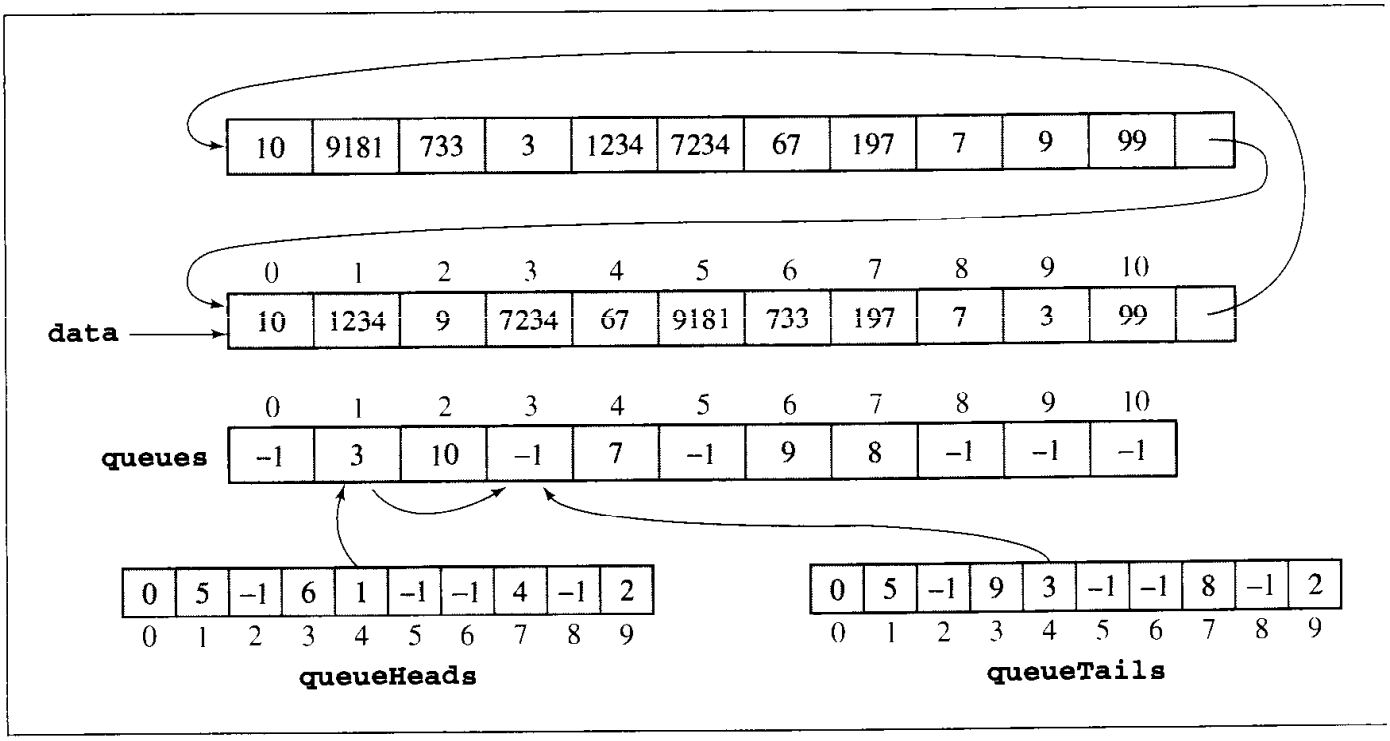
Quicker operations cannot outweigh a larger number of moves: `bitRadixsort()` is much slower than `radixsort()` because the queues are implemented as linked lists, and for each item included in a particular queue, a new node has to be created and attached to the queue. For each item copied back to the original array, the node has to be detached from the queue and disposed of using `delete`. Although theoretically obtained performance $O(n)$ is truly impressive, it does not include operations on queues, and it hinges upon the efficiency of the queue implementation.

A better implementation is an array of size n for each queue, which requires creating these queues only once. The efficiency of the algorithm depends only on the number of exchanges (copying to and from queues). However, if radix r is a large number and a large amount of data has to be sorted, then this solution requires r queues of size n and the number $(r+1) \cdot n$ (original array included) may be unrealistically large.

A better solution uses one integer array `queues` of size n representing linked lists of indexes of numbers belonging to particular queues. Cell i of the array `queueHeads` contains an index of the first number in `data` which belongs to this queue, whose dth digit is i. `queueTails[]` contains a position in `data` of the last number whose dth digit is i. Figure 9.16 illustrates the situation after the first pass, for $d = 1$. `queueHeads[4]` is 1, which means that the number in position 1 in `data`, 1234, is the first number found in `data` with 4 as the last digit. Cell `queues[1]` contains 3, which is an index of the next number in `data` with 4 as the last digit, 7234. Finally, `queues[3]` is −1 to indicate the end of the numbers meeting this condition.

The next stage orders data according to information gathered in `queues`. It copies all the data from the original array to some temporary storage and then back to

FIGURE 9.16 An implementation of radix sort.



this array. To avoid the second copy, two arrays can be used, constituting a two-element circular linked list. After copying, the pointer to the list is moved to the next node, and the array in this node is treated as storage of numbers to be sorted. The improvement is significant since the new implementation runs at least two times faster than the implementation that uses queues (Figure 9.18).

FIGURE **9.17** Demonstration of sorting functions.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional> // greater<>

using namespace std;

class Person {
public:
    Person(char *n = "", int a = 0) {
       name = new char[strlen(n)+1];
       strcpy(name,n);
       age = a;
    }
    bool operator==(const Person& p) const {
       return strcmp(name,p.name) == 0;
    }
    bool operator<(const Person& p) const {
       return strcmp(name,p.name) < 0;
    }
    bool operator>(const Person& p) const {
       return !(*this == p) && !(*this < p);
    }
private:
    char *name;
    int age;
    friend ostream& operator<< (ostream& out, const Person& p) {
        out << "(" << p.name << "," << p.age << ")";
        return out;
    }
};

bool f1(int n) {
    return n < 5;
}
```

Continues

FIGURE 9.17 *(continued)*

```
template<class T>
void printVector(char *s, const vector<T>& v) {
    cout << s << " = (";
    if (v.size() == 0) {
        cout << ")\n";
        return;
    }
    for (vector<T>::const_iterator i = v.begin();
                                   i < v.begin()+v.size()-1; i++)
        cout << *i << ',';
    cout << *i << ")\n";
}

void main() {
    int a[] = {1,4,3,6,7,2,5};
    vector<int> v1(a,a+7), v2(a,a+7), v3(6,9), v4(6,9);
    vector<int>::iterator i1, i2, i3, i4;
    partial_sort(v1.begin(),v1.begin()+3,v1.end());
    printVector("v1",v1);                          // v1 = (1,2,3,6,7,4,5)
    partial_sort(v2.begin()+1,v2.begin()+4,v2.end(),greater<int>());
    printVector("v2",v2);                          // v2 = (1,7,6,5,3,2,4)
    i3 = partial_sort_copy(v2.begin(),v2.begin()+4,v3.begin(),v3.end());
    printVector("v3",v3);                          // v3 = (1,5,6,7,9,9)
    cout << *(i3-1) << ' ' << *i3 << endl;         // 7 9
    i4 = partial_sort_copy(v1.begin(),v1.begin()+4,v4.begin(),v4.end(),
                           greater<int>());
    printVector("v4",v4);                          // v4 = (6,3,2,1,9,9)
    cout << *(i4-1) << ' ' << *i4 << endl;         // 1 9
    i1 = partition(v1.begin(),v1.end(),f1);        // v1 = (1,2,3,4,7,6,5)
    printVector("v1",v1);
    cout << *(i1-1) << ' ' << *i1 << endl;         // 4 7
    i2 = partition(v2.begin(),v2.end(),bind2nd(less<int>(),5));
    printVector("v2",v2);                          // v2 = (1,4,2,3,5,6,7)
    cout << *(i2-1) << ' ' << *i2 << endl;         // 3 5
    sort(v1.begin(),v1.end());                     // v1 = (1,2,3,4,5,6,7)
    sort(v1.begin(),v1.end(),greater<int>());      // v1 = (7,6,5,4,3,2,1)

    vector<Person> pv1, pv2;
    for (int i = 0; i < 20; i++) {
        pv1.push_back(Person("Josie",60 - i));
        pv2.push_back(Person("Josie",60 - i));
    }
```

FIGURE 9.17 *(continued)*

```
    sort(pv1.begin(),pv1.end());        // pv1 = ((Josie,41)...(Josie,60))
    stable_sort(pv2.begin(),pv2.end());// pv2 = ((Josie,60)...(Josie,41))

    vector<int> heap1, heap2, heap3(a,a+7), heap4(a,a+7);
    for (i = 1; i <= 7; i++) {
        heap1.push_back(i);
        push_heap(heap1.begin(),heap1.end());
        printVector("heap1",heap1);
    }
    // heap1 = (1)
    // heap1 = (2,1)
    // heap1 = (3,1,2)
    // heap1 = (4,3,2,1)
    // heap1 = (5,4,2,1,3)
    // heap1 = (6,4,5,1,3,2)
    // heap1 = (7,4,6,1,3,2,5)
    sort_heap(heap1.begin(),heap1.end()); // heap1 = (1,2,3,4,5,6,7)
    for (i = 1; i <= 7; i++) {
        heap2.push_back(i);
        push_heap(heap2.begin(),heap2.end(),greater<int>());
        printVector("heap2",heap2);
    }
    // heap2 = (1)
    // heap2 = (1,2)
    // heap2 = (1,2,3)
    // heap2 = (1,2,3,4)
    // heap2 = (1,2,3,4,5)
    // heap2 = (1,2,3,4,5,6)
    // heap2 = (1,2,3,4,5,6,7)
    sort_heap(heap2.begin(),heap2.end());    // heap2 = (7,6,5,4,3,2,1)
    make_heap(heap3.begin(),heap3.end());    // heap3 = (7,6,5,1,4,2,3)
    sort_heap(heap3.begin(),heap3.end());    // heap3 = (1,2,3,4,5,6,7)
    make_heap(heap4.begin(),heap4.end(),greater<int>());
    printVector("heap4",heap4);              // heap4 = (1,4,2,6,7,3,5)
    sort_heap(heap4.begin(),heap4.end(),greater<int>());
    printVector("heap4",heap4);              // heap4 = (7,6,5,4,3,2,1)
}
```

9.4 Sorting in the Standard Template Library

The STL provides many sorting functions, particularly in the library `<algorithm>`. The functions are implementing some of the sorting algorithms discussed in this chapter: quicksort, heap sort, and mergesort. A program in Figure 9.17 demonstrates these functions. For descriptions of the functions, see also Appendix B.

The first set of functions are partial sorting functions. The first version picks k = `middle - first` smallest elements from a container and puts them in order in the range [`first, middle`). For example,

```
partial_sort(v1.begin(),v1.begin()+3,v1.end());
```

takes the three smallest integers from the entire vector `v1` and puts them in the first three cells of the vector. The remaining integers are put in cells four through seven. In this way, `v1` = [1,4,3,6,7,2,5] is transformed into `v1` = [1,2,3,6,7,4,5]. This version orders elements in ascending order. An ordering relation can be changed if it is provided as the fourth parameter to the function call. For example,

```
partial_sort(v2.begin()+1,v2.begin()+4,v2.end(),greater<int>());
```

picks the largest three integers from the vector `v2` and puts them in positions two to four in descending order. The order of the remaining integers outside this range is not specified. This call transforms `v2` = [1,4,3,6,7,2,5] into `v2` = [1,7,6,5,3,2,4].

The third version of partial sorting takes k = `last1 - first1` or `last2 - first2`, whichever is smaller, first elements from the range [`first1,last1`), and writes them over elements in the range [`first2,last2`). The call

```
i3 = partial_sort_copy(v2.begin(),v2.begin()+4,v3.begin(),v3.end());
```

takes the first four integers in vector `v2` = [1,7,6,5], and puts them in ascending order in the first four cells of `v3` so that `v3` = [9,9,9,9,9,9] is transformed into `v3` = [1,5,6,7,9,9]. As an extra, an iterator is returned that refers to the first position after the last copied number. The fourth version of the partial sorting function is similar to the third, but it allows for providing a relation with which elements are sorted. The program in Figure 9.17 also demonstrates the partition function which orders two ranges with respect to one another by putting in the first range numbers for which a one-argument Boolean function is true; numbers for which the function is false are in the second range. The call

```
i1 = partition(v1.begin(),v1.end(),f1);
```

uses an explicitly defined function `f1()` and transforms `v1` = [1,2,3,6,7,4,5] into `v1` = [1,2,3,4,7,6,5] by putting all numbers less than 5 in front of numbers that are not less than 5. The call

```
i2 = partition(v2.begin(),v2.end(),bind2nd(less<int>(),5));
```

accomplished the same for `v2` with the built-in functional `bind2nd` which binds the second argument of the operator < to 5, effectively generating a function that works just like `f1()`.

Probably the most useful is the function `sort()` that implements quicksort. The call

```
sort(v1.begin(),v1.end());
```

transforms `v1` = [1,2,3,6,7,4,5] into fully ordered `v1` = [1,2,3,4,5,6,7]. The second version of `sort()` allows for specifying any ordering relation.

The STL also provides stable sorting functions. A sorting algorithm is said to be *stable* if equal keys remain in the same relative order in output as they are initially (that is, if `data[i]` equals `data[j]` for $i < j$ and the ith element ends up in the kth position and the jth element in the mth position, then $k < m$). To see the difference between sorting and stable sorting, consider a vector of objects of type `Person`. The definition of `operator<` orders the `Person` objects by name without taking age into account. Therefore, two objects with the same name but with different ages are considered equal, just as in the definition of `operator==`, although this operator is not used in sorting; only the less than operator is. After creating two equal vectors, `pv1` and `pv2`, both equal to [("Josie,"60) . . . ("Josie,"41)], we see that `sort()` transforms this vector into [("Josie,"41) . . . ("Josie,"60)], but `stable_sort()` leaves it intact by retaining the relative order of equal objects. The feat is possible by using mergesort in the stable sorting routine. Note that, for small numbers of objects, `sort()` is also stable because, for a small number of elements, insertion sort is used, not quicksort (cf. `quicksort2()` at the end of Section 9.3.3).

9.5 Concluding Remarks

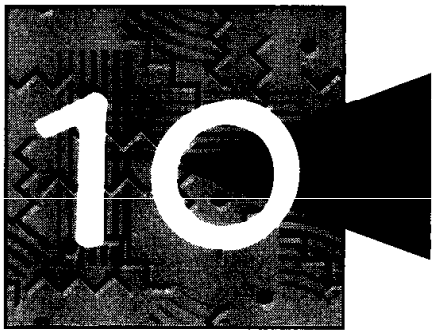
Figure 9.18 compares the run times for different sorting algorithms and different numbers of integers being sorted. They were all run on a Pentium 133MHz PC. At each stage, the number of integers has been doubled to see the factors by which the run times raise. These factors are included in each column except for the first three columns and are shown along with the run times. The factors are rounded to one decimal place, whereas run times (in seconds) are rounded to two decimal places. For example, heap sort required 0.25 seconds to sort an array of 40000 integers in ascending order and 0.55 seconds to sort 8,000 integers also in ascending order. Doubling the number of data is associated with the increase of run time by a factor of 0.55/0.25 = 2.2 and the number 2.2 follows 0.55 in the fourth column.

The table in Figure 9.18 indicates that the run time for elementary sorting methods, which are squared algorithms, grows approximately by a factor of 4 after the amount of data is doubled, whereas the same factor for nonelementary methods, whose complexity is $O(n \lg n)$, is approximately 2. This is also true for the four implementations of radix sort, whose complexity equals $2n$`digits`. The table also shows that quicksort is the fastest algorithm among all sorting methods; most of the time, it runs at least twice as fast as any other algorithm.

FIGURE **9.18** **Comparison of run times for different sorting algorithms and different numbers of integers to be sorted.**

	10,000 Ascending	10,000 Random	10,000 Descending	20,000 Ascending	20,000 Random	20,000 Descending
insertionSort	.00	3.18	6.54	.00 —	13.51 4.2	26.69 4.1
selectionSort	5.65	5.17	5.55	20.82 3.7	22.03 4.3	22.41 4.0
bubbleSort	5.22	10.88	15.60	20.87 4.0	45.09 4.1	1 m 2.51 4.0
ShellSort	.00	.05	.00	.00 —	.11 2.2	.05 —
heapSort	.06	.06	.05	.11 1.8	.11 1.8	.11 2.2
mergesort	.00	.05	.06	.11 —	.11 1.8	.11 1.8
quicksort	.00	.05	.00	.00 —	.05 1.0	.05 —
quicksort2	.00	.00	.00	.05 —	.06 —	.06 —
radixSort	.44	.44	.44	.82 1.9	.88 2.0	.88 2.0
bitRadixSort	.99	1.09	1.10	2.03 2.1	2.14 2.0	2.14 1.9
radixSort2	.17	.11	.17	.33 1.9	.33 3.1	.33 1.9
bitRadixSort2	.21	.22	.22	.44 .21	.44 2.0	.44 2.0

	40,000 Ascending	40,000 Random	40,000 Descending	80,000 Ascending	80,000 Random	80,000 Descending
insertionSort	.00 —	53.66 4.0	1 m 54.35 4.3	.05 —	3 m 46.62 4.2	7 m 40.94 4.0
selectionSort	1 m 24.47 4.2	1 m 31.01 4.1	1 m 30.96 4.1	5 m 55.64 4.2	6 m 6.30 4.0	6 m 21.13 4.2
bubbleSort	1 m 27.55 4.2	2 m 59.40 4.0	4 m 15.68 4.1	6 m 6.91 4.2	12 m 27.15 4.2	17 m 14.02 4.0
ShellSort	0.11 —	.28 2.5	.16 2.7	.22 2.0	.66 2.4	.33 2.1
heapSort	.25 2.3	.27 1.6	.28 2.5	.55 2.2	.66 2.4	.49 1.7
mergesort	.16 1.5	.22 2.0	.22 2.0	.49 2.2	.49 2.2	.44 2.0
quicksort	.06 —	.11 2.2	.05 1.0	.11 1.8	.28 2.5	.17 2.8
quicksort2	.05 1.0	.11 1.8	.06 1.0	.11 2.2	.27 2.5	.11 1.8
radixSort	1.70 2.1	1.76 2.0	1.76 2.0	3.46 2.0	3.52 2.1	3.52 2.0
bitRadixSort	4.07 2.0	4.17 1.9	4.23 2.0	8.41 2.1	8.40 2.0	8.45 2.0
radixSort2	.66 2.0	.60 1.8	.65 2.0	1.38 2.1	1.32 2.2	1.43 2.2
bitRadixSort2	.94 2.1	.99 2.2	.99 2.3	1.97 2.1	2.08 2.1	1.98 2.0



Hashing

The main operation used by the searching methods described in the preceding chapters was comparison of keys. In a sequential search, the table that stores the elements is searched successively to determine which cell of the table to check, and the key comparison determines whether or not an element has been found. In a binary search, the table that stores the elements is divided successively into halves to determine which cell of the table to check, and again, the key comparison determines whether or not an element has been found. Similarly, the decision to continue the search in a binary search tree in a particular direction is accomplished by comparing keys.

A different approach to searching calculates the position of the key in the table based on the value of the key. The value of the key is the only indication of the position. When the key is known, the position in the table can be accessed directly, without making any other preliminary tests, as required in a binary search or when searching a tree. This means that the search time is reduced from $O(n)$, as in a sequential search, or from $O(\lg n)$, as in a binary search, to 1 or at least $O(1)$; regardless of the number of elements being searched, the run time is always the same. But this is just an ideal, and in real applications, this ideal can only be approximated.

We need to find a function h which can transform a particular key K, be it a string, number, record, etc., into an index in the table used for storing items of the same type as K. The function h is called a *hash function*. If h transforms different keys into different numbers, it is called a *perfect hash function*. To create a perfect hash function, which is always the goal, the table has to contain at least the same number of positions as the number of elements being hashed. But the number of elements is not always known ahead of time. For example, a compiler keeps all variables used in a program in a symbol table. Real programs use only a fraction of the vast number of possible variable names, so a table size of 1000 cells is usually adequate.

But even if this table can accommodate all the variables in the program, how can we design a function h which allows the compiler to immediately access the position associated with each variable? All the letters of the variable name can be added together and the sum can be used as an index. In this case, the table needs 3782 cells (for a variable K made out of 31 letters "z," $h(K) = 31 \cdot 122 = 3782$). But even with this size, the function h does not return unique values. For example, h("abc") = h("acb"). This problem is called *collision* and is discussed later. The worth of a hash function depends on how well it avoids collisions. Avoiding collisions can be achieved by making the function more sophisticated, but this sophistication should not go too far because the computational cost in determining $h(K)$ can be prohibitive, and less sophisticated methods may be faster.

10.1 Hash Functions

The number of hash functions that can be used to assign positions to n items in a table of m positions (for $n \leq m$) is equal to m^n. The number of perfect hash functions is the same as the number of different placements of these items in the table and is equal to $\frac{m!}{(m-n)!}$. For example, for 50 elements and a 100-cell array, there are $100^{50} = 10^{100}$ hash functions out of which "only" 10^{94} (one in a million) are perfect. Most of these functions are too unwieldy for practical applications and cannot be represented by a concise formula. However, even among functions that can be expressed with a formula, the number of possibilities is vast. This section discusses some specific types of hash functions.

10.1.1 Division

A hash function must guarantee that the number it returns is a valid index to one of the table cells. The simplest way to accomplish this is to use division modulo *TSize* = *sizeof*(*table*), as in $h(K) = K \bmod TSize$, if K is a number. It is best if *TSize* is a prime number. Otherwise, $h(K) = (K \bmod p) \bmod TSize$ for some prime $p > TSize$ can be used. However, nonprime divisors may work equally well as prime divisors provided they do not have prime factors less than 20 (Lum et al., 1971). The division method is usually the preferred choice for the hash function if very little is known about the keys.

10.1.2 Folding

In this method, the key is divided into several parts (which conveys the true meaning of the word *hash*). These parts are combined or folded together and are often transformed in a certain way to create the target address. There are two types of folding: *shift folding* and *boundary folding*.

The key is divided into several parts and these parts are then processed using a simple operation such as addition to combine them in a certain way. In shift folding, they are put underneath one another and then processed. For example, a social security number (SSN) 123–45–6789 can be divided into three parts, 123, 456, 789, and

then these parts can be added. The resulting number, 1368, can be divided modulo *TSize* or, if the size of the table is 1000, the first three digits can be used for the address. To be sure, the division can be done in many different ways. Another possibility is to divide the same number 123-45-6789 into five parts (say, 12, 34, 56, 78, and 9), add them, and divide the result modulo *TSize*.

With boundary folding, the key is seen as being written on a piece of paper which is folded on the borders between different parts of the key. In this way, every other part will be put in the reverse order. Consider the same three parts of the SSN: 123, 456, and 789. The first part, 123, is taken in the same order, then the piece of paper with the second part is folded underneath it so that 123 is aligned with 654, which is the second part, 456, in reverse order. When the folding continues, 789 is aligned with the two previous parts. The result is 123+654+789=1566.

In both versions, the key is usually divided into even parts of some fixed size plus some remainder and then added. This process is simple and fast, especially when bit patterns are used instead of numerical values. A bit-oriented version of shift folding is obtained by applying the exclusive or operation, ^.

In the case of strings, one approach processes all characters of the string by "xor'ing" them together and using the result for the address. For example, for the string "abcd," *h*("abcd") = "a"^"b"^"c"^"d." However, this simple method results in addresses between the numbers 0 and 127. For better result, chunks of strings are "xor'ed" together rather than single characters. These chunks are composed of the number of characters equal to the number of bytes in an integer. Since an integer in a C++ implementation for the IBM PC computer is two bytes long, *h*("abcd") = "ab" xor "cd" (most likely divided modulo *TSize*). Such a function is used in the case study in this chapter.

10.1.3 Mid-Square Function

In the mid-square method, the key is *squared* and the middle or *mid* part of the result is used as the address. If the key is a string, it has to be preprocessed to produce a number by using, for instance, folding. In a mid-square hash function, the entire key participates in generating the address so that there is a better chance that different addresses are generated for different keys. For example, if the key is 3121, then 3121^2 = 9740641, and for the 1000-cell table, *h*(3121) = 406, which is the middle part of 3121^2. In practice, it is more efficient to choose a power of 2 for the size of the table and extract the middle part of the bit representation of the square of a key. If we assume that the size of the table is 1024, then, in this example, the binary representation of 3121^2 is the bit string 100101001010000101100001 with the middle part shown in italics. This middle part, the binary number 0101000010, is equal to 322. This part can easily be extracted by using a mask and a shift operation.

10.1.4 Extraction

In the extraction method, only a part of the key is used to compute the address. For the social security number 123–45–6789, this method might use the first four digits, 1234, the last four, 6789, the first two combined with the last two, 1289, or some other combination. Each time, only a portion of the key is used. If this portion is carefully

chosen, it can be sufficient for hashing provided the omitted portion distinguishes the keys only in an insignificant way. For example, in some university settings, all international students' ID numbers start with 999. Therefore, the first three digits can be safely omitted in a hash function which uses student IDs for computing table positions. Similarly, the starting digits of the ISBN code are the same for all books published by the same publisher (e.g., 0534 for Brooks/Cole Publishing Company). Therefore, they should be excluded from the computation of addresses if a data table contains only books from one publisher.

10.1.5 Radix Transformation

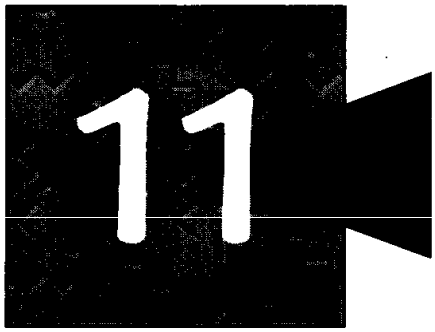
Using the radix transformation, the key K is transformed into another number base; K is expressed in a numerical system using a different radix. If K is the decimal number 345, then its value in base 9 (nonal) is 423. This value is then divided modulo *TSize*, and the resulting number is used as the address of the location to which K should be hashed. Collisions, however, cannot be avoided. If *TSize* = 100, then although 345 and 245 (decimal) are not hashed to the same location, 345 and 264 are because 264 decimal is 323 in the nonal system, and both 423 and 323 return 23 when divided modulo 100.

10.2 Collision Resolution

Note that straightforward hashing is not without its problems, since for almost all hash functions, more than one key can be assigned to the same position. For example, if the hash function h_1 applied to names returns the ASCII value of the first letter of each name, i.e., $h_1(name) = name[0]$, then all names starting with the same letter are hashed to the same position. This problem can be solved by finding a function which distributes names more uniformly in the table. For example, the function h_2 could add the first two letters, i.e., $h_2(name) = name[0] + name[1]$, which is better than h_1. But even if all the letters are considered, i.e., $h_3(name) = name[0] + \cdots + name[\texttt{strlen}(name) - 1]$, the possibility of hashing different names to the same location still exists. The function h_3 is the best of the three because it distributes the names most uniformly for the three defined functions, but it also tacitly assumes that the size of the table has been increased. If the table had only 26 positions, which is the number of different values returned by h_1, there is no improvement using h_3 instead of h_1. Therefore, one more factor can contribute to avoiding conflicts between hashed keys, namely, the size of the table. Increasing this size may lead to better hashing, but not always! These two factors—hash function and table size—may minimize the number of collisions, but they cannot completely eliminate them. The problem of collision has to be dealt with in a way that always guarantees a solution.

There are scores of strategies that attempt to avoid hashing multiple keys to the same location. Only a handful of these methods are discussed in this chapter.

Data Compression



Transfer of information is essential for the proper functioning of any structure on any level and any type of organization. The faster an exchange of information occurs, the smoother the structure functions. Improvement of the rate of transfer can be achieved by improving the medium through which data are transferred or by changing the data themselves so that the same information can be transmitted within a shorter time interval.

Information can be represented in a form which exhibits some redundancy. For example, in a database, it is enough to say about a person that he is "M" or she is "F," instead of spelling out the whole words, "male" and "female," or to use 1 and 2 to represent the same information. The number one hundred twenty-eight can be stored as 80 (hexadecimal), 128, 1000000 (binary), CXXVIII, $\rho\kappa\eta$ (the Greek language used letters as digits), or | | | . . . | (128 bars). If numbers are stored as the sequences of digits representing them, then 80 is the shortest form. Numbers are represented in binary form in computers.

11.1 Conditions for Data Compression

When transferring information, the choice of the data representation determines how fast the transfer is performed. A judicious choice can improve the throughput of a transmission channel without changing the channel itself. There are many different methods of *data compression* (or *compaction*) that reduce the size of the representation without affecting the information itself.

Assume that there are n different symbols used to code messages. For a binary code, $n = 2$; for morse code, $n = 3$: the dot, the dash, and the blank separating the se-

quences of dots and dashes that represent letters. Assume also that all symbols m_i forming a set M have been independently chosen and are known to have probabilities of occurrence $P(m_i)$, and the symbols are coded with strings of 0s and 1s. Then $P(m_1) + \cdots + P(m_n) = 1$. The information content of the set M, called the *entropy* of the source M, is defined by

$$L_{ave} = P(m_1)L(m_1) + \cdots + P(m_n)L(m_n) \tag{11.1}$$

where $L(m_i) = -\lg(P(m_i))$, which is the minimum length of a codeword for symbol m_i. Claude E. Shannon established in 1948 that Equation 11.1 gives the best possible average length of a codeword when the source symbols and the probabilities of their use are known. No data compression algorithm can be better than L_{ave}, and the closer it is to this number, the better is its compression rate.

For example, if there are three symbols m_1, m_2, and m_3 with the probabilities .25, .25, and .5, respectively, then the lengths of the codewords assigned to them are:

$$-\lg(P(m_1)) = -\lg(P(m_2)) = -\lg(.25) = \lg\left(\frac{1}{.25}\right) = \lg(4) = 2 \text{ and}$$

$$-\lg(P(m_3)) = \lg(2) = 1$$

and the average length of a codeword is

$$L_{ave} = P(m_1) \cdot 2 + P(m_2) \cdot 2 + P(m_3) \cdot 1 = 1.5$$

Various data compression techniques attempt to minimize the average codeword length by devising an optimal code (that is, an assignment of codewords to symbols that depends on the probability P with which a symbol is being used. If a symbol is issued infrequently, it can be assigned a long codeword. For frequently issued symbols, very short encodings are more to the point.

Some restrictions need to be imposed on the prospective codes:

1. Each codeword corresponds to exactly one symbol.
2. Decoding should not require any look ahead; after reading each symbol it should be possible to determine whether the end of a string encoding a symbol of the original message has been reached. A code meeting this requirement is called a code with the *prefix property*, and it means that no codeword is a prefix of another codeword. Therefore, no special punctuation is required to separate two codewords in a coded message.

The second requirement can be illustrated by three different encodings of three symbols as given in the following table:

Symbol	Code_1	Code_2	Code_3
A	1	1	11
B	2	22	12
C	12	12	21

The first code does not allow us to make a distinction between AB and C; both are coded as 12. The second code does not have this ambiguity, but it requires look ahead, as in 1222: The first 1 can be decoded as A. The following 2 shows that A was improperly chosen, and 12 should have been decoded

A is a proper choice if the third symbol is 2. Since 2 is found, AB is chosen as the tentatively decoded string, but the fourth symbol is another 2. Hence, the first turn was wrong, and A has been ill-chosen. The proper decoding is CB. All these problems arise because both $code_1$ and $code_2$ violate the prefix property. Only $code_3$ can be unambiguously decoded as read.

For an optimal code, two more stipulations are specified.

3. The length of the codeword for a given symbol m_j should not exceed the length of the codeword of a less probable symbol m_i; that is, if $P(m_i) \leq P(m_j)$, then $L(m_j) \geq L(m_i)$ for $1 \leq i, j \leq n$.
4. In an optimal encoding system, there should not be any unused short codewords either as stand-alone encodings or as prefixes for longer codewords, since this would mean that longer codewords were created unnecessarily. For example, the sequence of codewords 01, 000, 001, 100, 101 for a certain set of five symbols is not optimal because the codeword 11 is not used anywhere; this encoding can be turned into an optimal sequence 01, 10, 11, 000, 001.

In the following sections, several data compression methods are presented. To compare the efficiency of these methods when applied to the same data, the same measure is used. This measure is the *compression rate* (also called the *fraction of data reduction*), and it is defined as the ratio

$$\frac{\text{length(input)} - \text{length(output)}}{\text{length(input)}} \tag{11.2}$$

It is expressed as a percentage indicating the amount of redundancy removed from the input.

11.2 Huffman Coding

The construction of an optimal code was developed by David Huffman, who utilized a tree structure in this construction: a binary tree for a binary code. The algorithm is surprisingly simple and can be summarized as follows:

```
Huffman()
    for each symbol create a tree with a single root node and order all trees
        according to the probability of symbol occurrence;
    while more than one tree is left
        take the two trees t1, t2 with the lowest probabilities p1, p2 (p1 ≤ p2)
            and create a tree with t1 and t2 as its children and with
            the probability in the new root equal to p1 + p2;
    associate 0 with each left branch and 1 with each right branch;
    create a unique codeword for each symbol by traversing the tree from the root
        to the leaf containing the probability corresponding to this
        symbol and by putting all encountered 0s and 1s together;
```

The resulting tree has a probability of 1 in its root.

It should be noted that the algorithm is not deterministic in the sense of producing a unique tree because, for trees with equal probabilities in the roots, the algorithm does not prescribe their positions with respect to each other either at the beginning or during execution. If t_1 with probability p_1 is in the sequence of trees and the new tree t_2 is created with $p_2 = p_1$, should t_2 be positioned to the left of t_1 or to the right? Also, if there are three trees t_1, t_2, and t_3 with the same lowest probability in the entire sequence, which two trees should be chosen to create a new tree? There are three possibilities for choosing two trees. As a result, different trees can be obtained depending on where the trees with equal probabilities are placed in the sequence with respect to each other. Regardless of the shape of the tree, the average length of the codeword remains the same.

To assess the compression efficiency of the Huffman algorithm, a definition of the *weighted path length* is used, which is the same as Equation 11.1 except that $L(m_i)$ is interpreted as the number of 0s and 1s in the codeword assigned to symbol m_i by this algorithm.

Figure 11.1 contains an example for the five letters A, B, C, D, and E with probabilities .39, .21, .19, .12, and .09, respectively. The tree in Figures 11.1a and 11.1b are different in the way in which the two nodes containing a probability of .21 have been chosen to be combined with tree .19 to create a tree of .40. Regardless of the choice, the lengths of the codewords associated with the five letters A through E are the same, namely, 2, 2, 2, 3, and 3, respectively. However, the codewords assigned to them are slightly different, as shown in Figures 11.1c and 11.1d, which present abbreviated (and more commonly used) versions of the way the trees in Figures 11.1a and 11.1b were created. The average length for the latter two trees is

$$L_{\text{Huf}} = .39 \cdot 2 + .21 \cdot 2 + .19 \cdot 2 + .12 \cdot 3 + .09 \cdot 3 = 2.21$$

which is very close to 2.09 (only 5% off), the average length computed according to Equation 11.1:

$$L_{\text{ave}} = .39 \cdot 1.238 + .21 \cdot 2.252 + .19 \cdot 2.396 + .12 \cdot 3.059 + .09 \cdot 3.474 = 2.09$$

Corresponding letters in Figures 11.1a and 11.1b have been assigned codewords of the same length. Obviously, the average length for both trees is the same. But each way of building a Huffman tree, starting from the same data, should result in the same average length, regardless of the shape of the tree. Figure 11.2 shows two Huffman trees for the letters P, Q, R, S, and T with the probabilities .1, .1, .1, .2 and .5, respectively. Depending on how the lowest probabilities are chosen, different codewords are assigned to these letters with different lengths, at least for some of them. However, the average length remains the same and is equal to 2.0.

The Huffman algorithm can be implemented in a variety of ways, at least as many as the number of ways a priority queue can be implemented. The priority queue is the natural data structure in the context of the Huffman algorithm since it requires removing the two smallest probabilities and inserting the new probability in the proper position.

One way to implement this algorithm is to use a singly linked list of pointers to trees, which reflects closely what Figure 11.1a illustrates. The linked list is initially ordered according to the probabilities stored in the trees, all of them consisting of just a

FIGURE **11.1** Two Huffman trees created for five letters A, B, C, D, and E with probabilities .39, .21, .19, .12, and .09.

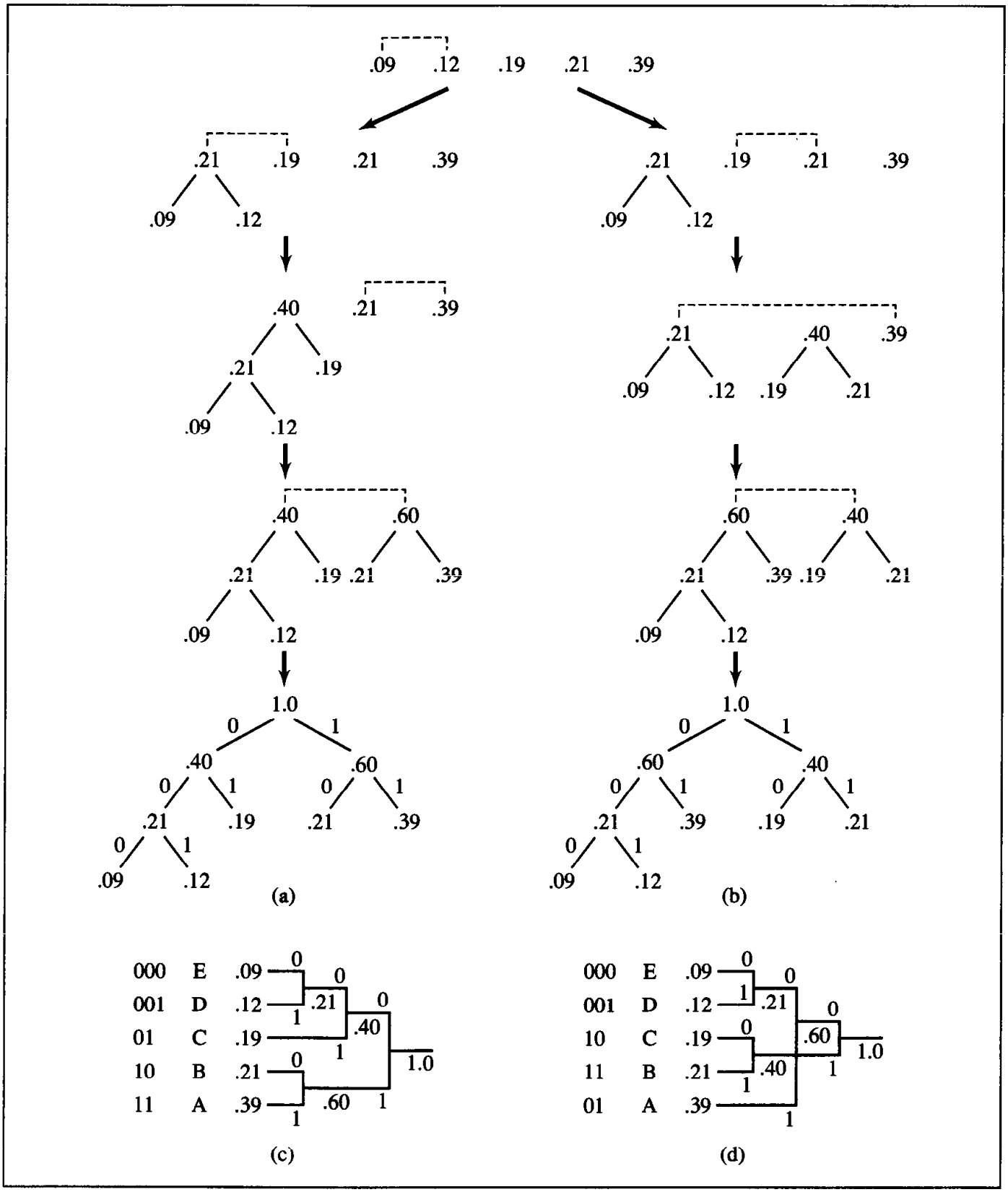
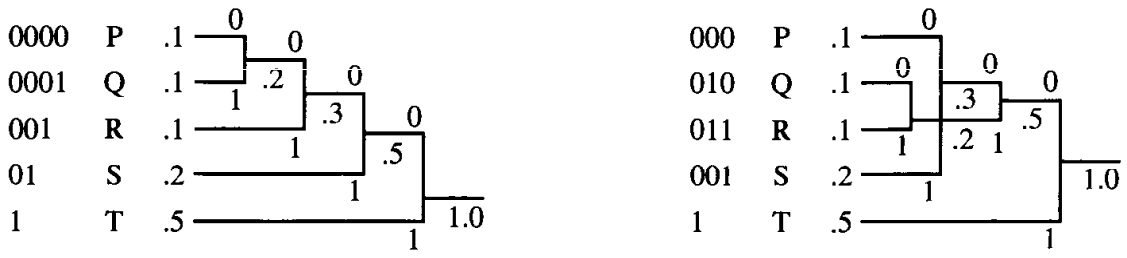


FIGURE **11.2** Two Huffman trees generated for letters P, Q, R, S, and T with probabilities .1, .1, .1, .2, and .5.



root. Then, repeatedly, the two trees with the smallest probabilities are chosen; the tree with the smaller probability is replaced by a newly created tree, and the node with the pointer to the tree with the higher probability is removed from the linked list. From trees having the same probability in their roots, the first tree encountered is chosen.

In another implementation, all probability nodes are first ordered, and that order is maintained throughout the operation. From such an ordered list, the first two trees are always removed to create a new tree from them, which is inserted close to the end of the list. To that end, a doubly linked list of pointers to trees with immediate access to the beginning and to the end of this list can be used. Figure 11.3 contains a trace of the execution of this algorithm for the letters A, B, C, D, and E with the same probabilities as in Figure 11.1. Codewords assigned to these letters are also indicated in Figure 11.3. Note that they are different from the codewords in Figure 11.1, although their lengths are the same.

The two preceding algorithms built Huffman trees bottom-up by starting with a sequence of trees and collapsing them together to a gradually smaller number of trees and, eventually, to one tree. However, this tree can be built top-down, starting from the highest probability. But only the probabilities to be placed in the leaves are known. The highest probability, to be put in the root, is known if lower probabilities, in the root's children, have been determined; the latter are known if still lower probabilities have been computed and so on. Therefore, creating nonterminal nodes has to be deferred until the probabilities to be stored in them are found. It is very convenient to use the following recursive algorithm to implement a Huffman tree:

```
createHuffmanTree(prob)
    declare the probabilities p1, p2, and the Huffman tree Htree;
    if only two probabilities are left in prob
        return a tree with p1, p2 in the leaves and p1 + p2 in the root;
    else remove the two smallest probabilities from prob and assign them to p1 and p2;
        insert p1 + p2 to prob;
        Htree = CreateHuffmanTree(prob);
        in Htree make the leaf with p1 + p2 the parent of two leaves with p1 and p2;
        return Htree;
```

FIGURE **11.3** Using a doubly linked list to create the Huffman tree for the letters from Figure 11.1.

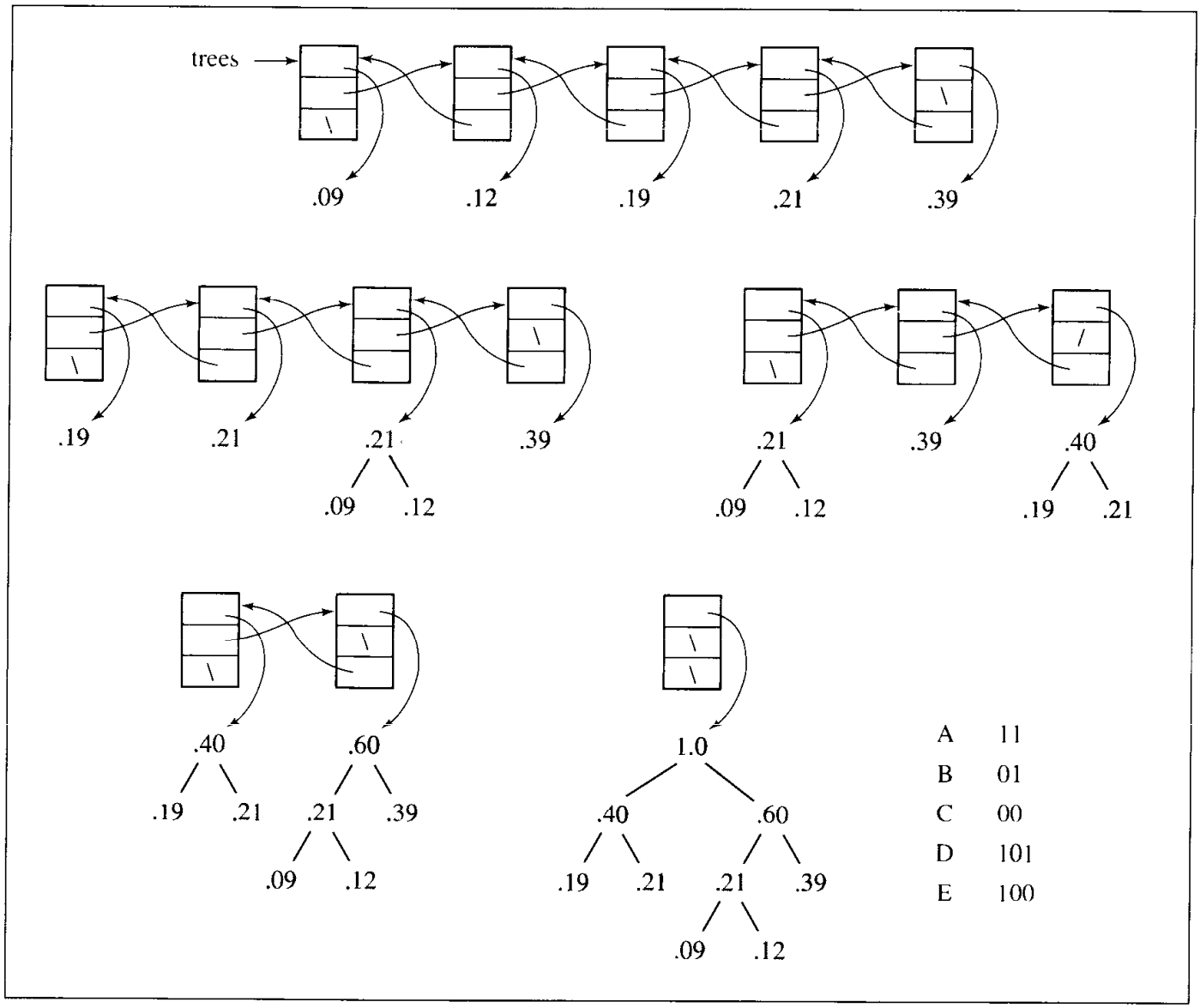
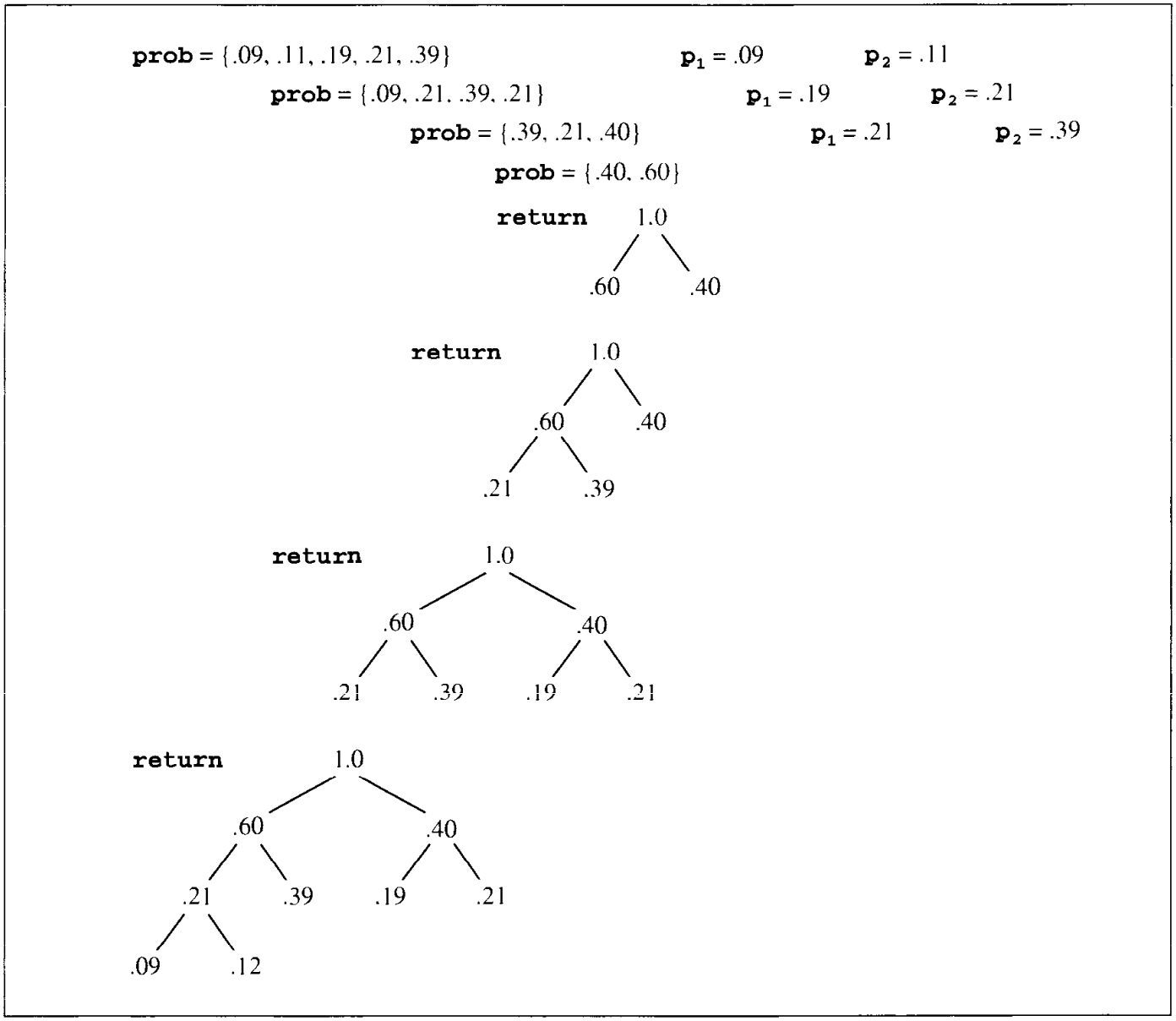


Figure 11.4 contains a summary of the trace of the execution of this algorithm for the letters A, B, C, D, and E with the probabilities as shown in Figure 11.1. Indentation indicates consecutive calls to `createHuffmanTree()`.

One implementation of a priority queue is a min heap which can also be used to implement this algorithm. In this heap, each nonterminal node has a smaller probability than the probabilities in its children, and because the smallest probability is in the root, that one is simple to remove. But after it is removed, the root is empty. Therefore, the largest element is put in the root and the heap property is restored. Then the second element can be removed from the root and replaced with a new element which represents the sum of the probability of the root and the probability previously removed.

FIGURE **11.4** Top-down construction of a Huffman tree using recursive implementation.



Afterward, the heap property has to be restored again. After one such sequence of operations, the heap has one less node: Two probabilities from the previous heap have been removed and a new one has been added. But it is not enough to create the Huffman tree: The new probability is a parent of the probabilities just removed, and this information must be retained. To that end, three arrays can be used: *indexes* containing the indexes of the original probabilities and the probabilities created during the process of creating the Huffman tree; *probabilities*, an array of the original and newly created probabilities; and *parents*, an array of indexes indicating the position of the parents of the elements stored in *probabilities*. A positive number in *parents* indicates the left child, and a negative number indicates the right child. Codewords are created

by accumulating 0s and 1s when going from leaves to the root using the array *parents*, which functions as an array of pointers. It is important to note that in this particular implementation probabilities are sorted indirectly: The heap is actually made up of indexes to probabilities, and all exchanges take place in *indexes*.

Figure 11.5 illustrates an example of using a heap to implement the Huffman algorithm. The heaps in steps (a), (e), (i), and (m) in Figure 11.5 are ready for processing. First, the highest probability is put in the root, as in steps (b), (f), (j), and (n) of Figure 11.5. Next, the heap is restored, as in steps (c), (g), (k), and (o), and the root probability is set to the sum of the two smallest probabilities, as in steps (d), (h), (l), and (p). Processing is complete when there is only one node in the heap.

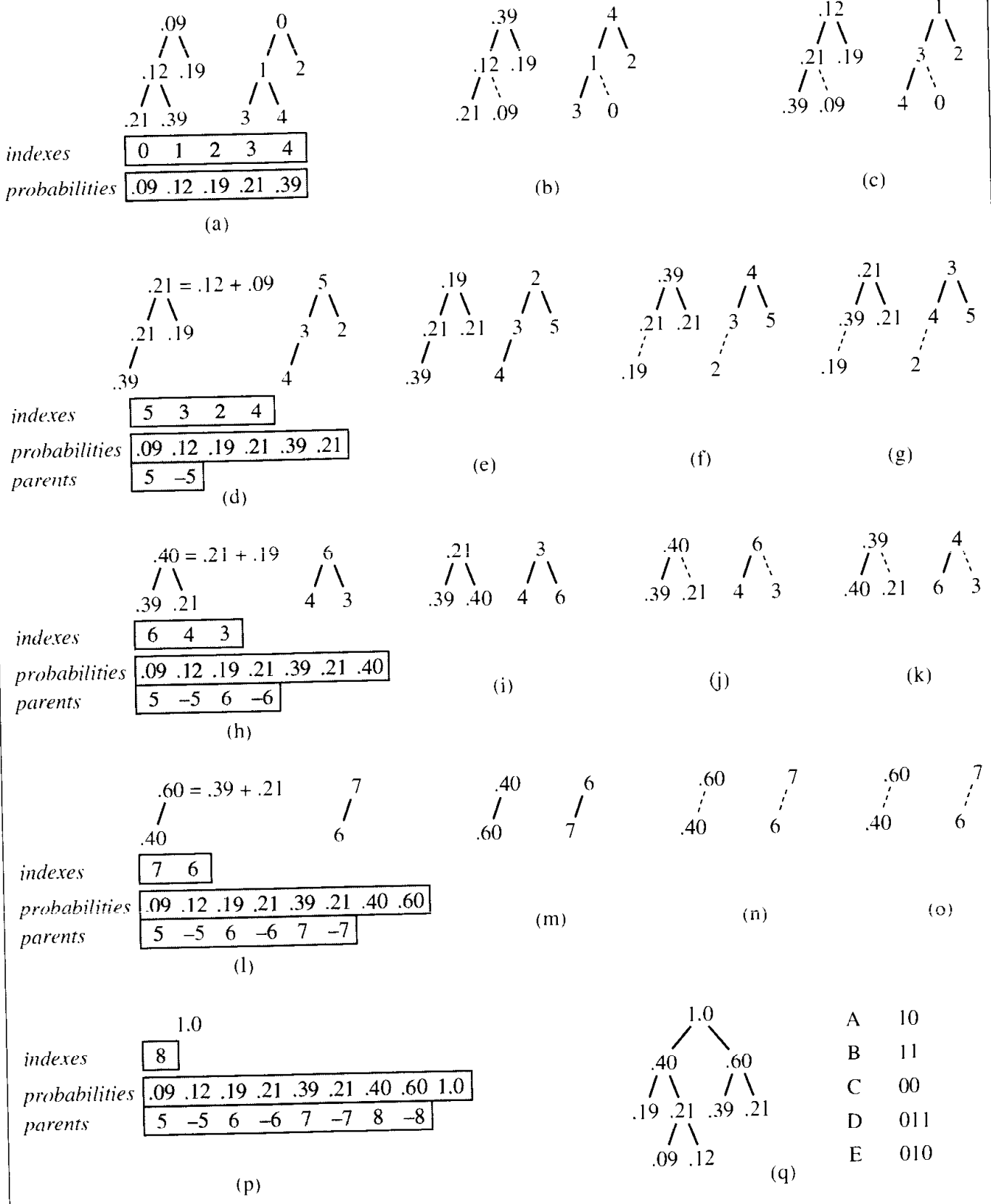
Using the Huffman tree, a table can be constructed which gives the equivalents for each symbol in terms of 1s and 0s encountered along the path leading to each of the leaves of the tree. For our example, the tree from Figure 11.3 will be used, and the resulting table is

A	11
B	01
C	00
D	101
E	100

The coding process transmits coded equivalents of the symbols to be sent. For example, instead of sending ABAAD, the sequence 11011111101 is dispatched with the average number of bits per one letter equal to 11/5 = 2.2, almost the same as 2.09, the value specified by the formula for L_{ave}. To decode this message, the conversion table has to be known to the message receiver. Using this table, a Huffman tree can be constructed with the same paths as the tree used for coding, but its leaves would (for the sake of efficiency) store the symbols instead of their probabilities. In this way, upon reaching a leaf, the symbol can be retrieved directly from it. Using this tree, each symbol can be decoded uniquely. For example, if 1001101 is received, then we try to reach a leaf of the tree using the path indicated by leading 1s and 0s. In this case, 1 takes us to the right, 0 to the left, and another 0 again to the left, whereby we end up in a leaf containing E. After reaching this leaf, decoding continues by starting from the root of the tree and trying to reach a leaf using the remaining 0s and 1s. Since 100 has been processed, 1101 has to be decoded. Now, 1 takes us to the right and another 1 again to the right, which is a leaf with A. We start again from the root, and the sequence 01 is decoded as B. The entire message is now decoded as EAB.

At this point, a question can be asked: Why send 11011111101 instead of ABAAD? This is supposed to be data compression, but the coded message is twice as long as the original. Where is the advantage? Note precisely the way in which messages are sent. A, B, C, D, and E are single letters, and letters, being characters, require one byte (eight bits) to be sent, using the extended ASCII code. Therefore, the message ABAAD requires five bytes (40 bits). On the other hand, 0s and 1s in the coded version can be sent as single bits. Therefore, if 11011111101 is regarded not as a sequence of the characters "0" and "1," but as a sequence of bits, then only 11 bits are needed to send the message, about one-fourth of what is required to send the message in its original form, ABAAD.

FIGURE **11.5** Huffman algorithm implemented with a heap.



This example raises one problem: Both the encoder and the decoder have to use the same coding, the same Huffman tree. Otherwise, the decoding will be unsuccessful. How can the encoder let the decoder know which particular code has been used? There are at least three possibilities:

1. Both the encoder and decoder agree beforehand on a particular Huffman tree and both use it for sending any message.
2. The encoder constructs the Huffman tree afresh every time a new message is sent and sends the conversion table along with the message. The decoder either uses the table to decode the message or reconstructs the corresponding Huffman tree and then performs the translation.
3. The decoder constructs the Huffman tree during transmission and decoding.

The second strategy is more versatile, but its advantages are visible only when large files are encoded and decoded. For our simple example, ABAAD, sending both the table of codewords and the coded message 11011111101 is hardly perceived as data compression. However, if a file contains a message of 10,000 characters using the characters A through E, then the space saved is significant. Using the probabilities indicated earlier for these letters, we project that there are approximately 3900 As, 2100 Bs, 1900 Cs, 1200 Ds, and 900 Es. Hence, the number of bits needed to code this file is

$$3900 \cdot 2 + 2100 \cdot 2 + 1900 \cdot 2 + 1200 \cdot 3 + 900 \cdot 3 = 22{,}100 \text{ bits} = 2762.5 \text{ bytes}$$

which is approximately one-fourth the 10,000 bytes required for sending the original file. Even if the conversion table is added to the file, this proportion is only minimally affected.

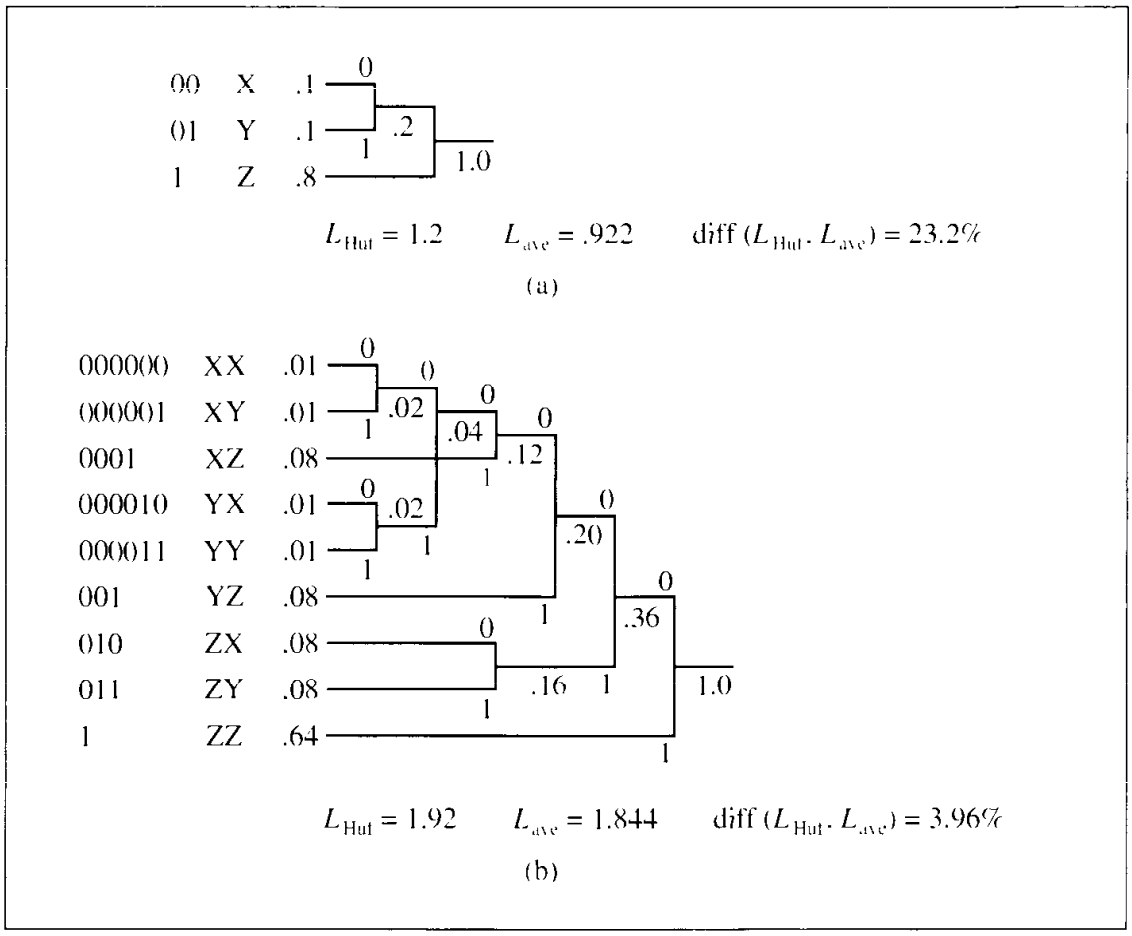
However, even with this approach, there may be some room for improvement. As indicated, an ideal compression algorithm should give the same average codeword length as computed from Equation 11.1. The symbols from Figure 11.1 have been assigned codewords whose average length is 2.21, approximately 5% worse than the ideal 2.09. Sometimes, however, the difference is larger. Consider, for example, three symbols X, Y, and Z with probabilities .1, .1, and .8. Figure 11.6a shows a Huffman tree for these symbols, with codewords assigned to them. The average length, according to this tree, is

$$L_{\text{Huf}} = 2 \cdot .1 + 2 \cdot .1 + 1 \cdot .8 = 1.2$$

and the best expected average, L_{ave}, is .922. Therefore, there is a possibility we can improve the Huffman coding by approximately 23.2%, ignoring the fact that, at this point, a full 23.2% improvement is not possible because the average is below 1. How is this possible? As already stated, all Huffman trees result in the same average weighted path length. Therefore, no improvement can be expected if only the symbols X, Y, and Z are used to construct this tree.

On the other hand, if all possible pairs of symbols are used for building a Huffman tree, the data rate can be reduced. Figure 11.6b illustrates this procedure. Out of three symbols X, Y, and Z, nine pairs are created whose probabilities are computed by multiplying the probability of both symbols. For example, since the probability for both X and Y is .1, the probability of pair XY is .01 = .1 · .1. The average L_{Huf} is 1.92

FIGURE **11.6** Improving the average length of the codeword by applying the Huffman algorithm (b) to pairs of letters (a) instead of single letters.



and the expected average L_{ave} is 1.84 (twice the previous L_{ave}), with the difference between these averages being 4%. This represents a 19.2% improvement at the cost of including a larger conversion table (nine entries instead of three) as part of the message to be sent. If the message is large and the number of symbols used in the message is relatively small, then the increase in the size of the table is insignificant. However, for a large number of symbols, the size of the table may be much too large to notice any improvement. For 26 English letters, the number of pairs is 676 which is considered relatively small. But if all printable characters have to be distinguished in an English text, from the blank character (ASCII code 32), to the tilde (ASCII code 126), plus the carriage return character, then there are $(126 - 32 + 1) + 1 = 96$ characters and 9216 pairs of characters. Many of these pairs are not likely to occur at all (e.g., XQ or KZ), but even if 50% of them are found, the resulting table containing these pairs along with codewords associated with them may be too large to be useful.

Using pairs of symbols is still a good idea, even if the number of symbols is large. For example, a Huffman tree can be constructed for all symbols and for all pairs of symbols that occur at least five times. The efficiency of the variations of Huffman encoding can be measured by comparing the size of compressed files. Experiments were

performed on an English text, PL/1 program file, and a digitized photographic image (Rubin 1976). When only single characters were used, the compression rates were approximately 40%, 60%, and 50%, respectively. When single characters were used along with the 100 most frequent groups (not only two characters long), the compression rates were 49%, 73%, and 52%. When the 512 most frequent groups were used, the compression rates were around 55%, 71%, and 62%.

11.2.1 Adaptive Huffman Coding

The foregoing discussion assumed that the probabilities of messages are known in advance. A natural question is: How do we know them?

One solution computes the number of occurrences of each symbol expected in messages in some fairly large sample of texts of, say, 10 million characters. For messages in natural languages such as English, such samples may include some literary works, newspaper articles, and a portion of an encyclopedia. After each character's frequency has been determined, a conversion table can be constructed for use by both the sending and receiving ends of the data transfer. This eliminates the need to include such a table every time a file is transmitted.

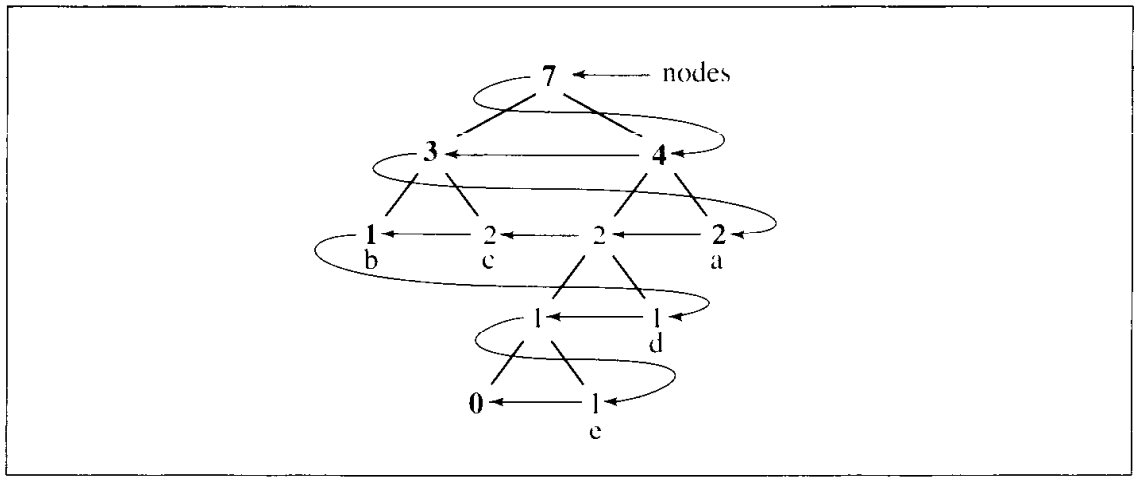
However, this method may not be useful for sending some specialized files, even if written in English. A computer science paper includes a much higher percentage of digits and parentheses, especially if it includes extensive illustrations in LISP or C++ code, than a paper on the prose of Jane Austen. In such circumstances, it is more judicious to use the text to be sent to determine the needed frequencies, which also requires enclosing the table as overhead in the file being sent. A preliminary pass through this file is required before an actual conversion table can be constructed. However, the file to be preprocessed may be very large, and preprocessing slows down the entire transmission process. Second, the file to be sent may not be known in its entirety when it is being sent, and yet compression is necessary: For example, when a text is being typed and sent line by line, then there is no way to know the contents of the whole file at the time of sending. In such a situation, adaptive compression is a viable solution.

An adaptive Huffman encoding technique was devised first by Robert G. Gallager and then improved by Donald Knuth. The algorithm is based on the following *sibling property:* If each node has a sibling (except for the root) and the breadth-first right-to-left tree traversal generates a list of nodes with nonincreasing frequency counters, it can be proven that a tree with the sibling property is a Huffman tree (Faller 1974, Gallager 1978).

In adaptive Huffman coding, the Huffman tree includes a counter for each symbol, and the counter is updated every time a corresponding input symbol is being coded. Checking whether the sibling property is retained assures that the Huffman tree under construction is still a Huffman tree. If the sibling property is violated, the tree has to be restructured to restore this property. Here is how this is accomplished.

First, it is assumed that the algorithm maintains a doubly linked list `nodes` that contains the nodes of the tree ordered by breadth-first right-to-left tree traversal. A $block_i$ is a part of the list where each node has frequency i, and the first node in each block is called a *leader*. For example, Figure 11.7 shows the Huffman tree and also the list `nodes` = (**7 4 3 2** 2 2 1 1 1 1 **0**) that has six blocks—$block_7$, $block_4$, $block_3$, $block_2$, $block_1$, and $block_0$—with leaders shown with counters in boldface.

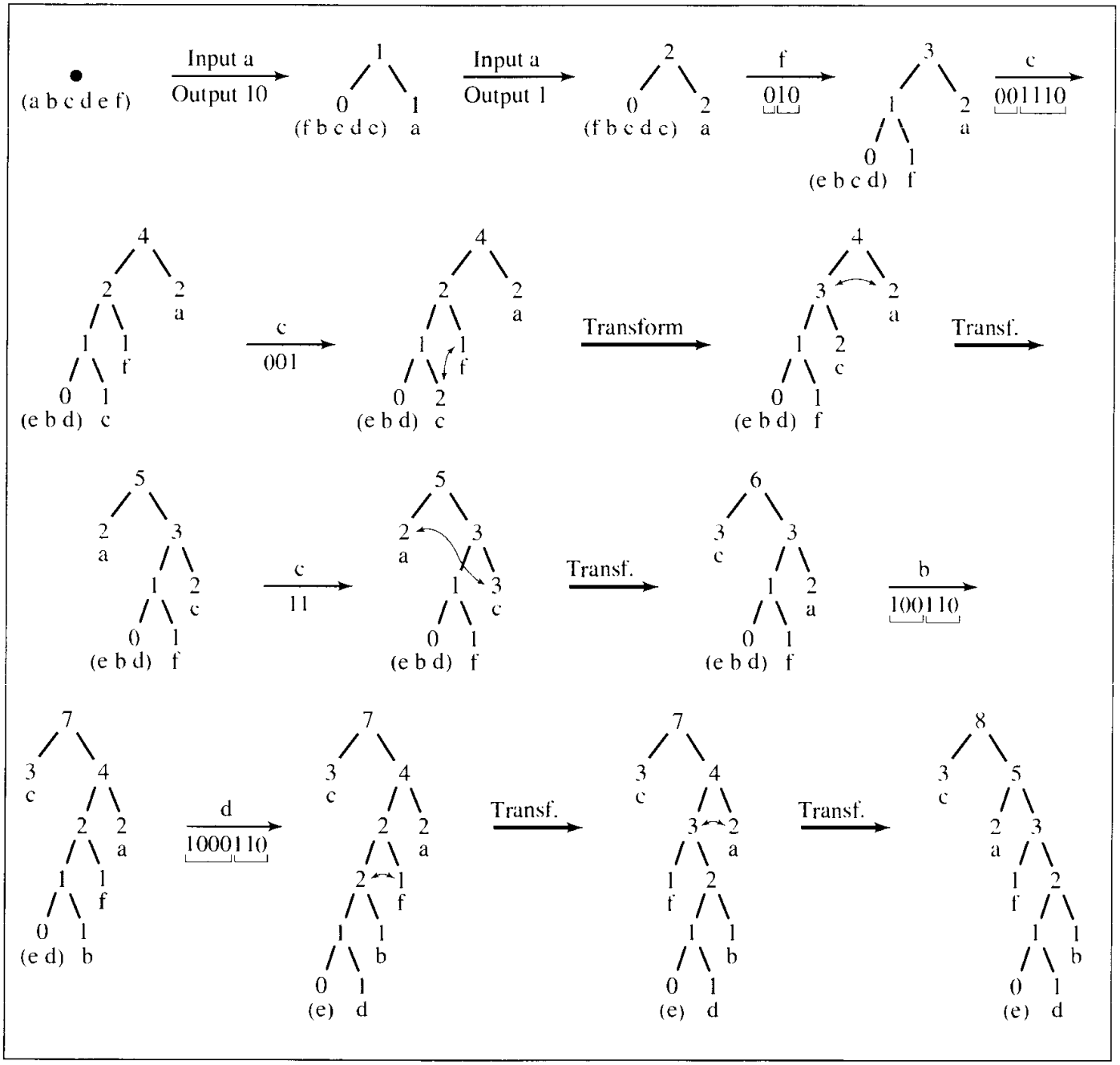
FIGURE **11.7** Doubly-linked list nodes formed by breadth-first right-to-left tree traversal.



All unused symbols are kept in one node with a frequency of 0, and each symbol encountered in the input has its own node in the tree. Initially, the tree has just one 0-node that includes all symbols. If an input symbol did not yet appear in the input, the 0-node is split in two, with the new 0-node containing all symbols except the newly encountered one and the node referring to this new symbol with counter set to 1; both nodes become children of the one parent whose counter is also set to 1. If an input symbol already has a node `p` in the tree, its counter is incremented. However, such an increment may endanger the sibling property, so this property has to be restored by exchanging the node `p` with the leader of the block to which `p` currently belongs, except when this leader is `p`'s parent. This node is found by going in `nodes` from `p` toward the beginning of this list. If `p` belongs to $\texttt{block}_i$ before increment, it is swapped with the leader of this block, whereby it is included in $\texttt{block}_{i+1}$. Then the counter increment is done for the `p`'s possibly new parent, which may also lead to a tree transformation to restore the sibling property. This process is continued until the root is reached. In this way, the counters are updated on the *new* path from `p` to the root rather than on its old path. For each symbol, the codeword is issued, which is obtained by scanning the Huffman tree from the root to the node corresponding to this symbol *before* any transformation in the tree takes place.

There are two different types of codewords transmitted during this process. If a symbol being coded has already appeared, then the normal coding procedure is applied: The Huffman tree is scanned from the root to the node holding this symbol to determine its codeword. If a symbol appears in the input for the first time, it is in the 0-node, but just sending the Huffman codeword of the 0-node does not suffice. Therefore, along with the codeword allowing us to reach the 0-node, the codeword is sent which indicates the position of the encountered symbol. For the sake of simplicity, we assume that position n is coded as n 1s followed by a 0. Zero is used to separate the 1s from those belonging to the next codeword. For example, when the letter c is coded for the first time, its codeword, 001110, is a combination of the codeword for the 0-node 00, and the codeword 1110 indicating that c can be found in the third position in the

FIGURE **11.8** Transmitting the message "aafcccbd" using an adaptive Huffman algorithm.



list of unused symbols associated with 0-node. These two codewords (or rather, parts of one codeword) are marked in Figure 11.8 by underlining them separately. After a symbol is removed from the list in 0-node, its place is taken by the last symbol of this list. This also indicates that the encoder and receiver have to agree on the alphabet being used and its ordering. The algorithm is shown in this pseudocode:

```
FGKDynamicHuffmanEncoding(symbol s)
    p = leaf that contains symbol s;
    c = Huffman codeword for s;
```

```
if p is the 0-node
  c = c concatenated with the number of 1s representing position of s in 0-node and with 0;
  write the last symbol in 0-node over s in this node;
  create a new node q for symbol s and set its counter to 1;
  p = new node to become the parent of both 0-node and node q;
  counter(p) = 1;
  include the two new nodes to nodes;
else increment counter(p);
while p is not the root
  if p violates the sibling property
    if the leader of the block_i that still includes p is not parent(p)
      swap p with the leader;
  p = parent(p);
  increment counter(p);
return codeword c;
```

A step-by-step example for string *aafcccbd* is shown in Figure 11.8.

1. Initially, the tree includes only the 0-node with all the source letters (*a, b, c, d, e, f*). After the first input letter, *a*, only the codeword for the position occupied by *a* in the 0-node is output. Because it is the first position, one 1 is output followed by a 0. The last letter in the 0-node is placed in the first position, and a separate node is created for the letter *a*. The node, with the frequency count set to 1, becomes a child of another new node that is also the parent of the 0-node.
2. After the second input letter, also an *a*, 1 is output, which is the Huffman codeword for the leaf that includes *a*. The frequency count of *a* is incremented to 2, which violates the sibling property, but because the leader of the block is the parent of node p (that is, of node *a*), no swap takes place; only p is updated to point to its parent, and then p's frequency count is incremented.
3. The third input letter, *f*, is a letter output for the first time; thus, the Huffman codeword for the 0-node, 0, is generated first, followed by the number of 1s corresponding to the position occupied by *f* in the 0-node, followed by 0: 10. The letter *e* is put in place of letter *f* in the 0-node, a new leaf for *f* is created, and a new node becomes the parent of the 0-node and the leaf just created. Node p, which is the parent of leaf *f*, does not violate the sibling property, so p is updated, p = *parent*(p), thereby becoming the root that is incremented.
4. The fourth input letter is *c*, which appears for the first time in the input. The Huffman codeword for the 0-node is generated followed by three 1s and a 0 because *c* is the third letter in the 0-node. After that, *d* is put in place of *c* in the 0-node, and *c* is put in a newly created leaf; p is updated twice allowing for incrementing counters of two nodes, left child of the root and the root itself.
5. The letter *c* is the next input letter; thus, first the Huffman codeword for its leaf is given, 001. Next, because the sibling property is violated, the node p (that is, the leaf *c*), is swapped with the leader *f* of $block_1$ that still includes this leaf. Then, p = *parent*(p), and the new parent p of the *c* node is incremented, which leads to another violation of the sibling property and to an exchange of node p with the leader of

$block_2$, namely, with the node *a*. Next, p = *parent*(p), node p is incremented, but because it is the root, the process of updating the tree is finished.

6. The sixth input letter is *c*, which has a leaf in the tree; so first, the Huffman codeword, 11, of the leaf is generated and the counter of node *c* is incremented. The node p, which is the node *c*, violates the sibling property, so p is swapped with the leader, node *a*, of $block_3$. Now, p = *parent*(p), p's counter is incremented, and because p is the root, the tree transformation is concluded for this input letter. The remaining steps can be traced in Figure 11.8.

It is left to the reader to make appropriate modifications to this pseudocode to obtain a `FGKDynamicHuffmanDecoding(`*codeword* `c)` algorithm.

It is possible to design a Huffman coding that does not require any initial knowledge of the set of symbols used by the encoder (Cormack and Horspool 1984). The Huffman tree is initialized to a special escape character. If a new symbol is to be sent, it is preceded by the escape character (or its current codeword in the tree) and followed by the symbol itself. The receiver can now know this symbol so that if its codeword arrives later on, it can be properly decoded. The symbol is inserted in the tree by making the leaf *L* with the lowest frequency a nonleaf so that *L* has two children, one pertaining to the symbol previously in *L* and one to the new symbol.

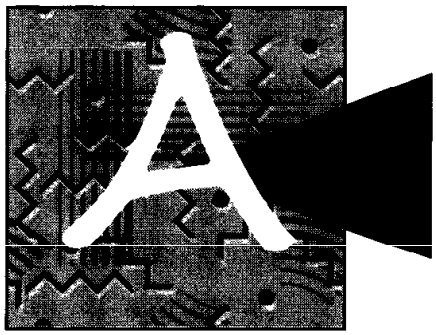
Adaptive Huffman coding surpasses simple Huffman coding in two respects: It requires only one pass through the input, and it adds only an alphabet to the output. Both versions are relatively fast, and more important, they can be applied to any kind of file, not only to text files. In particular, they can compress object or executable files, not only text files. The problem with executable files, however, is that they generally use larger character sets than source code files, and the distribution of these characters is more uniform than in text files. Therefore, the Huffman trees are large, the codewords are of similar length, and the output file is not much smaller than the original; it is compressed merely by 10–20%.

11.3 Shannon-Fano Code

Another efficient method that generates an optimal code as *n* approaches infinity was developed by C. E. Shannon and R. M. Fano. The algorithm is as follows:

```
Order the set of symbols according to the frequency of occurrence;
ShannonFano (sequence S)
    if S has two elements
        attach 0 to the codeword of one element and 1 to the codeword of another;
    else if S has more than one element
        divide S into two subsequences, S1 and S2, with the minimal
            difference between probabilities of each subsequence;
        extend the codeword for each symbol in S1 by attaching 0, and attaching
            1 to each codeword for symbols in S2;
        ShannonFano (S1);
        ShannonFano (S2);
```

Computing Big-O



A.1 Harmonic Series

In some computations in this book, the convention H_n is used for harmonic numbers. The *harmonic numbers* H_n are defined as the sums of the *harmonic series*, a series of the form $\sum_{i=1}^{n} \frac{1}{i}$. This is a very important series for the analysis of searching and sorting algorithms. It is proved that

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \epsilon$$

where $n \geq 1, 0 < \epsilon < \frac{1}{256n^6}$, and *Euler's constant* $\gamma \approx 0.5772$. This approximation, however, is very unwieldy and, in the context of our analyses, not necessary in this form. H_n's largest term is almost always $\ln n$, the only increasing term in H_n. Thus, H_n can be referred to as $O(\ln n)$.

A.2 Approximation of the Function lg(N!)

The roughest approximation of $\lg(n!)$ can be obtained by observing that each number in the product $n! = 1 \cdot 2 \cdot \cdots \cdot (n-1) \cdot n$ is less than or equal to n. Thus, $n! \leq n^n$ (only for $n = 1$, $n = n^n$), which implies that $\lg(n!) < \lg(n^n) = n \lg n$—that is, $n \lg n$ is an upper bound of $\lg(n!)$—and that $\lg(n!)$ is $O(n \lg n)$.

Let us also find a lower bound for $\lg(n!)$. If the elements of the product $n!$ are grouped appropriately, as in

$$P_{n!} = (1 \cdot n)(2 \cdot (n-1))(3 \cdot (n-2)) \cdots (i \cdot (n-i+1)) \ldots, \text{for } 1 \leq i \leq \frac{n}{2}$$

then it can be noted that there are $\frac{n}{2}$ such terms and $n! = P_{n!}$ for even ns or $\frac{n+1}{2}$ terms and $n! = P_{n!}\frac{n+1}{2}$ for odd ns. We claim that each term of $P_{n!}$ is not less than n, or

$$1 \le i \le \frac{n}{2} \Rightarrow i(n-i+1) \ge n$$

In fact, this holds because

$$\frac{n}{2} \ge i = \frac{i(i-1)}{i-1} \Rightarrow i(n-2i+2) \ge n$$

and, as can easily be checked,

$$i \ge 1 \Rightarrow (n-2i+2) \le (n-i+1)$$

We have shown that $n! = P_{n!} \ge n^{\frac{n}{2}}$, which means that $\lg(n!) \ge \frac{n}{2} \lg n$. This assumes that n is even. If n is odd, it has to be raised to the power of $\frac{n+1}{2}$, which introduces no substantial change.

The number $\lg(n!)$ has been estimated using the lower and upper bounds of this function, and the result is $\frac{n}{2} \lg n \le \lg(n!) \le n \lg n$. To approximate $\lg(n!)$, lower and upper bounds have been used that both grow at the rate of $n \lg n$. This implies that $\lg(n!)$ grows at the same rate as $n \lg n$ or that $\lg(n!)$ is not only $O(n \lg n)$ but also $\Theta(n \lg n)$. In other words, any sorting algorithm using comparisons on an array of size n must make at least $O(n \lg n)$ comparisons in the worst case. Thus, the function $n \lg n$ approximates the optimal number of comparisons in the worst case.

However, this result seems unsatisfactory because it refers only to the worst case, and such a case occurs only occasionally. Most of the time, average cases with random orderings of data occur. Is the number of comparisons really better in such cases and is it a reasonable assumption that the number of comparisons in the average case can be better than $O(n \lg n)$? Unfortunately, this conjecture has to be rejected, and the following computations prove it false.

Our conjecture is that, in any binary tree with m leaves and two children for each nonterminal node, the average number of arcs leading from the root to a leaf is greater than or equal to $\lg m$.

For $m = 2$, $\lg m = 1$, if there is just a root with two leaves, then there is only one arc to every one of them. Assume that the proposition holds for a certain $m \ge 2$ and that

$$\text{Ave}_m = \frac{p_1 + \cdots + p_m}{m} \ge \lg m$$

where each p_i is a path (the number of arcs) from the root to node i. Now consider a randomly chosen leaf with two children about to be attached. This leaf converted to a nonterminal node has an index m (this index is chosen to simplify the notation) and a path from the root to the node m is p_m. After adding two new leaves, the total number of leaves is incremented by one and the path for both these appended leaves is $p_{m+1} = p_m + 1$. Is it now true that

$$\text{Ave}_{m+1} = \frac{p_1 + \cdots + p_{m-1} + 2p_m + 2}{m+1} \ge \lg(m+1)$$

From the definition of Ave_m and Ave_{m+1} and from the fact that $p_m = \text{Ave}_m$ (since leaf m was chosen randomly),

$$(m+1)\text{Ave}_{m+1} = m\text{Ave}_m + p_m + 2 = (m+1)\text{Ave}_m + 2$$

Is it now true that

$$(m+1)\text{Ave}_{m+1} \geq (m+1)\lg(m+1)$$

or

$$(m+1)\text{Ave}_{m+1} = (m+1)\text{Ave}_m + 2 \geq (m+1)\lg m + 2 \geq (m+1)\lg(m+1)$$

This is transformed into

$$2 \geq \lg\left(\frac{m+1}{m}\right)^{m+1} = \lg\left(1+\frac{1}{m}\right) + \lg\left(1+\frac{1}{m}\right)^{m} \rightarrow \lg 1 + \lg e = \lg e \approx 1.44$$

which is true for any $m \geq 1$. This completes the proof of the conjecture.

This proves that for a randomly chosen leaf of an m-leaf decision tree, the reasonable expectation is that the path from the root to the leaf is not less than lg m. The number of leaves in such a tree is not less than $n!$, which is the number of all possible orderings of an n-element array. If $m \geq n!$, then $\lg m \geq \lg(n!)$. That is the unfortunate result indicating that an average case also requires, like the worst case, $\lg(n!)$ comparisons (length of path = number of comparisons), and as already estimated, $\lg(n!)$ is $O(n \lg n)$. This is also the best that can be expected in average cases.

A.3 Big-O for Average Case of Quicksort

Let $C(n)$ be the number of comparisons required to sort an array of n cells. Because the arrays of size 1 and 0 are not partitioned, $C(0) = C(1) = 0$. Assuming a random ordering of an n-element array, any element can be chosen as the bound; the probability that any element will become the bound is the same for all elements. With $C(i-1)$ and $C(n-i)$ denoting the numbers of the comparisons required to sort the two subarrays, there are

$$C(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{n}(C(i-1) + C(n-i)), \text{ for } n \geq 2$$

comparisons, where $n-1$ is the number of comparisons in the partition of the array of size n. First, some simplification can be done:

$$\begin{aligned} C(n) &= n - 1 + \frac{1}{n}\left(\sum_{i=1}^{n} C(i-1) + \sum_{i=1}^{n} C(n-i)\right) \\ &= n - 1 + \frac{1}{n}\left(\sum_{i=1}^{n} C(i-1) + \sum_{j=1}^{n} C(j-1)\right) \\ &= n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} C(i) \end{aligned}$$

or

$$nC(n) = n(n-1) = 2\sum_{i=0}^{n-1} C(i)$$

To solve the equation, the summation operator is removed first. To that end, the last equation is subtracted from an equation obtained from it,

$$(n+1)C(n+1) = (n+1)n + 2\sum_{i=0}^{n} C(i)$$

resulting in

$$(n+1)C(n+1) - nC(n) = (n+1)n - n(n-1) + 2\left(\sum_{i=0}^{n} C(i) - \sum_{i=0}^{n-1} C(i)\right) = 2C(n) + 2n$$

from which

$$\frac{C(n+1)}{n+2} = \frac{C(n)}{n+1} + \frac{2n}{(n+1)(n+2)} = \frac{C(n)}{n+1} + \frac{4}{n+2} - \frac{2}{n+1}$$

This equation can be expanded, which gives

$$\frac{C(2)}{3} = \frac{C(1)}{2} + \frac{4}{3} - \frac{2}{2} = \frac{4}{3} - \frac{2}{2}$$

$$\frac{C(3)}{4} = \frac{C(2)}{3} + \frac{4}{4} - \frac{2}{3}$$

$$\frac{C(4)}{5} = \frac{C(3)}{4} + \frac{4}{5} - \frac{2}{4}$$

$$\vdots$$

$$\frac{C(n)}{n+1} = \frac{C(n-1)}{n} + \frac{4}{n+1} - \frac{2}{n}$$

$$\frac{C(n+1)}{n+2} = \frac{C(n)}{n+1} + \frac{4}{n+2} - \frac{2}{n+1}$$

from which

$$\begin{aligned}\frac{C(n+1)}{n+2} &= \left(\frac{4}{3} - \frac{2}{2}\right) + \left(\frac{4}{4} - \frac{2}{3}\right) + \left(\frac{4}{5} - \frac{2}{4}\right) + \cdots + \left(\frac{4}{n+1} - \frac{2}{n}\right) \\ &\quad + \left(\frac{4}{n+2} - \frac{2}{n+1}\right) \\ &= -\frac{2}{2} + \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \cdots + \frac{2}{n} + \frac{2}{n+1} + \frac{4}{n+2} \\ &= -4 + 2H_{n+2} + \frac{2}{n+2}\end{aligned}$$

Note that H_{n+2} is a harmonic number. Using an approximation for this number (cf. Appendix A.1)

$$\begin{aligned}C(n) &= (n+1)\left(-4 + 2H_{n+1} + \frac{2}{n+1}\right) \\ &= (n+1)\left(-4 + 2O(\ln n) + \frac{2}{n+1}\right) \\ &= O(n \lg n)\end{aligned}$$

A.4 Average Path Length in a Random Binary Tree

In Chapter 6, an approximation is used for the average path length in a randomly created binary search tree. Assuming that

$$P_n(i) = \frac{(i-1)(P_{i-1}+1) + (n-i)(P_{n-i}+1)}{n}$$

this approximation is given by this recurrence relation

$$P_1 = 0$$

$$P_n = \frac{1}{n}\sum_{i=1}^{n} P_n(i) = \frac{1}{n^2}\sum_{i=1}^{n} ((i-1)(P_{i-1}+1) + (n-i)(P_{n-i}+1))$$

$$P_n = \frac{2}{n^2}\sum_{i=1}^{n-1} i(P_i+1) \tag{1}$$

From this, we also have

$$P_{n-1} = \frac{2}{(n-1)^2}\sum_{i=1}^{n-2} i(P_i+1) \tag{2}$$

After multiplying this equation by $\frac{(n-1)^2}{n^2}$ and subtracting the resulting equation from (1), we have

$$P_n = P_{n-1}\frac{(n-1)^2}{n^2} + \frac{2(n-1)(P_{n-1}+1)}{n^2} = \frac{(n-1)}{n^2}((n+1)P_{n-1}+2)$$

After successive applications of this formula to each P_{n-i}, we have

$$P_n = \frac{n-1}{n^2}\left((n+1)\frac{(n-2)}{(n-1)^2}\left(n\frac{(n-3)}{(n-2)^2}\left((n-1)\frac{(n-4)}{(n-3)^2}\left(\cdots\frac{1}{2^2}(3P_1+2)\cdots\right)+2\right)+2\right)+2\right)$$

$$P_n = 2\left(\frac{n-1}{n^2} + \frac{(n+1)(n-2)}{(n-1)n^2} + \frac{(n+1)(n-3)}{n(n-1)(n-2)} + \frac{(n+1)(n-4)}{n(n-2)(n-3)} + \cdots + \frac{n+1}{2\cdot 3n}\right)$$

$$P_n = 2\left(\frac{n+1}{n}\right)\sum_{i=1}^{n-1}\frac{n-i}{(n-i+1)(n-i+2)} = 2\left(\frac{n+1}{n}\right)\sum_{i=1}^{n-1}\left(\frac{2}{n-i+2} - \frac{1}{n-i+1}\right)$$

$$P_n = 2\left(\frac{n+1}{n}\right)\frac{2}{n+1} + 2\left(\frac{n+1}{n}\right)\left(\sum_{i=1}^{n}\frac{1}{i} - 2\right) = 2\left(\frac{n+1}{n}\right)H_n - 4$$

So, P_n is $O(2 \ln n)$.