DATA STRUCTURES and ALGORITHMS in C++

FOURTH EDITION

Adam Drozdek

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, a character, or a compound item such as a structure or union. However, this content can also be the location of another variable; variables with such contents are called *pointers.* Pointers are usually auxiliary variables that 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 that 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; 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`.

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

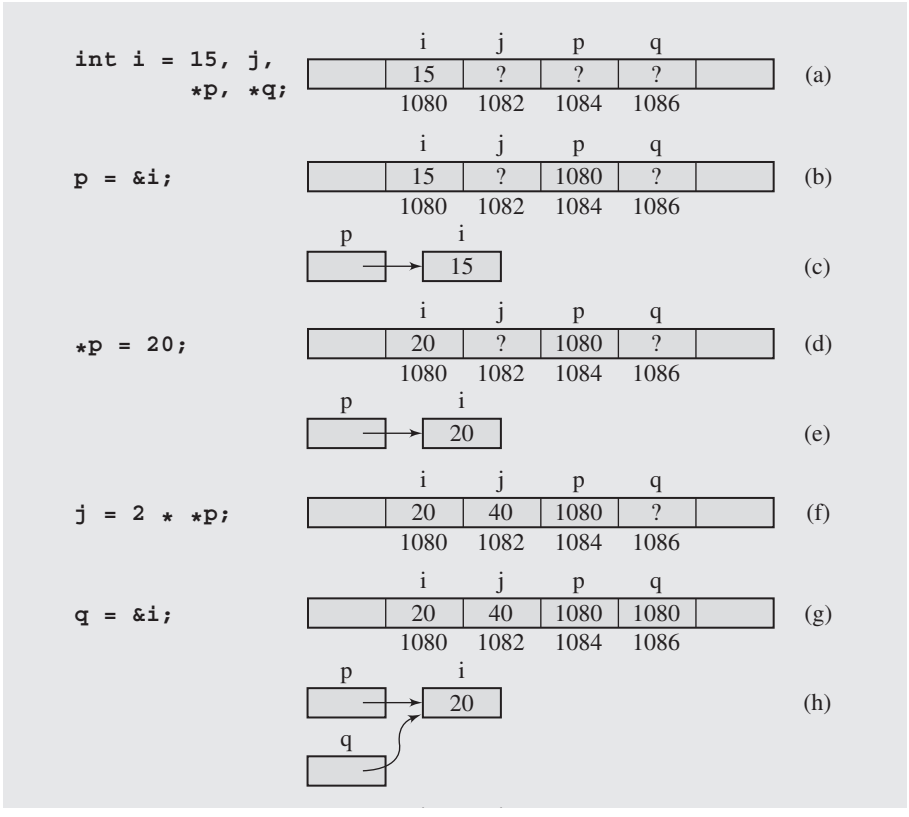
```
*p = 20;
```

The star (`*`) here is an indirection operator that 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.

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 aside by the memory manager, which is performed dynamically during the run of the program, unlike for variables, whose locations are allocated at compilation time.

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).



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

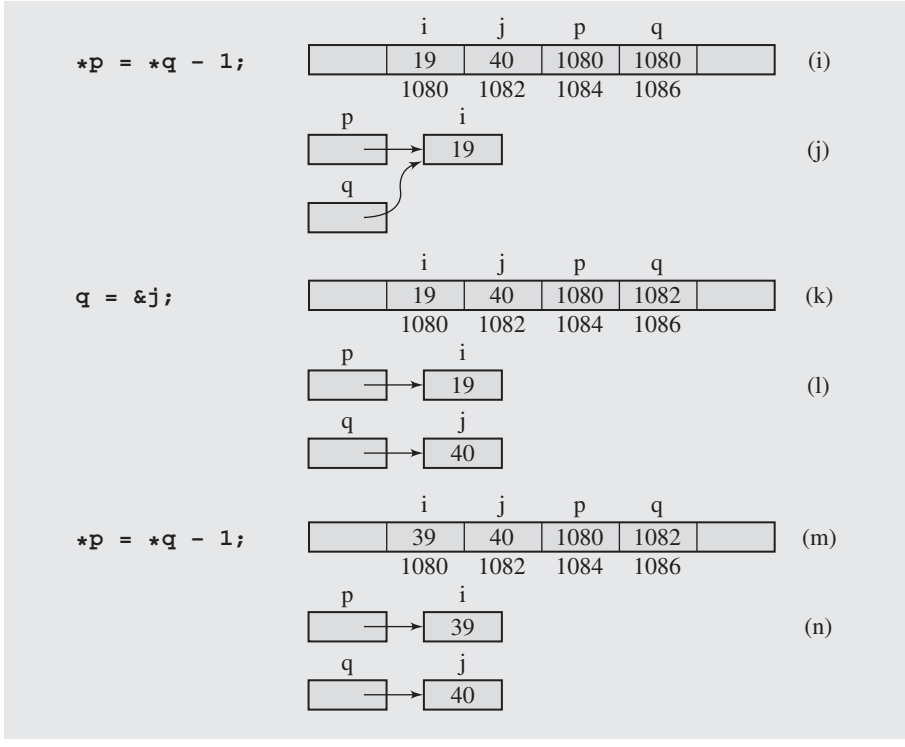
```
p = new int;
```

the program requests enough space to store one integer, from the memory manager, and the address of the beginning of this portion of memory is stored in `p`. Now the values can be assigned to the memory block pointed to by `p` only indirectly through a pointer, either pointer `p` or any other pointer `q` that 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;
```

FIGURE 1.1 *(continued)*



However, after executing this statement, the beginning addresses of the released memory block are 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 to null or points to null but that `p` becomes null or `p` is null.

Another problem associated with delete is the *memory leak.* Consider the following two lines of code:

```
p = new int;
p = new int;
```

After allocating one cell for an integer, the same pointer p is used to allocate another cell. After the second assignment, the first cell becomes inaccessible and also unavailable for subsequent memory allocations for the duration of the program. The problem is with not releasing with delete the memory accessible from p before the second assignment is made. The code should be:

```
p = new int;
delete p;
p = new int;
```

Memory leaks can become a serious problem when a program uses more and more memory without releasing it, eventually exhausting memory and leading to abnormal termination. This is especially important in programs that are executed for a long time, such as programs in servers.

1.4.1 Pointers and Arrays

In the previous section, 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 where we would need 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 use of pointers. Consider Figure 1.1b. In this figure, pointer p points to location 1080. But it also allows accessing of locations 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 a 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 that 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 1 byte, and c+1 equals 1059 because one long number occupies 4 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. Also, `delete` should be used with pointers that were assigned a value with `new`. For this reason, the two following applications of `delete` are very likely to lead to a program crash:

```
int a[10], *p = a;
delete [] p;
int n = 10, *q = &n;
delete q;
```

A very important type of array is a string, or an array of characters. Many predefined functions 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 end 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 = strdup(n);
      age = a;
    }
};
```

The intention of the declarations

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

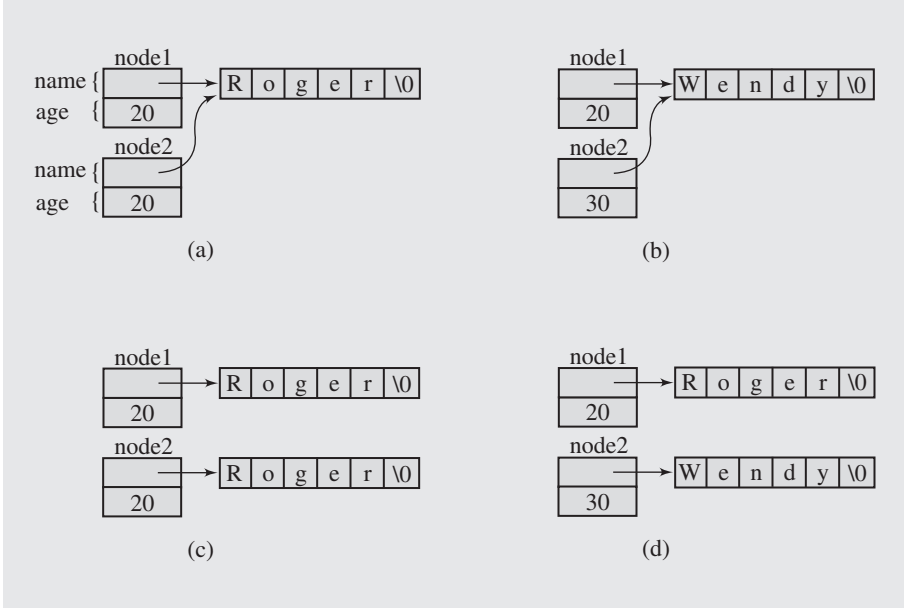
is to create object `node1`, assign values to the two data members in `node1`, and then create object `node2` and initialize its data members to the same values as in `node1`. These objects are to be independent entities so that assigning values to one of them should not affect values in the other. However,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

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

the printing statement

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

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



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;
```

are executed, `node2.age` is properly updated, but the string `"Roger"` pointed to by the `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 = strdup(n);
         age = a;
    }
    Node(const Node& n) { // copy constructor;
         name = strdup(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 `node1` 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)
            free(name);
        name = strdup(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 the `name` of some other object or to a string variable) and memory occupied by this string can no longer be released, which leads to a memory leak. 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 the class `Node`, a destructor can be defined as

```
~Node() {
     if (name != 0)
         free(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 the 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,

```
f2(a,3) = 6;
```

which assigns 6 to `a[3]` so that `a` = [1 2 3 6 5]. Note that we can accomplish the same with pointers, but dereferencing has to be used explicitly:

```
int* f3(int a[], int i) {
    return &a[i];
  }
```

and then

```
*f3(a,3) = 6;
```

Reference variables and the reference return type have to be used with caution because there is a possibility of compromising the information-hiding principle when they are used improperly. Consider class `C`:

```
class C {
public:
     int& getRefN() {
          return n;
     }
     int getN() {
          return n;
     }
private:
     int n;
} c;
```

and these assignments:

```
int& k = c.getRefN();
k = 7;
cout << c.getN();
```

Although `n` is declared `private`, after the first assignment it can be accessed at will from the outside through `k` and assigned any value. An assignment can also be made through `getRefN()`:

```
c.getRefN() = 9;
```

1.4.5 Pointers to Functions

As indicated in Section 1.4.1, one of the attributes of a variable is its address indicating its position in computer memory. The same is true for functions: One of the attributes of a function is the address indicating the location of the body of the function in memory. Upon a function call, the system transfers control to this location to execute the function. For this reason, it is possible to use pointers to functions. These pointers are very useful in implementing functionals (that is, functions that take functions as arguments) such as an integral.

Consider a simple function:

```
double f(double x) {
    return 2*x;
}
```

With this definition, `f` is the pointer to the function `f()`, `*f` is the function itself, and `(*f)(7)` is a call to the function.

Consider now writing a C++ function that computes the following sum:

$$\sum_{i=n}^{m} f(i)$$

To compute the sum, we have to supply not only limits n and m, but also a function f. Therefore, the desired implementation should allow for passing numbers not only as arguments, but also as functions. This is done in C++ in the following fashion:

```
double sum(double (*f)(double), int n, int m) {
    double result = 0;
    for (int i = n; i <= m; i++)
        result += f(i);
    return result;
}
```

In this definition of `sum()`, the declaration of the first formal argument

```
double (*f)(double)
```

means that `f` is a pointer to a function with one double argument and a double return value. Note the need for parentheses around `*f`. Because parentheses have precedence over the dereference operator `*`, the expression

```
double *f(double)
```

declares a function that returns a pointer to a double value.

The function `sum()` can be called now with any built-in or user-defined double function that takes one double argument, as in

```
cout << sum(f,1,5) << endl;
cout << sum(sin,3,7) << endl;
```

Another example is a function that finds a root of a continuous function in an interval. The root is found by repetitively bisecting an interval and finding a midpoint

of the current interval. If the function value at the midpoint is zero or the interval is smaller than some small value, the midpoint is returned. If the values of the function on the left limit of the current interval and on the midpoint have opposite signs, the search continues in the left half of the current interval; otherwise, the current interval becomes its right half. Here is an implementation of this algorithm:

```
double root(double (*f)(double), double a, double b, double epsilon) {
    double middle = (a + b) / 2;
    while (f(middle) != 0 && fabs(b - a) > epsilon) {
        if (f(a) * f(middle) < 0)   // if f(a) and f(middle) have
              b = middle;           // opposite signs;
        else a = middle;
        middle = (a + b) / 2;
    }
    return middle;
}
```

Complexity Analysis

2

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 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 usually 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 100 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 Ada 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 that 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:

$$f(n) = n^2 + 100n + \log_{10} n + 1{,}000 \quad \textbf{(2.1)}$$

For small values of *n,* the last term, 1,000, is the largest. When *n* equals 10, the second (100*n*) and last (1,000) 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 for large *n*.

FIGURE 2.1 The growth rate of all terms of function $f(n) = n^2 + 100n + \log_{10} n + 1{,}000$.

n	f(n)	n^2		100n		$\log_{10} n$		1,000	
	Value	Value	%	Value	%	Value	%	Value	%
1	1,101	1	0.1	100	9.1	0	0.0	1,000	90.83
10	2,101	100	4.76	1,000	47.6	1	0.05	1,000	47.60
100	21,002	10,000	47.6	10,000	47.6	2	0.001	1,000	4.76
1,000	1,101,003	1,000,000	90.8	100,000	9.1	3	0.0003	1,000	0.09
10,000	101,001,004	100,000,000	99.0	1,000,000	0.99	4	0.0	1,000	0.001
100,000	10,010,001,005	10,000,000,000	99.9	10,000,000	0.099	5	0.0	1,000	0.00

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 $n \geq N$.

*Bachmann introduced the notation quite casually in his discussion of an approximation of a function and defined it rather succinctly: "with the symbol $O(n)$ we express a magnitude whose order in respect to *n* does not exceed the order of *n*" (Bachmann 1894, p. 401).

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 of 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 2.2 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

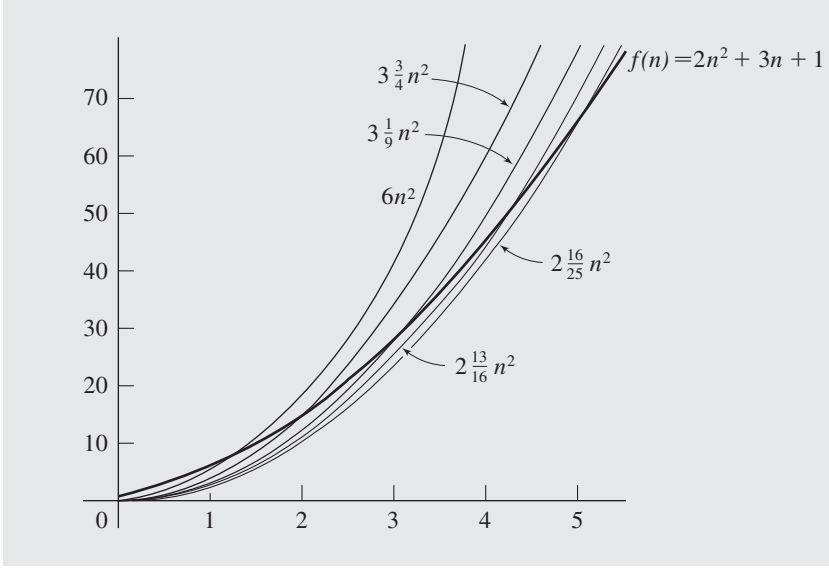
$$2 + \frac{3}{n} + \frac{1}{n^2} \leq c$$

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 $g(n)$. Because 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 $n > 1.5$. 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 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.

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



The inherent imprecision of the big-O notation goes even further, because 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 $f(n)$ is $O(h(n))$. (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 $O(h(n))$.

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

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

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 "≥" with "≤." 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 (i.e., 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, because the functions $2n^2$, $3n^2$, $4n^2$, . . . 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 that 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 $O(g(n))$ 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 10^8n and $10n^2$. The first function is $O(n)$ and the second is $O(n^2)$. Using just the big-O information, the second algorithm is rejected because the number of steps grows too fast. It is true but, again, in the long run, because 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 many 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 $O(g(n))$ and the constant c is too large to have practical significance. Thus, 10^8n 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 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
logarithmic	$O(\lg n)$	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.

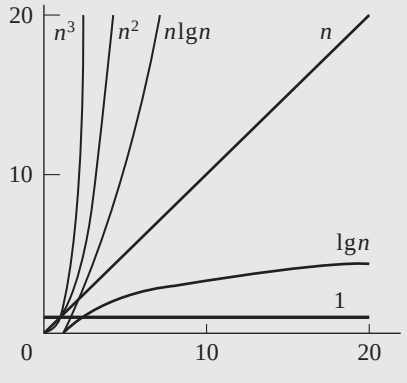


Figure 2.4 indicates that some ill-designed algorithms, or algorithms whose complexity cannot be improved, have no practical application on available computers. To process 1 million items with a quadratic algorithm, over 11 days are needed, and for a cubic algorithm, thousands of years. Even if a computer 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 1,000-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 of which 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.

Algorithms with nested loops usually have a larger complexity than algorithms 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;
}
```

The outer loop is executed $n - 4$ times. For each i, the inner loop is executed only four times; for each iteration of the outer loop there are eight assignments in the inner loop, and this number does not depend on the size of the array. With one initialization of `i`, $n - 4$ autoincrements of `i`, and $n - 4$ initializations of `j` and `sum`, the program makes $1 + 8 \cdot (n - 4) + 3 \cdot (n - 4) = O(n)$ assignments.

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 $n - 1 - i$ times for each $i \in \{0, \ldots, n - 2\}$. 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 T>  // overloaded operator < is used;
int binarySearch(const T arr[], int arrSize, const T& 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,

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

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, $\Sigma_i p(input_i) = 1$.

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 having two tries equals $\frac{1}{n}$, . . . , and the probability of having 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 worst cases.

The complexity for the three cases was relatively easy to determine for a 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 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 after four tries in the middle of the eighth eighth of the array; or after try lg n in the first cell, or after try lg n in the third cell; 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,

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

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

$$\ldots$$

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

$$\ldots$$

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

$$2^{\lg n-1-2j} \geq \frac{\frac{\lg n}{2}-j-1}{\frac{\lg n}{2}-j} = 1 - \frac{1}{\frac{\lg n}{2}-j} \qquad \textbf{(2.5)}$$

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.

2.11 EXERCISES

1. Explain the meaning of the following expressions:

a. $f(n)$ is $O(1)$.

b. $f(n)$ is $\Theta(1)$.

c. $f(n)$ is $n^{O(1)}$.

2. Assuming that $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, prove the following statements:

a. $f_1(n) + f_2(n)$ is $O(\max(g_1(n),g_2(n)))$.

b. If a number k can be determined such that for all $n > k$, $g_1(n) \leq g_2(n)$, then $O(g_1(n)) + O(g_2(n))$ is $O(g_2(n))$.

c. $f_1(n) * f_2(n)$ is $O(g_1(n) * g_2(n))$ (rule of product).

d. $O(cg(n))$ is $O(g(n))$.

e. c is $O(1)$.

3. Prove the following statements:

a. $\Sigma^n_{i=1}\, i^2$ is $O(n^3)$ and more generally, $\Sigma^n_{i=1}\, i^k$ is $O(n^{k+1})$.

b. $an^k/\lg n$ is $O(n^k)$, but $an^k/\lg n$ is not $\Theta(n^k)$.

c. $n^{1.1} + n \lg n$ is $\Theta(n^{1.1})$.

d. 2^n is $O(n!)$ and $n!$ is not $O(2^n)$.

e. 2^{n+a} is $O(2^n)$.

f. 2^{2n+a} is not $O(2^n)$.

g. $2^{\sqrt{\lg n}}$ is $O(n^a)$.

4. Make the same assumptions as in Exercise 2 and, by finding counterexamples, refute the following statements:

a. $f_1(n) - f_2(n)$ is $O(g_1(n) - g_2(n))$.

b. $f_1(n)/f_2(n)$ is $O(g_1(n)/g_2(n))$.

5. Find functions f_1 and f_2 such that both $f_1(n)$ and $f_2(n)$ are $O(g(n))$, but $f_1(n)$ is not $O(f_2)$.

6. Is it true that

a. if $f(n)$ is $\Theta(g(n))$, then $2^{f(n)}$ is $\Theta(2^{g(n)})$?

b. $f(n) + g(n)$ is $\Theta(\min(f(n),g(n)))$?

c. 2^{na} is $O(2^n)$?

7. The algorithm presented in this chapter for finding the length of the longest subarray with the numbers in increasing order is inefficient, because there is no need to continue to search for another array if the length already found is greater

than the length of the subarray to be analyzed. Thus, if the entire array is already in order, we can discontinue the search right away, converting the worst case into the best. The change needed is in the outer loop, which now has one more test:

```
for (i = 0, length = 1; i < n-1 && length < n==i; i++)
```

What is the worst case now? Is the efficiency of the worst case still $O(n^2)$?

8. Find the complexity of the function used to find the *k*th smallest integer in an unordered array of integers:

```
int selectkth(int a[], int k, int n) {
    int i, j, mini, tmp;
    for (i = 0; i < k; i++) {
        mini = i;
        for (j = i+1; j < n; j++)
            if (a[j]<a[mini])
                mini = j;
        tmp = a[i];
        a[i] = a[mini];
        a[mini] = tmp;
    }
    return a[k-1];
}
```

9. Determine the complexity of the following implementations of the algorithms for adding, multiplying, and transposing $n \times n$ matrices:

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        a[i][j] = b[i][j] + c[i][j];

for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        for (k = a[i][j] = 0; k < n; k++)
            a[i][j] += b[i][k] * c[k][j];

for (i = 0; i < n - 1; i++)
    for (j = i+1; j < n; j++) {
        tmp = a[i][j];
        a[i][j] = a[j][i];
        a[j][i] = tmp;
    }
```

10. Find the computational complexity for the following four loops:

a.
```
for (cnt1 = 0, i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
          cnt1++;
```

b.
```
for (cnt2 = 0, i = 1; i <= n; i++)
      for (j = 1; j <= i; j++)
          cnt2++;
```

c.
```
for (cnt3 = 0, i = 1; i <= n; i *= 2)
      for (j = 1; j <= n; j++)
          cnt3++;
```

d.
```
for (cnt4 = 0, i = 1; i <= n; i *= 2)
      for (j = 1; j <= i; j++)
          cnt4++;
```

11. Find the average case complexity of sequential search in an array if the probability of accessing the last cell equals $\frac{1}{2}$, the probability of the next to last cell equals $\frac{1}{4}$, and the probability of locating a number in any of the remaining cells is the same and equal to $\frac{1}{4(n-2)}$.

Linked Lists

3

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.

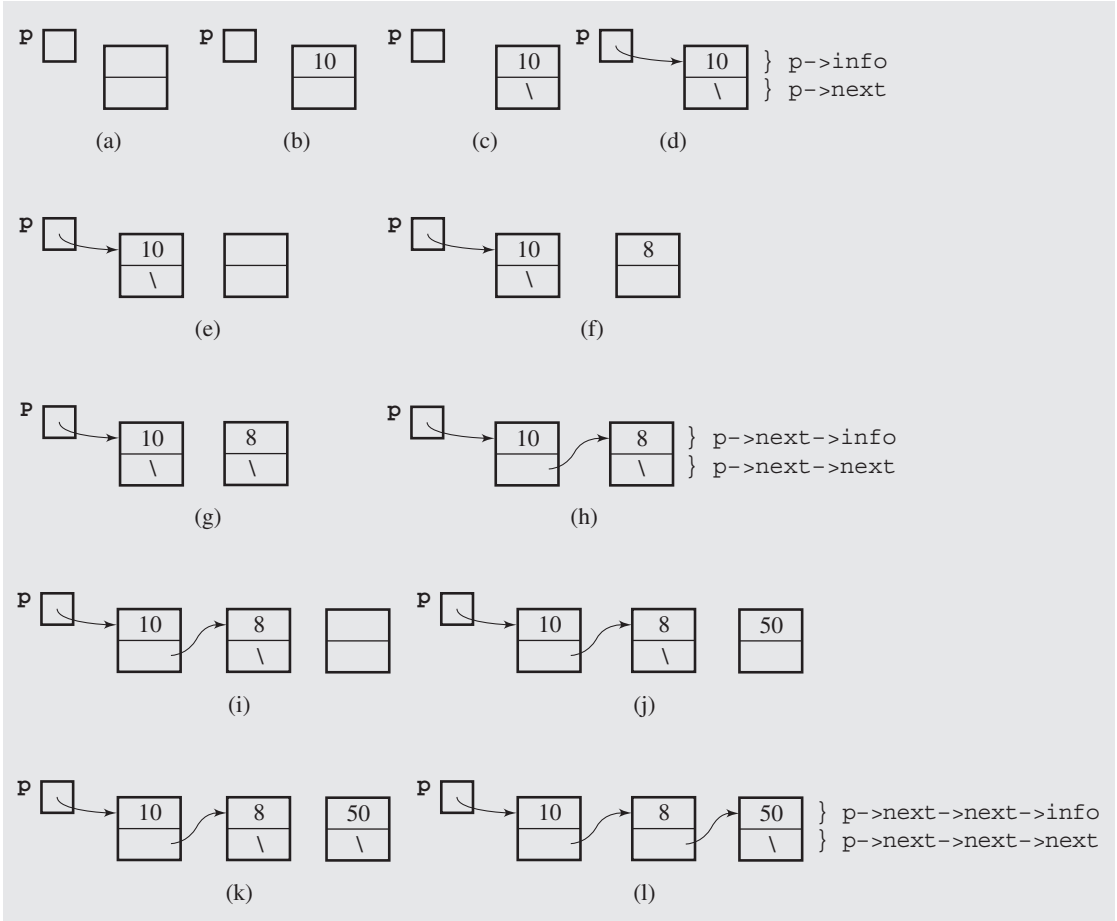
3.1 Singly Linked Lists

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 IntSLLNode {
public:
    IntSLLNode() {
        next = 0;
    }
    IntSLLNode(int i, IntSLLNode *in = 0) {
        info = i; next = in;
    }
```

FIGURE 3.1 A singly linked list.




```
    int info;
    IntSLLNode *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 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 `IntSLLNode` 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. Objects that include such a data member are called self-referential objects.

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 3.1l. One way to create this three-node linked list is to first generate the node for number 10, then the node for 8, and finally the node for 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

```
IntSLLNode *p = new IntSLLNode(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 `IntSLLNode` 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. The second and third steps—initialization of data members of the new `IntSLLNode`—are performed by invoking the constructor `IntSLLNode(10)`, which turns into the constructor `IntSLLNode(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 IntSLLNode(8);
```

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

1. A new node is created (Figure 3.1e).
2. The constructor assigns the number 8 to the `info` member of this node (Figure 3.1f).
3. The constructor assigns null to its `next` member (Figure 3.1g).
4. The new node is included 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 IntSLLNode(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 1,003rd node on the list? Typing 1,003 `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.

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

#ifndef INT_LINKED_LIST
#define INT_LINKED_LIST

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

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

FIGURE 3.2 *(continued)*

```
private:
    IntSLLNode *head, *tail;
};

#endif

//************************  intSLList.cpp ************************

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

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

Continues

FIGURE 3.2 *(continued)*

```
        delete head;
        head = tail = 0;
    }
    else {              // if more than one node in the list,
        IntSLLNode *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 nonempty 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
            IntSLLNode *tmp = head;
            head = head->next;
            delete tmp;            // and old head is deleted;
        }
        else {                      // if more than one node in the list
            IntSLLNode *pred, *tmp;
            for (pred = head, tmp = head->next; // and a nonhead 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;
            }
        }
}
bool IntSLList::isInList(int el) const {
    IntSLLNode *tmp;
    for (tmp = head; tmp != 0 && !(tmp->info == el); tmp = tmp->next);
    return tmp != 0;
}
```

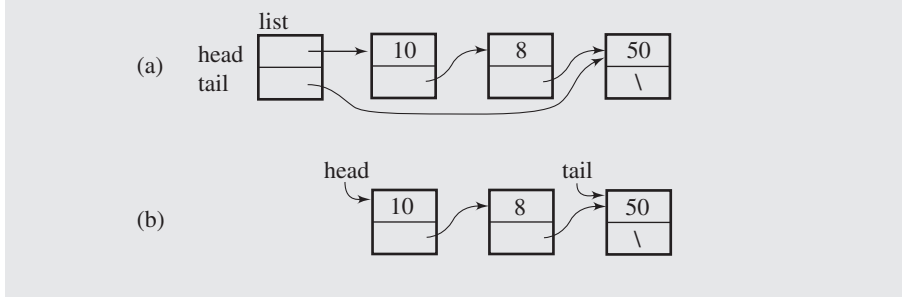
The singly linked list implementation in Figure 3.2 uses two classes: one class, `IntSLLNode`, for nodes of the list, and another, `IntSLList`, for access to the list. The class `IntSLList` defines two data members, `head` and `tail`, which are pointers to the first and the last nodes of a list. This explains why all members of `IntSLLNode` 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 `IntSLLNode` 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.

FIGURE 3.3 A singly linked list of integers.



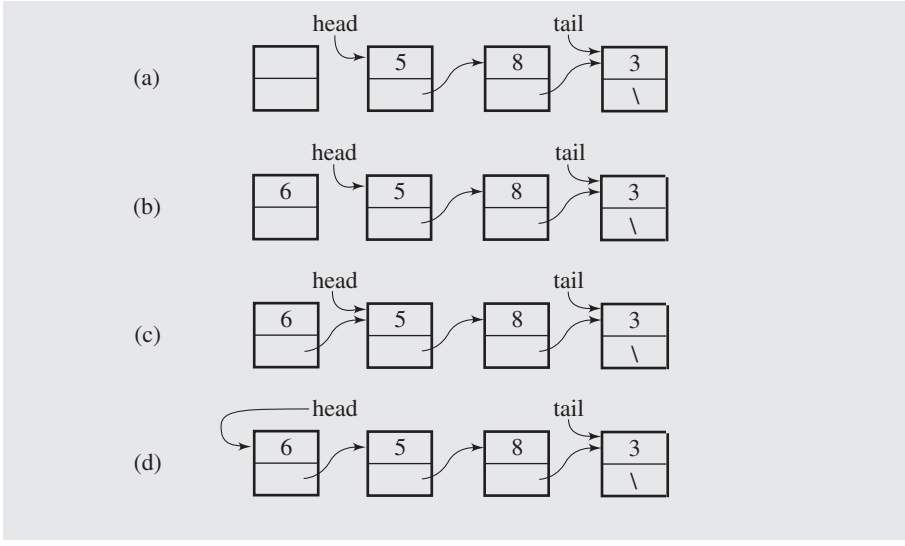
Besides the `head` and `tail` members, the class `IntSLList` also defines member functions that allow us to manipulate the lists. We now look more closely at some basic operations on linked lists presented in Figure 3.2.

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 3.4 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 `IntSLLNode(el,head)`. The last step is executed directly in the function by assigning the address of the newly created node to `head`.

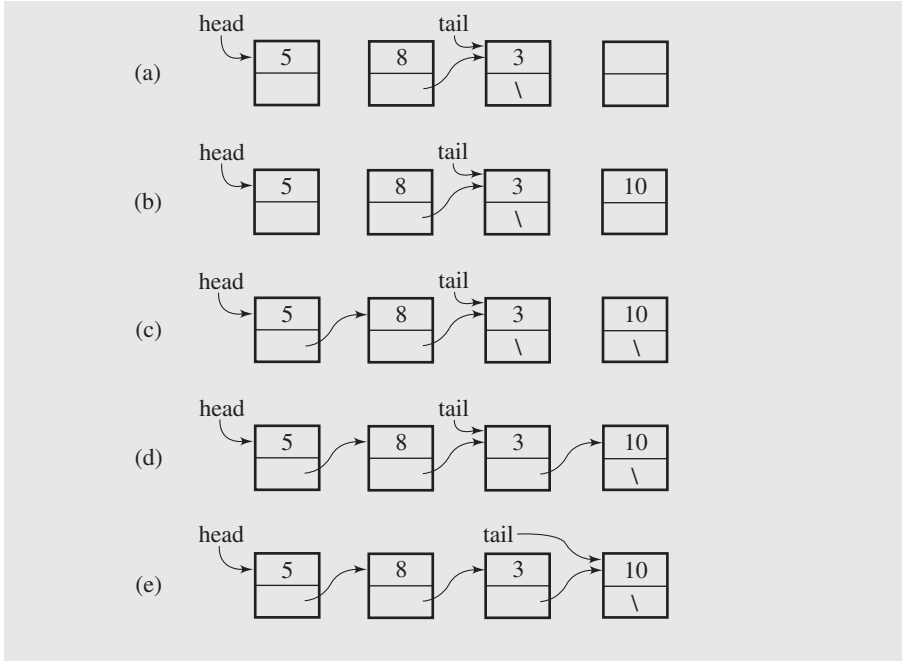
The member function `addToHead()` singles out one special case, namely, inserting a new node in an empty linked list. In an empty linked list, both `head` and `tail` are null; therefore, both become pointers to the only node of the new list. When inserting in a nonempty list, only `head` needs to be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

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

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.

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



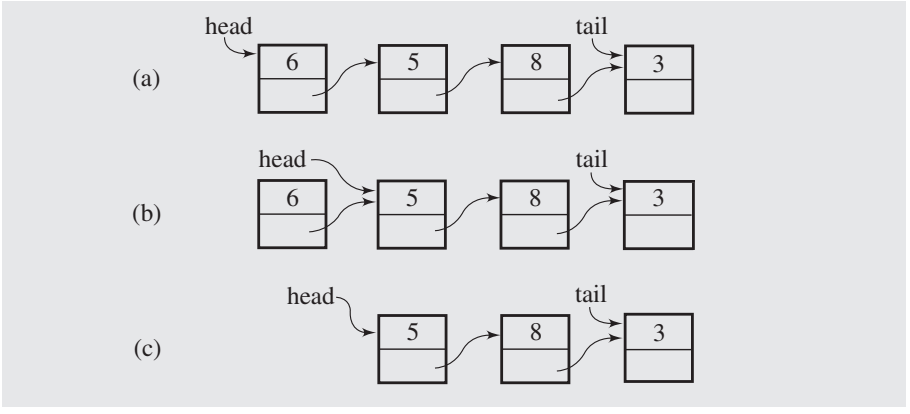
The process of inserting a new node at the beginning of the list is very similar to the process of inserting a node at the end of the list. This is because the implementation of `IntSLList` uses two pointer members: `head` and `tail`. For this reason, both `addToHead()` and `addToTail()` can be executed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

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



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 a 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 the caller may wish to continue 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` statement, with the solution appropriate to the particular case. If the statement 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 delete;
```

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

```
int IntSLList::deleteFromHead() {
    if (!isEmpty()) {          // if nonempty 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 a 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 nonempty 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. In either case, this means that the if statement in deleteFromHead() is redundant.

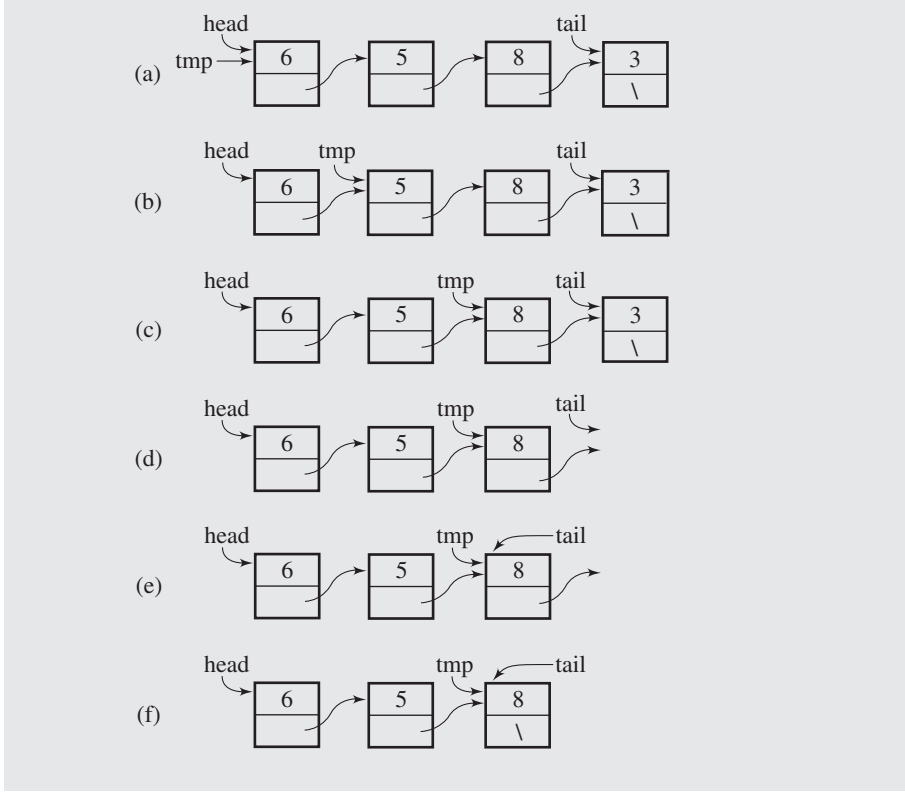
The second special case is when the list has only one node to be removed. In this case, the list becomes empty, which requires setting tail and head to null.

The second deletion operation consists of 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 that scans 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 to 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);
```

FIGURE 3.7 Deleting a node from the end of a singly linked list.



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.

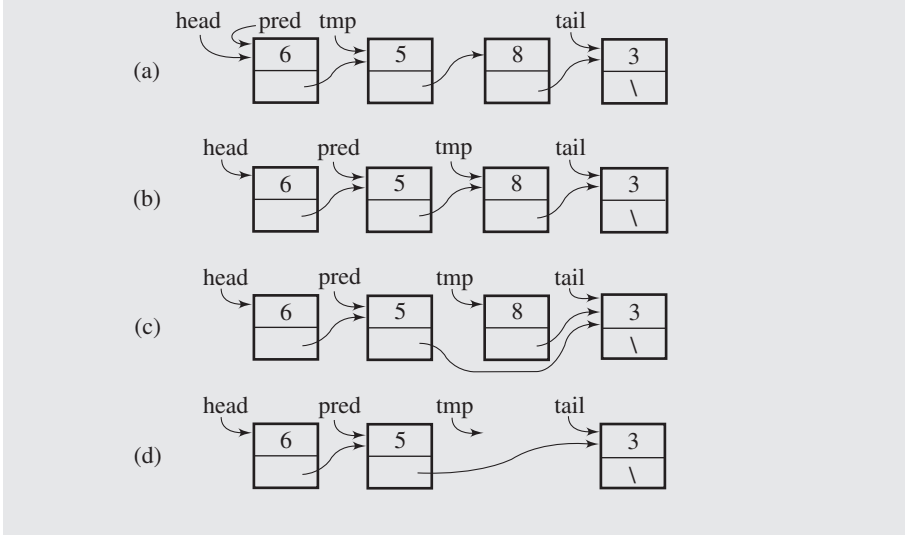
The most time-consuming part of `deleteFromTail()` is finding the next to last node performed by the `for` loop. It is clear that the loop performs $n - 1$ iterations in a list of n nodes, which is the main reason this member function takes $O(n)$ time to delete the last node.

The two discussed deletion operations remove a node from the head or from the tail (that is, always from the same position) and return the 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.

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 by first 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 the 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).

FIGURE 3.8 Deleting a node from a singly linked list.



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

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 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 != 0`: if `tmp` is not null, `el` was found and `true` is returned. If `tmp` is null, the search was unsuccessful and `false` is returned.

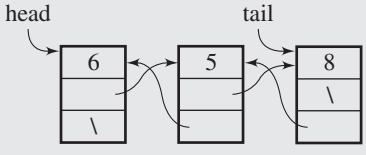
With reasoning similar to that used to determine the efficiency of `deleteNode()`, `isInList()` takes $O(1)$ time in the best case and $O(n)$ in the worst and average cases.

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 store only integers. If we wanted a linked list for float numbers or for 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 that 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.

FIGURE 3.9 A doubly linked list.



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

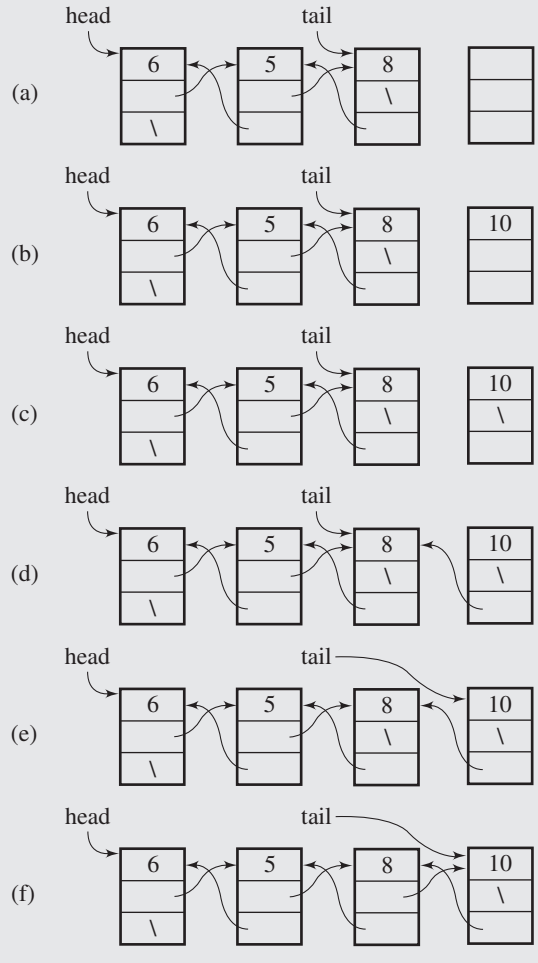
FIGURE 3.10 An implementation of a doubly linked list.

```
//*****************************  genDLList.h  ***************************
#ifndef DOUBLY_LINKED_LIST
#define DOUBLY_LINKED_LIST

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

template<class T>
class DoublyLinkedList {
public:
    DoublyLinkedList() {
        head = tail = 0;
    }
    void addToDLLTail(const T&);
    T deleteFromDLLTail();
    . . . . . .
protected:
    DLLNode<T> *head, *tail;
};
template<class T>
void DoublyLinkedList<T>::addToDLLTail(const T& el) {
    if (tail != 0) {
         tail = new DLLNode<T>(el,0,tail);
         tail->prev->next = tail;
    }
    else head = tail = new DLLNode<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;
```

FIGURE 3.10 *(continued)*

```
            delete tail->next;
            tail->next = 0;
        }
        return el;
    }
    . . . . . .
    #endif
```

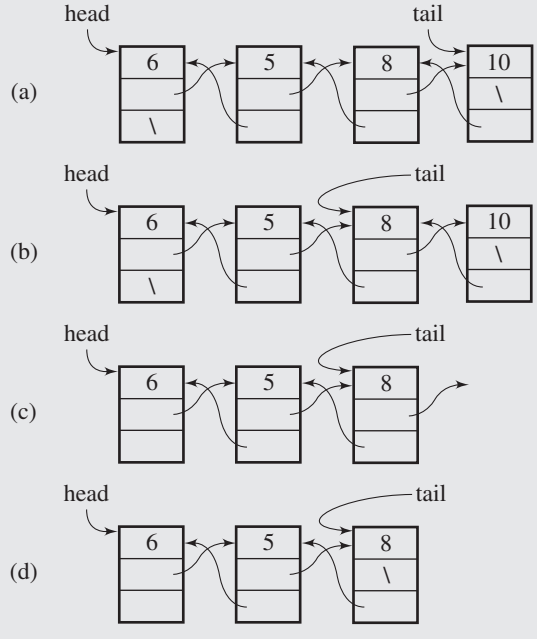
FIGURE 3.11 Adding a new node at the end of a doubly linked list.



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 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 the `prev` member to 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, the 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 the object stored in the removed node.

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



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 attempting to delete the last node. As with the singly linked list's `deleteFromHead()`, the caller should have an `if` statement

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

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.

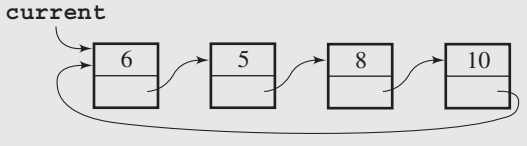
Because of the immediate accessibility of the last node, both `addToDLLTail()` and `deleteFromDLLTail()` execute in constant time $O(1)$.

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 ensure that each process has a fair share of the resource. 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 in 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.

FIGURE 3.13 A circular singly linked list.



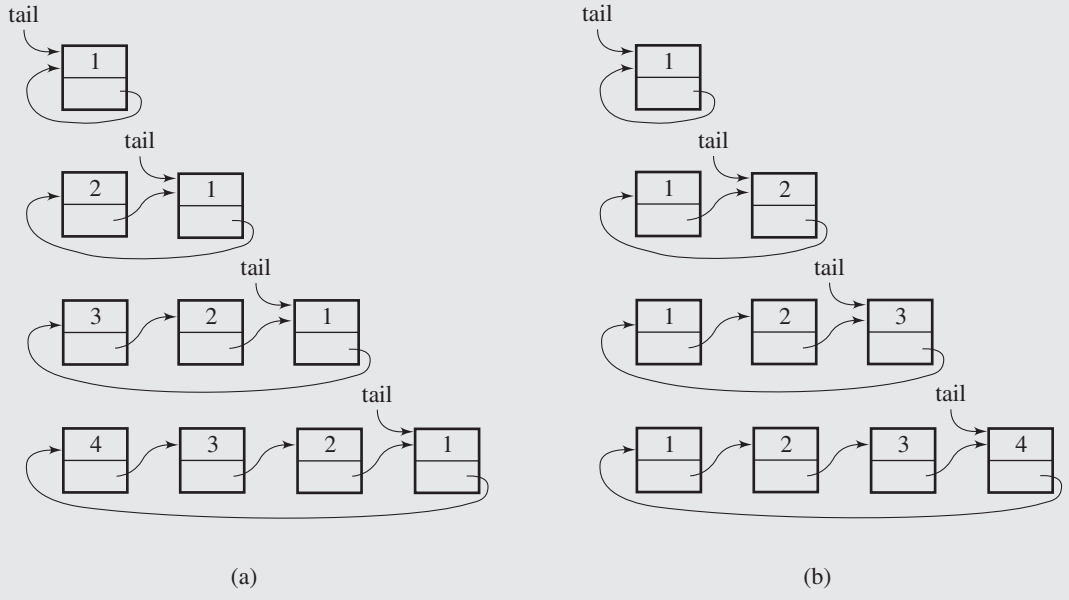
In an implementation of a circular singly linked list, we can use only one permanent pointer, `tail`, to the list even though operations on the list require access to the tail and its successor, the head. To that end, a linear singly linked list as discussed in Section 3.1 uses two permanent pointers, `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 IntSLLNode(el);
        tail->next = tail;
```

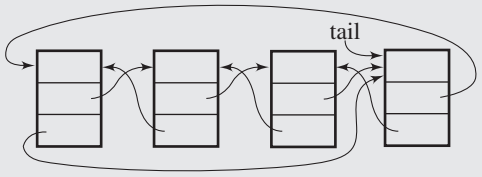
```
        }
        else {
            tail->next = new IntSLLNode(el,tail->next);
            tail = tail->next;
        }
    }
```

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



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 to its predecessor after deleting the node. 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 because there is direct access to the next to last node that needs to be updated in the case of such a deletion. In this list, both insertion and deletion of the tail node can be done in $O(1)$ time.

FIGURE 3.15 A circular doubly linked list.



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* because the 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.

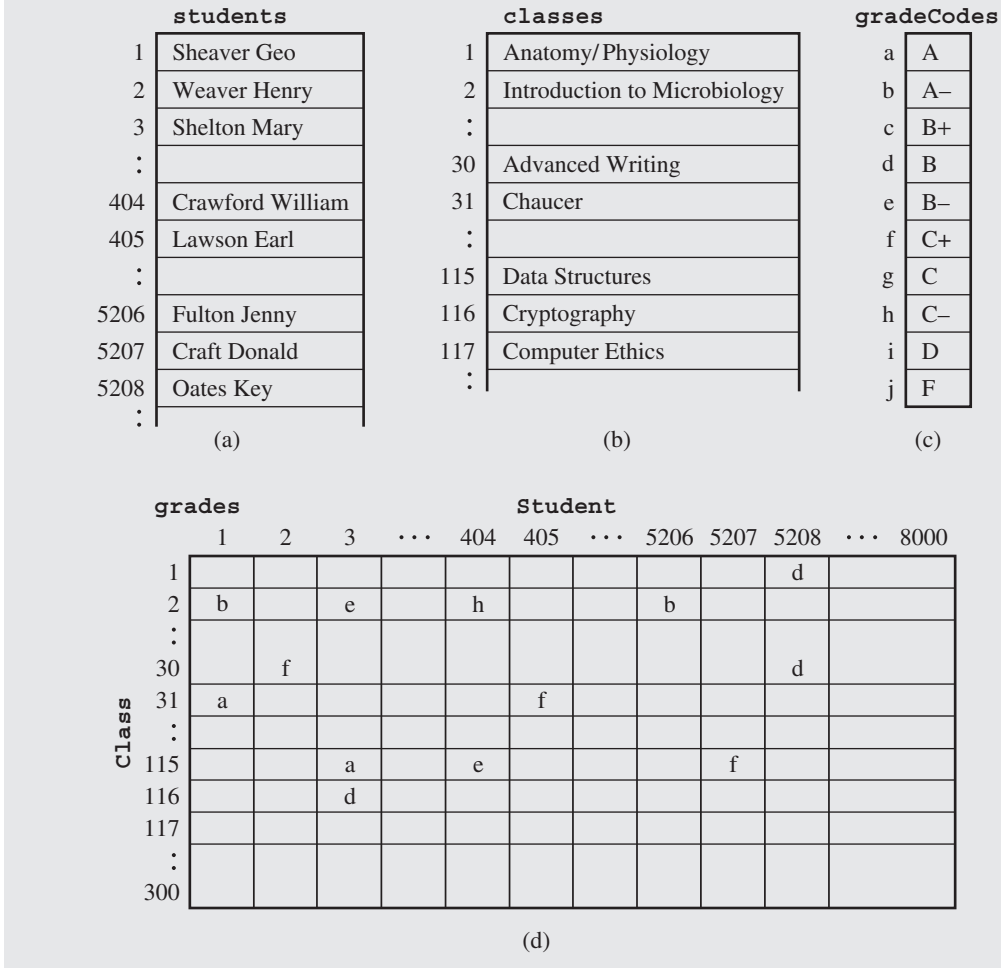
As an example, consider the problem of storing grades for all students in a university for a certain semester. Assume that there are 8,000 students and 300 classes. A natural implementation is a two-dimensional array `grades` where student numbers are indexes of the columns and class numbers are the 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 2 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 that requires only one byte of storage.

The entire table (Figure 3.21d) occupies 8,000 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.

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

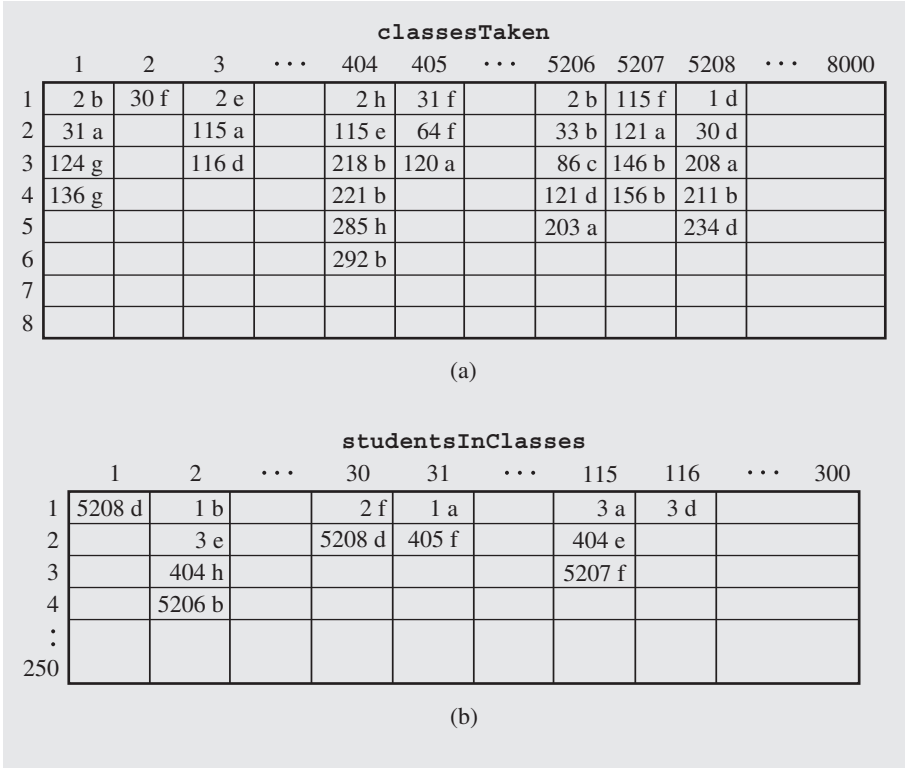
FIGURE 3.21 Arrays and sparse table used for storing student grades.



A better solution is to use two two-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 members: a student or class number and a grade. We assume that a student can take at most 8 classes and that there can be at most 250 students signed up for a class. We need two arrays, because 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 2 bytes to store an integer. With this new structure, 3 bytes are needed for

FIGURE 3.22 Two-dimensional arrays for storing student grades.

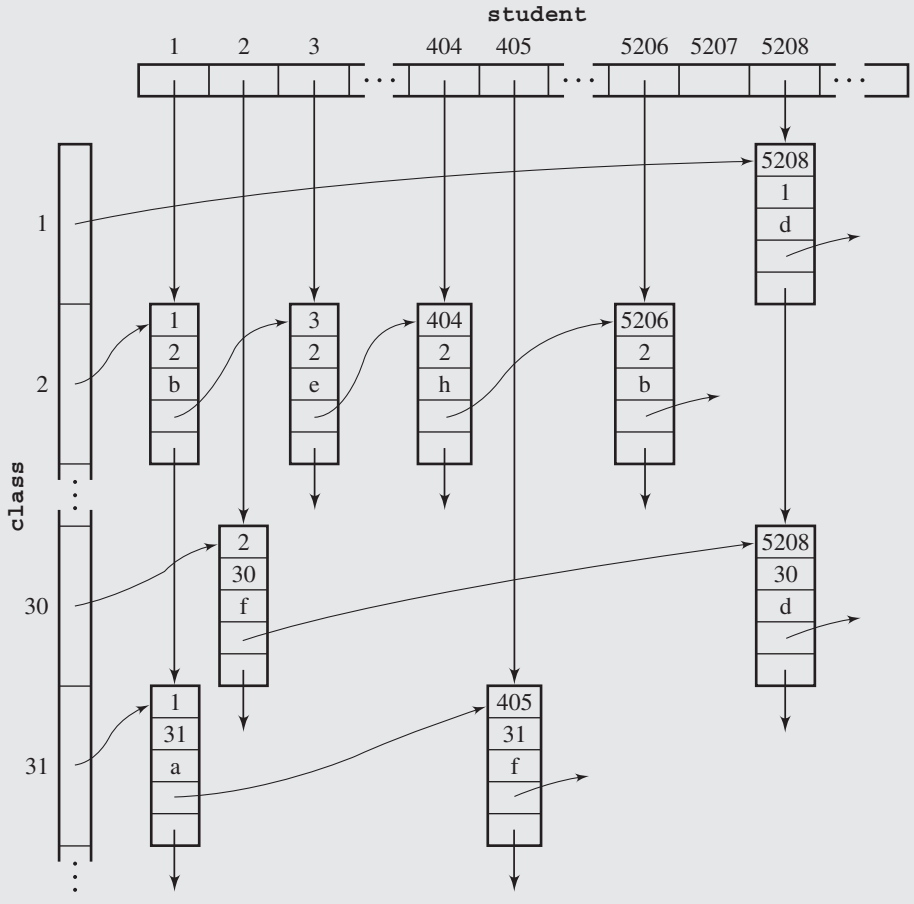


each cell. Therefore, `classesTaken` occupies 8,000 students · 8 classes · 3 bytes = 192,000 bytes, `studentsInClasses` occupies 300 classes · 250 students · 3 bytes = 225,000 bytes; both tables require a total of 417,000 bytes, less than one-fifth the number of bytes required for the sparse table in Figure 3.21.

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 because most classes have fewer than 250 students, and most students take fewer than 8 classes. This structure is also inflexible: if a class can be taken by more than 250 students, a problem occurs that has to be circumvented in an artificial way. One way is to create a nonexistent class that 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 2 bytes and one node occupies 9 bytes, the entire structure can be stored in 8,000 students · 4

FIGURE 3.23 Student grades implemented using linked lists.

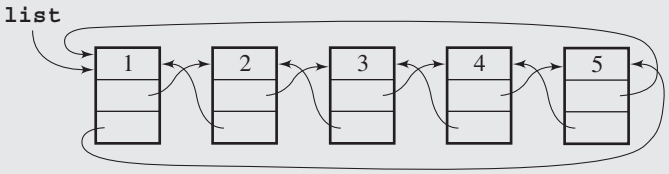


classes (on the average) · 9 bytes = 288,000 bytes, which is approximately 10% of the space required for the first implementation and about 70% of the space for the second. No space is used unnecessarily, there is no restriction imposed on the number of students per class, and the lists of students taking a class can be printed immediately.

3.10 EXERCISES

1. Assume that a circular doubly linked list has been created, as in Figure 3.29. After each of the following assignments, indicate changes made in the list by showing which links have been modified. The second assignment should make changes in the list modified by the first assignment, and so on.

FIGURE 3.29 A circular doubly linked list.




```
list->next->next->next = list->prev;

list->prev->prev->prev = list->next->next->next->prev;

list->next->next->next->prev = list->prev->prev->prev;

list->next = list->next->next;

list->next->prev->next = list->next->next->next;
```

2. How many nodes does the shortest linked list have? The longest linked list?
3. The linked list in Figure 3.1l was created in Section 3.2 with three assignments. Create this list with only one assignment.
4. Merge two ordered singly linked lists of integers into one ordered list.
5. Delete an *i*th node from a linked list. Be sure that such a node exists.
6. Delete from list L_1 nodes whose positions are to be found in an ordered list L_2. For instance, if L_1 = (A B C D E) and L_2 = (2 4 8), then the second and the fourth nodes are to be deleted from list L_1 (the eighth node does not exist), and after deletion, L_1 = (A C E).
7. Delete from list L_1 nodes occupying positions indicated in ordered lists L_2 and L_3. For instance, if L_1 = (A B C D E), L_2 = (2 4 8), and L_3 = (2 5), then after deletion, L_1 = (A C).
8. Delete from an ordered list L nodes occupying positions indicated in list L itself. For instance, if L = (1 3 5 7 8), then after deletion, L = (3 7).
9. A linked list does not have to be implemented with pointers. Suggest other implementations of linked lists.
10. Write a member function to check whether two singly linked lists have the same contents.

11. Write a member function to reverse a singly linked list using only one pass through the list.

12. Insert a new node into a singly linked list (a) before and (b) after a node pointed to by `p` in this list (possibly the first or the last). Do not use a loop in either operation.

13. Attach a singly linked list to the end of another singly linked list.

14. Put numbers in a singly linked list in ascending order. Use this operation to find the median in the list of numbers.

15. How can a singly linked list be implemented so that insertion requires no test for whether `head` is null?

16. Insert a node in the middle of a doubly linked list.

17. Write code for class `IntCircularSLList` for a circular singly linked list that includes equivalents of the member functions listed in Figure 3.2.

18. Write code for class `IntCircularDLList` for a circular doubly linked list that includes equivalents of the member functions listed in Figure 3.2.

3.11 PROGRAMMING ASSIGNMENTS

1. Farey fractions of level one are defined as sequence $(\frac{0}{1}, \frac{1}{1})$. This sequence is extended in level two to form a sequence $(\frac{0}{1}, \frac{1}{2}, \frac{1}{1})$, sequence $(\frac{0}{1}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{1}{1})$ at level three, sequence $(\frac{0}{1}, \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \frac{1}{1})$ at level four, so that at each level *n,* a new fraction $\frac{a+b}{c+d}$ is inserted between two neighbor fractions $\frac{a}{c}$ and $\frac{b}{d}$ only if $c + d \leq n$. Write a program that for a number *n* entered by the user creates—by constantly extending it—a linked list of fractions at level *n* and then displays them.

2. Write a simple airline ticket reservation program. The program should display a menu with the following options: reserve a ticket, cancel a reservation, check whether a ticket is reserved for a particular person, and display the passengers. The information is maintained on an alphabetized linked list of names. In a simpler version of the program, assume that tickets are reserved for only one flight. In a fuller version, place no limit on the number of flights. Create a linked list of flights with each node including a pointer to a linked list of passengers.

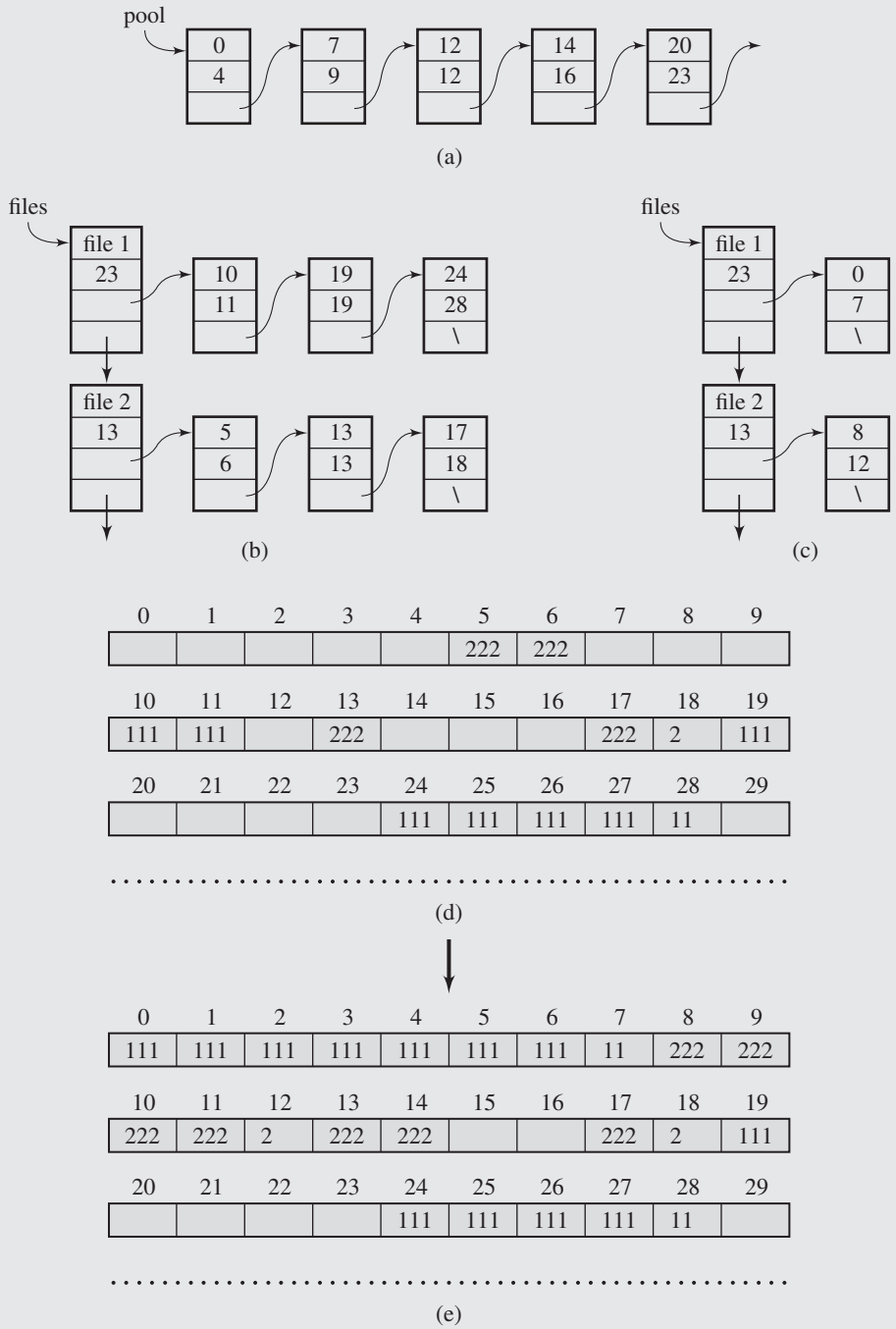
3. Read Section 12.1 about sequential-fit methods. Implement the discussed methods with linked lists and compare their efficiency.

4. Write a program to simulate managing files on disk. Define the disk as a one-dimensional array `disk` of size `numOfSectors*sizeOfSector`, where `sizeOfSector` indicates the number of characters stored in one sector. (For the sake of debugging, make it a very small number.) A pool of available sectors is kept in a linked list `sectors` of three field structures: two fields to indicate ranges of available sectors and one `next` field. Files are kept in a linked list `files` of four field structures: filename, the number of characters in the file, a pointer to a linked list of sectors where the contents of the file can be found, and the `next` field.

a. In the first part, implement functions to save and delete files. Saving files requires claiming a sufficient number of sectors from `pool`, if available. The sectors may not be contiguous, so the linked list assigned to the file may contain several nodes. Then the contents of the file have to be written to the sectors assigned to the file. Deletion of a file only requires removing the nodes corresponding with this file (one from `files` and the rest from its own linked list of sectors) and transferring the sectors assigned to this file back to `pool`. No changes are made in `disk`.

b. File fragmentation slows down file retrieval. In the ideal situation, one cluster of sectors is assigned to one file. However, after many operations with files, it may not be possible. Extend the program to include a function `together()` to transfer files to contiguous sectors; that is, to create a situation illustrated in Figure 3.30. Fragmented files `file1` and `file2` occupy only one cluster of sectors after

FIGURE 3.30 Linked lists used to allocate disk sectors for files: (a) a pool of available sectors; two files (b) before and (c) after putting them in contiguous sectors; the situation in sectors of the disk (d) before and (e) after this operation.



`together()` is finished. However, particular care should be taken not to overwrite sectors occupied by other files. For example, `file1` requires eight sectors; five sectors are free at the beginning of `pool`, but sectors 5 and 6 are occupied by `file2`. Therefore, a file *f* occupying such sectors has to be located first by scanning `files`. The contents of these sectors must be transferred to unoccupied positions, which requires updating the sectors belonging to *f* in the linked list; only then can the released sectors be utilized. One way of accomplishing this is by copying from the area into which one file is copied chunks of sectors of another file into an area of the disk large enough to accommodate these chunks. In the example in Figure 3.30, contents of `file1` are first copied to sectors 0 through 4, and then copying is temporarily suspended because sector 5 is occupied. Thus, contents of sectors 5 and 6 are moved to sector 12 and 14, and the copying of `file1` is resumed.

5. Write a simple line editor. Keep the entire text on a linked list, one line in a separate node. Start the program with entering `EDIT file`, after which a prompt appears along with the line number. If the letter `I` is entered with a number *n* following it, then insert the text to be followed before line *n.* If `I` is not followed by a number, then insert the text before the current line. If `D` is entered with two numbers *n* and *m,* one *n,* or no number following it, then delete lines *n* through *m,* line *n,* or the current line. Do the same with the command `L`, which stands for listing lines. If `A` is entered, then append the text to the existing lines. Entry `E` signifies exit and saving the text in a file. Here is an example:

```
 EDIT testfile
1> The first line
2>
3> And another line
4> I 3
3> The second line
4> One more line
5> L
1> The first line
2>
3> The second line
4> One more line
5> And another line    // This is now line 5, not 3;
5> D 2                 // line 5, since L was issued from line 5;
4> L                   // line 4, since one line was deleted;
1> The first line
2> The second line     // this and the following lines
3> One more line       // now have new numbers;
4> And another line
4> E
```

6. Extend the case study program in this chapter to have it store all the information in the file `Library` at exit and initialize all the linked lists using this information at the invocation of the program. Also, extend it by adding more error checking, such as not allowing the same book to be checked out at the same time to more than one patron or not including the same patron more than once in the library.

Stacks and Queues

4

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

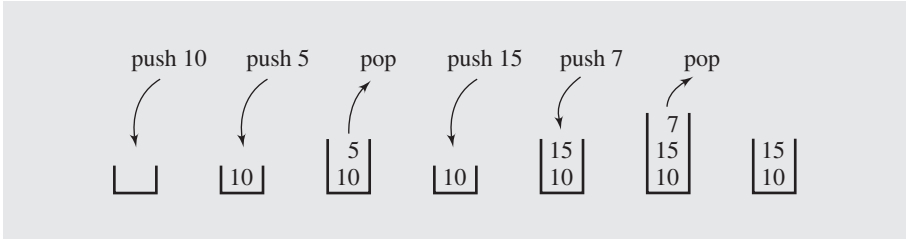
A *stack* is a linear data structure that can be accessed only at one of its ends for storing and retrieving data. Such a stack resembles a stack of trays in a cafeteria: new trays are put on the top of the stack and taken off the top. The last tray put on the stack is the first tray removed from the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

A tray can be taken only if there is at least one tray 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 that change its status and operations that 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.

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

FIGURE 4.1 A series of operations executed on a stack.



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 ')', ']', or '}'
            if ch and popped off delimiter do not match
               failure;
      else if ch is '/'
            read the next character;
            if this character is '*'
                  skip all characters until "*/" is found and report an error
                  if the end of file is reached before "*/" is encountered;
            else ch = the character read in;
                  continue; // go to the beginning of the loop;
   // 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`. 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 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.

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	;	

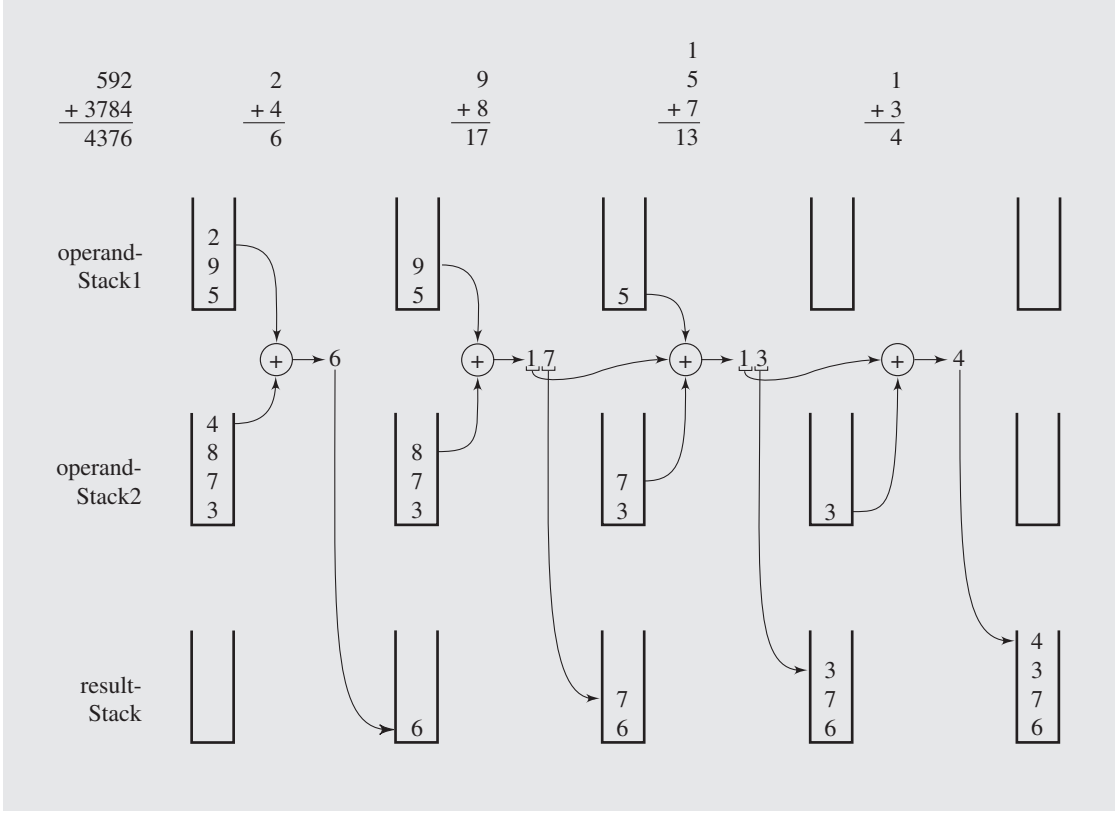
As another example of stack application, consider adding very large numbers. The largest magnitude of integers is limited, so we are not able to add 18,274,364,583,929,273,748,459,595,684,373 and 8,129,498,165,026,350,236, because integer variables cannot hold such large values, let alone their sum. 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;
    carry = 0;
    while at least one stack is not empty
        pop a number from each nonempty stack and add them to carry;
        push the unit part on the result stack;
        store carry in carry;
    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 the application of this algorithm. In this example, numbers 592 and 3,784 are added.

1. Numbers corresponding to digits composing the first number are pushed onto `operandStack1`, and numbers corresponding to the digits of 3,784 are pushed onto `operandStack2`. Note the order of digits on the stacks.

FIGURE 4.3 An example of adding numbers 592 and 3,784 using 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 `carry` 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 `carry`.
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.

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 4.4 A vector implementation of a stack.

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

#ifndef STACK
#define STACK

#include <vector>

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() {
```

FIGURE 4.4 *(continued)*

```
        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>

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

Continues

FIGURE 4.5 *(continued)*

```
    }
    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
```

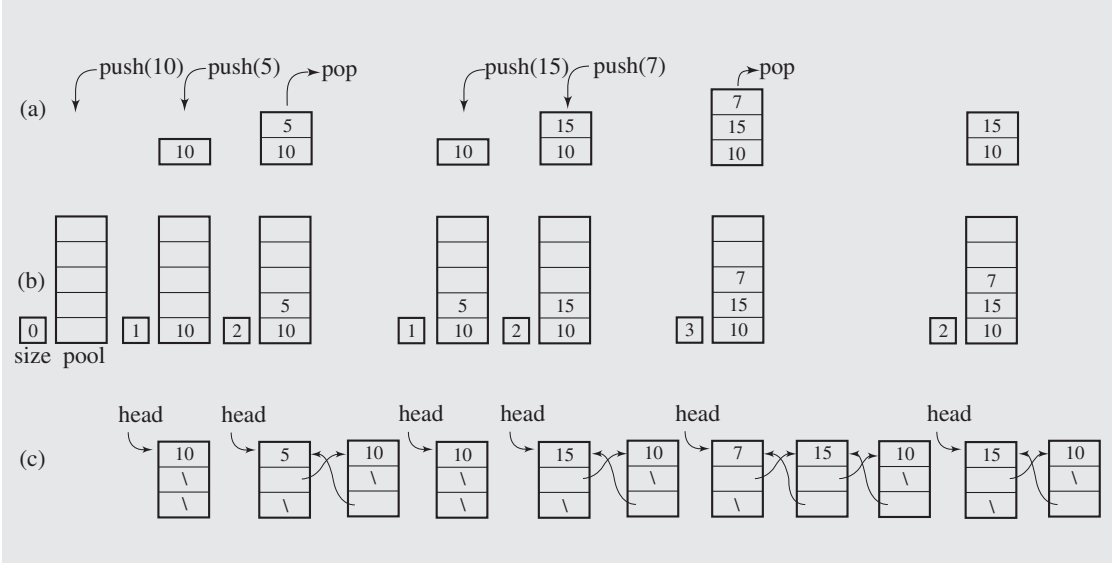
Figure 4.6 shows the same sequence of push and pop operations as Figure 4.1 with the changes that take place in the stack implemented as a vector (Figure 4.6b) and as a linked list (Figure 4.6c).

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 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 vector 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.

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



4.2 Queues

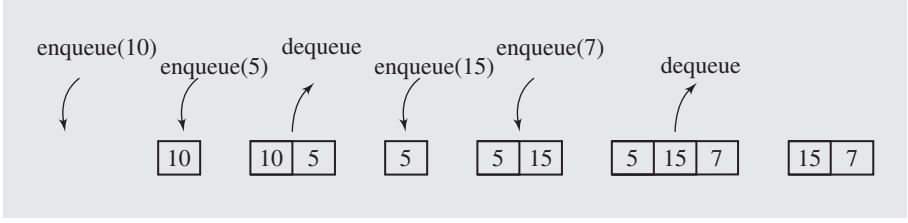
A *queue* is simply a waiting line that grows by adding elements to its end and shrinks by taking elements from its front. Unlike a stack, a queue is a structure in which both ends are used: one for adding new elements and one for removing them. Therefore, the last element has to wait until all elements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

FIGURE 4.7 A series of operations executed on a queue.



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.

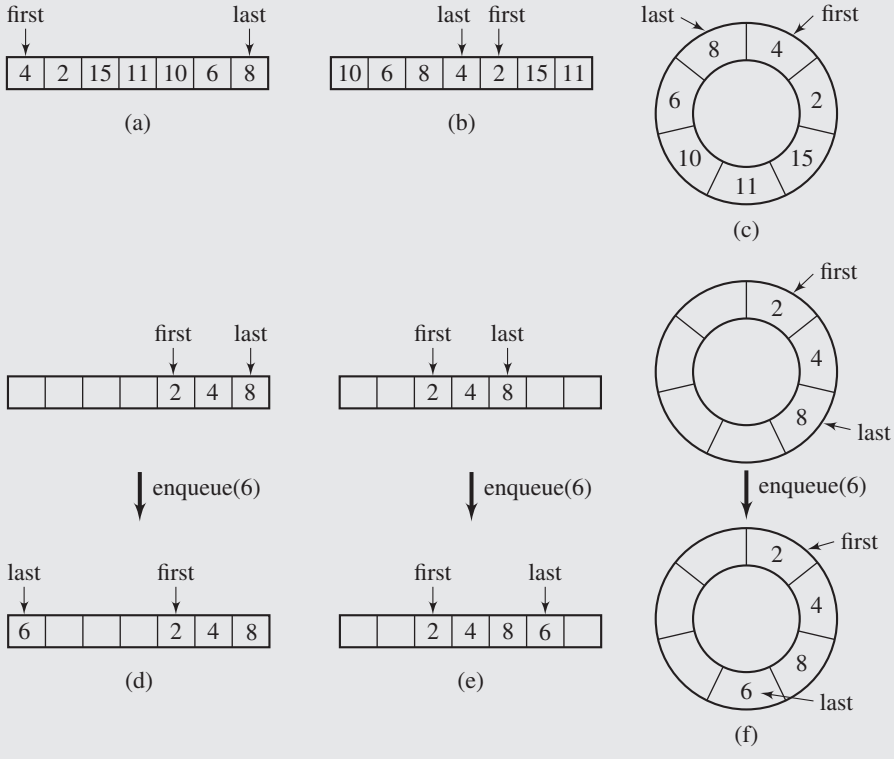
The poem is dedicated to Ruth Dymes, which is indicated not only by the last word of the poem, but also by reading in sequence the first letters of each line, which also spells Ruth. This type of poem is called an acrostic, and it is characterized by initial letters that form a word or phrase when taken 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.

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 the last element in the counterclockwise direction. 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

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. (f) Enqueuing number 6 to a queue storing 2, 4, and 8. (d–e) The same queue seen as a one-dimensional array with the last element (d) at the end of the array and (e) in the middle.



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 4.8d). If the last element is in any other position, then the new element is put after the last, space permitting (Figure 4.8e). These two situations must be distinguished when implementing a queue viewed as a circular array (Figure 4.8f).

Figure 4.9 contains possible implementations of member functions that 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

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() {
```

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>

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

Continues

FIGURE 4.10 *(continued)*

```
    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
```

primarily to scan the list and stop at the next to last node (see the 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 that 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 types 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 three 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 one-minute intervals throughout the day. For 15% of such intervals, no customers arrived, for 20%, only one arrived, and so on. Six clerks were employed, no lines were ever observed, and the bank management wanted to know whether six clerks were too

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

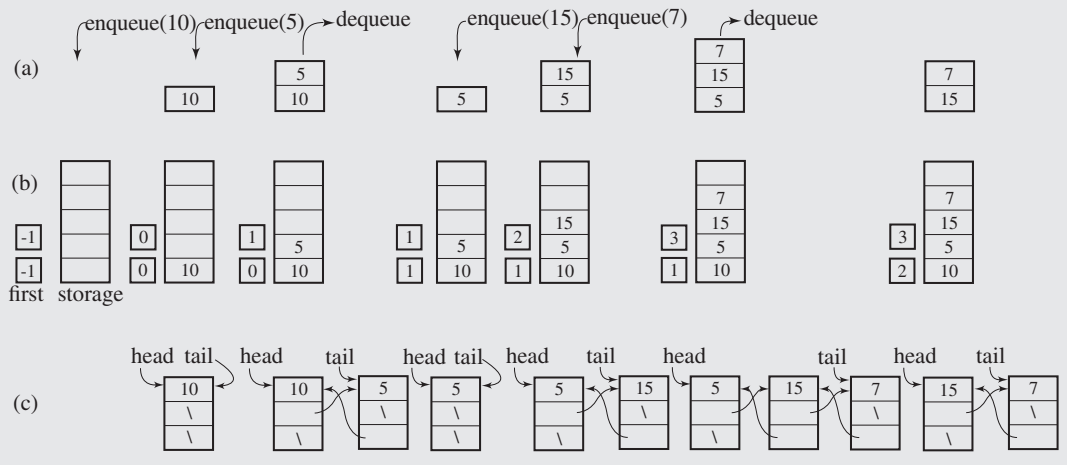


FIGURE 4.12 Bank One example: (a) data for number of arrived customers per one-minute interval and (b) transaction time in seconds per customer.

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)

Amount 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)

many. Would five suffice? Four? Maybe even three? Can lines be expected at any time? To answer these questions, a simulation program was written that applied the recorded data and checked 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 one-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 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, and so on, 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 one-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.

FIGURE 4.13 Bank One example: implementation code.

```
#include <iostream>
#include <cstdlib>

using namespace std;

#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;
}

int 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++) {
```

FIGURE 4.13 *(continued)*

```
        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()) {
             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.";
        return 0;
}
```

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 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 of 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, because 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, and the like). 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 that allows relatively fast enqueuing and dequeuing. Because 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 always having 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 10 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.

4.9 EXERCISES

1. Reverse the order of elements on stack S
   a. using two additional stacks
   b. using one additional queue
   c. using one additional stack and some additional nonarray variables
2. Put the elements on the stack S in ascending order using one additional stack and some additional nonarray variables.
3. Transfer elements from stack S_1 to stack S_2 so that the elements from S_2 are in the same order as on S_1
   a. using one additional stack
   b. using no additional stack but only some additional nonarray variables
4. Suggest an implementation of a stack to hold elements of two different types, such as structures and float numbers.
5. Using additional nonarray variables, order all elements on a queue using also
   a. two additional queues
   b. one additional queue
6. In this chapter, two different implementations were developed for a stack: class `Stack` and class `LLStack`. The names of member functions in both classes suggest that the same data structure is meant; however, a tighter connection between these two classes can be established. Define an abstract base class for a stack and derive from it both class `Stack` and class `LLStack`.
7. Define a stack in terms of a queue; that is, create a class

```
template <class T>
class StackQ {
     Queue<T> pool;
     . . . . . .
     void push(const T& el) {
          pool.enqueue(el);
          . . . . . .
```

8. Define a queue in terms of a stack.
9. A generic queue class could be defined in terms of a vector:

```
template<class T, int capacity = 30>
class QueueV {
    . . . . . .
private:
    vector<T> pool;
}
```

   Is this a viable solution?

10. Modify the program from the case study to print out the path without dead ends and, possibly, with detours. For example, for an input maze

```
1111111
1e00001
1110111
1000001
100m001
1111111
```

the program from the case study outputs the processed maze

```
1111111
1e....1
111.111
1.....1
1..m..1
1111111
Success
```

The modified program should, in addition, generate the path from the exit to the mouse:

```
[1 1] [1 2] [1 3] [2 3] [3 3] [3 4] [3 5] [4 5] [4 4] [4 3]
```

which leaves out two dead ends, `[1 4] [1 5]` and `[3 2] [3 1] [4 1] [4 2]`, but retains a detour, `[3 4] [3 5] [4 5] [4 4]`.

11. Modify the program from the previous exercise so that it prints the maze with the path without dead ends; the path is indicated by dashes and vertical bars to indicate the changes of direction of the path; for the input maze from the previous exercise, the modified program should output

```
1111111
1e--..1
111|111
1..|--1
1..m-|1
1111111
```

4.10 PROGRAMMING ASSIGNMENTS

1. Write a program that determines whether an input string is a palindrome; that is, whether it can be read the same way forward and backward. At each point, you can read only one character of the input string; do not use an array to first store this string and then analyze it (except, possibly, in a stack implementation). Consider using multiple stacks.

2. Write a program to convert a number from decimal notation to a number expressed in a number system whose base (or radix) is a number between 2 and 9. The conversion is performed by repetitious division by the base to which a number is being converted and then taking the remainders of division in the reverse order. For example, in converting to binary, number 6 requires three such divisions: 6/2 = 3 remainder 0, 3/2 = 1 remainder 1, and finally, 1/2 = 0 remainder 1. The remainders 0, 1, and 1 are put in reverse order so that the binary equivalent of 6 is equal to 110.

   Modify your program so that it can perform a conversion in the case when the base is a number between 11 and 27. Number systems with bases greater than 10 require more symbols. Therefore, use capital letters. For example, a hexadecimal system requires 16 digits: 0, 1, . . . , 9, A, B, C, D, E, F. In this system, decimal number 26 is equal to 1A in hexadecimal notation because 26/16 = 1 remainder 10 (that is, A), and 1/16 = 0 remainder 1.

3. Write a program that implements the algorithm `delimiterMatching()` from Section 4.1.

4. Write a program that implements the algorithm `addingLargeNumbers()` from Section 4.1.

5. Write a program to add any number of large integers. The problem can be approached in at least two ways.

   a. First, add two numbers and then repeatedly add the next number with the result of the previous addition.

   b. Create a vector of stacks and then use a generalized version of

Recursion

5

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 that 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, and so on. 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 ∈ **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 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 Section 9.3.3, 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

$$1, 2, \frac{5}{2}, \frac{29}{10}, \frac{941}{290}, \frac{969{,}581}{272{,}890}, \ldots$$

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 because 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

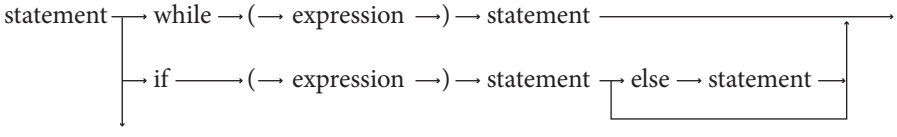
simplifies the computational process and allows us to find the answer for an integer n without computing the values for integers 0, 1, . . . , n – 1. For example, a definition of the sequence g,

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

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 because 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, local variables must be stored. If function `f1()`, which contains a declaration of a local 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 local variables, by the values of the function's parameters, and by the return address indicating where to restart in the calling function. The data area containing all this information is called an *activation record* or a *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 life span because they are dynamically allocated at function entry and deallocated upon exiting. Only the activation record of `main()` outlives every other activation record.

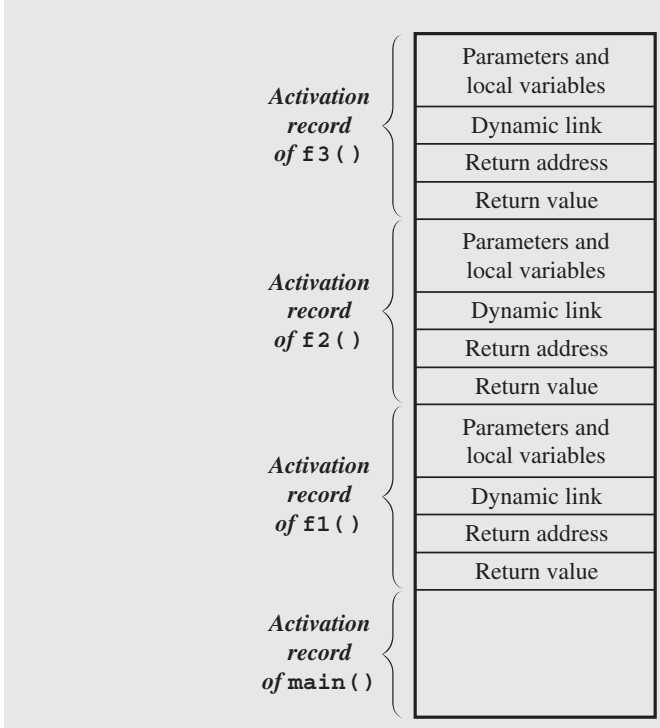
An activation record usually contains the following information:

- 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 variables that 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`. Because 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, 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 because the activation record for `f4()` is created on the stack and the activity of `f3()` is suspended.

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



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 receives $x \cdot x \cdot x$, multiplies it by x, and returns the final result. In this way, each new call increases the level of recursion, as follows:

call 1	$x^4 = x \cdot x^3 \quad = x \cdot x \cdot x \cdot x$
call 2	$x \cdot x^2 \quad = x \cdot x \cdot x$
call 3	$x \cdot x^1 \quad = x \cdot x$
call 4	$x \cdot x^0 = x \cdot 1 = x$
call 5	1

or alternatively, as

```
call 1              power(x,4)
call 2                    power(x,3)
call 3                          power(x,2)
call 4                                power(x,1)
call 5                                      power(x,0)
call 5                                      1
call 4                                x
call 3                          x·x
call 2                    x·x·x
call 1              x·x·x·x
```

What does the system do as the function is being executed? As we already know, the system keeps track of all calls on its run-time stack. Each line of code is assigned a number by the system,[1] and if a line is a function call, then its number is a return address. The address is used by the system to remember where to resume execution after the function has completed. For this example, assume that the lines in the function `power()` are assigned the numbers 102 through 105 and that it is called `main()` from the statement

```
          int main()
          { ...
/* 136 */  y = power(5.6,2);
            ...
          }
```

A trace of the recursive calls is relatively simple, as indicated by this diagram

```
call 1              power(5.6,2)
call 2                    power(5.6,1)
call 3                          power(5.6,0)
call 3                          1
call 2                    5.6
call 1              31.36
```

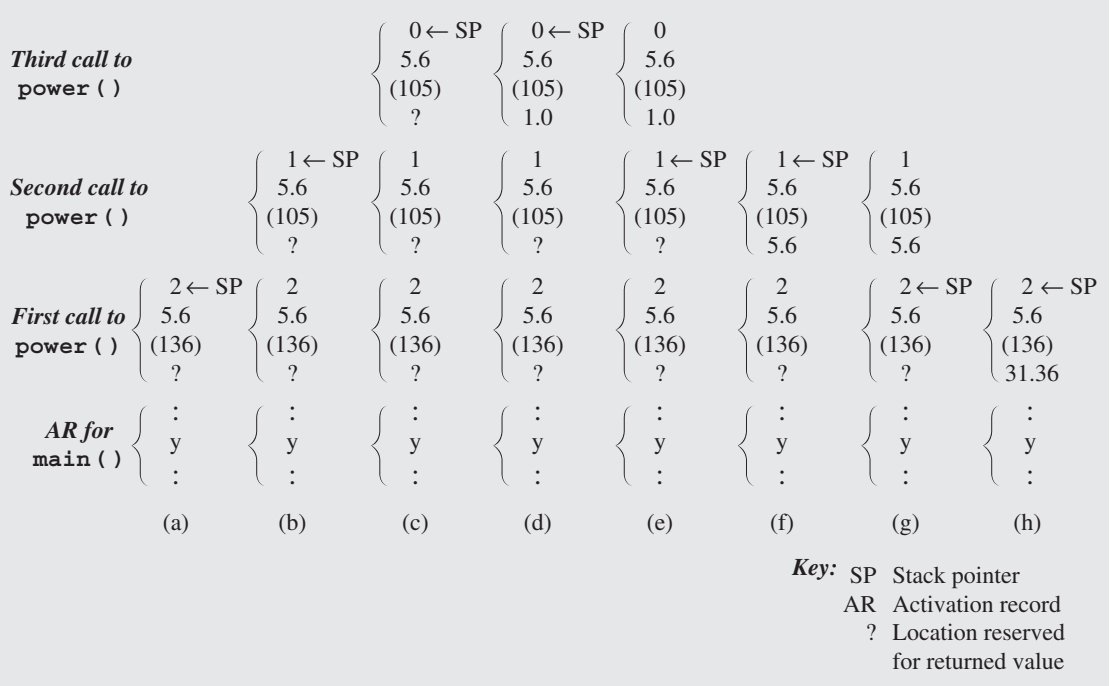
because most of the operations are performed on the run-time stack.

When the function is invoked for the first time, four items are pushed onto the run-time stack: the return address 136, the actual parameters 5.6 and 2, and one location reserved for the value returned by `power()`. Figure 5.2a represents this situation. (In this and subsequent diagrams, SP is a stack pointer, AR is an activation record, and question marks stand for locations reserved for the returned values. To distinguish values from addresses, the latter are parenthesized, although addresses are numbers exactly like function arguments.)

Now the function `power()` is executed. First, the value of the second argument, 2, is checked, and `power()` tries to return the value of 5.6 · `power(5.6,1)` because

[1]This is not quite precise because the system uses machine code rather than source code to execute programs. This means that one line of source program is usually implemented by several machine instructions.

FIGURE 5.2 Changes to the run-time stack during execution of `power(5.6,2)`.



that argument is not 0. This cannot be done immediately because the system does not know the value of `power(5.6,1)`; it must be computed first. Therefore, `power()` is called again with the arguments 5.6 and 1. But before this call is executed, the run-time stack receives new items, and its contents are shown in Figure 5.2b.

Again, the second argument is checked to see if it is 0. Because it is equal to 1, `power()` is called for the third time, this time with the arguments 5.6 and 0. Before the function is executed, the system remembers the arguments and the return address by putting them on the stack, not forgetting to allocate one cell for the result. Figure 5.2c contains the new contents of the stack.

Again, the question arises: is the second argument equal to zero? Because it finally is, a concrete value—namely, 1.0—can be returned and placed on the stack, and the function is finished without making any additional calls. At this point, there are two pending calls on the run-time stack—the calls to `power()`—that have to be completed. How is this done? The system first eliminates the activation record of `power()` that has just finished. This is performed logically by popping all its fields (the result, two arguments, and the return address) off the stack. We say "logically" because physically all these fields remain on the stack and only the SP is decremented appropriately. This is important because we do not want the result to be destroyed since it has not been used yet. Before and after completion of the last call of `power()`, the stack looks the same, but the SP's value is changed (see Figures 5.2d and 5.2e).

Now the second call to `power()` can complete because it waited for the result of the call `power(5.6,0)`. This result, 1.0, is multiplied by 5.6 and stored in the field allocated for the result. After that, the system can pop the current activation record off the stack by decrementing the SP, and it can finish the execution of the first call to `power()` that needed the result for the second call. Figure 5.2f shows the contents of the stack before changing the SP's value, and Figure 5.2g shows the contents of the stack after this change. At this moment, `power()` can finish its first call by multiplying the result of its second call, 5.6, by its first argument, also 5.6. The system now returns to the function that invoked `power()`, and the final value, 31.36, is assigned to `y`. Right before the assignment is executed, the content of the stack looks like Figure 5.2h.

The function `power()` can be implemented differently, without using any recursion, as in the following loop:

```
double nonRecPower(double x, unsigned int n) {
  double result = 1;
  for (result = x; n > 1; --n)
       result *= x;
  return result;
}
```

Do we gain anything by using recursion instead of a loop? The recursive version seems to be more intuitive because it is similar to the original definition of the power function. The definition is simply expressed in C++ without losing the original structure of the definition. The recursive version increases program readability, improves self-documentation, and simplifies coding. In our example, the code of the nonrecursive version is not substantially larger than in the recursive version, but for most recursive implementations, the code is shorter than it is in the nonrecursive implementations.

5.4 Tail Recursion

All recursive definitions contain a reference to a set or function being defined. There are, however, a variety of ways such a reference can be implemented. This reference can be done in a straightforward manner or in an intricate fashion, just once or many times. There may be many possible levels of recursion or different levels of complexity. In the following sections, some of these types are discussed, starting with the simplest case, *tail recursion.*

Tail recursion is characterized by the use of only one recursive call at the very end of a function implementation. In other words, when the call is made, there are no statements left to be executed by the function; the recursive call is not only the last statement but there are no earlier recursive calls, direct or indirect. For example, the function `tail()` defined as

```
void tail(int i) {
  if (i > 0) {
    cout << i << '';
    tail(i-1);
  }
}
```

is an example of a function with tail recursion, whereas the function `nonTail()` defined as

```
void nonTail(int i) {
  if (i > 0) {
    nonTail(i-1);
    cout << i << '';
    nonTail(i-1);
  }
}
```

is not. Tail recursion is simply a glorified loop and can be easily replaced by one. In this example, it is replaced by substituting a loop for the `if` statement and decrementing the variable `i` in accordance with the level of recursive call. In this way, `tail()` can be expressed by an iterative function:

```
void iterativeEquivalentOfTail(int i) {
  for ( ; i > 0; i--)
    cout << i << '';

}
```

Is there any advantage in using tail recursion over iteration? For languages such as C++, there may be no compelling advantage, but in a language such as Prolog, which has no explicit loop construct (loops are simulated by recursion), tail recursion acquires a much greater weight. In languages endowed with a loop or its equivalents, such as an `if` statement combined with a `goto` statement, tail recursion should not be used.

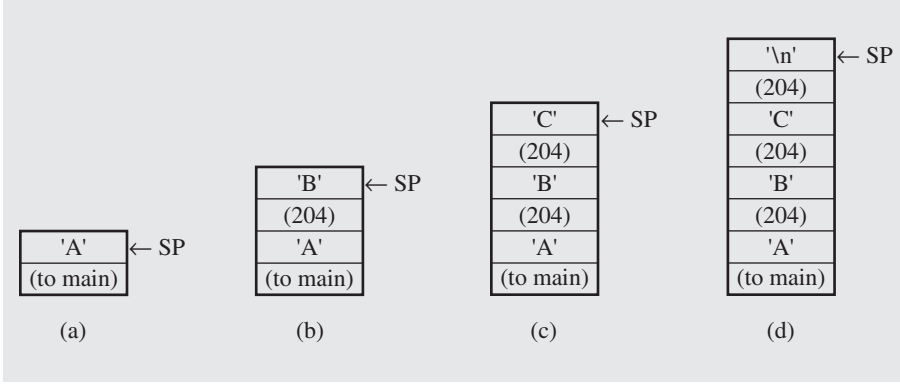
5.5 Nontail Recursion

Another problem that can be implemented in recursion is printing an input line in reverse order. Here is a simple recursive implementation:

```
/* 200 */ void reverse() {
                char ch;
/* 201 */       cin.get(ch);
/* 202 */       if (ch != '\n') {
/* 203 */           reverse();
/* 204 */           cout.put(ch);
                }
          }
```

Where is the trick? It does not seem possible that the function does anything. But it turns out that, by the power of recursion, it does exactly what it was designed for. `main()` calls `reverse()` and the input is the string: "ABC." First, an activation record is created with cells for the variable `ch` and the return address. There is no need to reserve a cell for a result, because no value is returned, which is indicated by using `void` in front of the function's name. The function `get()` reads in the first character, "A." Figure 5.3a shows the contents of the run-time stack right before `reverse()` calls itself recursively for the first time.

FIGURE 5.3 Changes on the run-time stack during the execution of `reverse()`.



The second character is read in and checked to see if it is the end-of-line character, and if not, `reverse()` is called again. But in either case, the value of `ch` is pushed onto the run-time stack along with the return address. Before `reverse()` is called for a third time (the second time recursively), there are two more items on the stack (see Figure 5.3b).

Note that the function is called as many times as the number of characters contained in the input string, including the end-of-line character. In our example, `reverse()` is called four times, and the run-time stack during the last call is shown in Figure 5.3d.

On the fourth call, `get()` finds the end-of-line character and `reverse()` executes no other statement. The system retrieves the return address from the activation record and discards this record by decrementing SP by the proper number of bytes. Execution resumes from line 204, which is a print statement. Because the activation record of the third call is now active, the value of `ch`, the letter "C," is output as the first character. Next, the activation record of the third call to `reverse()` is discarded and now SP points to where "B" is stored. The second call is about to be finished, but first, "B" is assigned to `ch` and then the statement on line 204 is executed, which results in printing "B" on the screen right after "C." Finally, the activation record of the first call to `reverse()` is reached. Then "A" is printed, and what can be seen on the screen is the string "CBA." The first call is finally finished and the program continues execution in `main()`.

Compare the recursive implementation with a nonrecursive version of the same function:

```
void simpleIterativeReverse() {
  char stack[80];
  register int top = 0;
  cin.getline(stack,80);
  for (top = strlen(stack) - 1; top >= 0; cout.put(stack[top--]));
}
```

The function is quite short and, perhaps, a bit more cryptic than its recursive counterpart. What is the difference then? Keep in mind that the brevity and relative simplicity of the second version are due mainly to the fact that we want to reverse

a string or array of characters. This means that functions like `strlen()` and `getline()` from the standard C++ library can be used. If we are not supplied with such functions, then our iterative function has to be implemented differently:

```
void iterativeReverse() {
  char stack[80];
  register int top = 0;
  cin.get(stack[top]);
  while(stack[top]!='\n')
    cin.get(stack[++top]);
  for (top -= 2; top >= 0; cout.put(stack[top--]));
}
```

The `while` loop replaces `getline()` and the autoincrement of variable `top` replaces `strlen()`. The `for` loop is about the same as before. This discussion is not purely theoretical because reversing an input line consisting of integers uses the same implementation as `iterativeReverse()` after changing the data type of `stack` from `char` to `int` and modifying the `while` loop.

Note that the variable name `stack` used for the array is not accidental. We are just making explicit what is done implicitly by the system. Our stack takes over the run-time stack's duty. Its use is necessary here because one simple loop does not suffice, as in the case of tail recursion. In addition, the statement `put()` from the recursive version has to be accounted for. Note also that the variable `stack` is local to the function `iterativeReverse()`. However, if it were a requirement to have a global stack object `st`, then this implementation can be written as

```
void nonRecursiveReverse() {
   int ch;
   cin.get(ch);
   while (ch != '\n') {
       st.push(ch);
       cin.get(ch);
   }
   while (!st.empty())
       cout.put(st.pop());
}
```

with the declaration `Stack<char> st` outside the function.

After comparing `iterativeReverse()` to `nonRecursiveReverse()`, we can conclude that the first version is better because it is faster, no function calls are made, and the function is self-sufficient, whereas `nonRecursiveReverse()` calls at least one function during each loop iteration, slowing down execution.

One way or the other, the transformation of nontail recursion into iteration usually involves the explicit handling of a stack. Furthermore, when converting a function from a recursive into an iterative version, program clarity can be diminished and the brevity of program formulation lost. Iterative versions of recursive C++ functions are not as verbose as in other programming languages, so program brevity may not be an issue.

To conclude this section, consider a construction of the von Koch snowflake. The curve was constructed in 1904 by Swedish mathematician Helge von Koch as an example of a continuous and nondifferentiable curve with an infinite length and yet encompassing a finite area. Such a curve is a limit of an infinite sequence of snowflakes, of which the first three are presented in Figure 5.4. As in real snowflakes, two of these curves have six petals, but to facilitate the algorithm, it is treated as a combination of three identical curves drawn in different angles and joined together. One such curve is drawn in the following fashion:

1. Divide an interval *side* into three even parts.
2. Move one-third of *side* in the direction specified by *angle*.
3. Turn to the right 60° (i.e., turn –60°) and go forward one-third of *side*.
4. Turn to the left 120° and proceed forward one-third of *side*.
5. Turn right 60° and again draw a line one-third of *side* long.

The result of these five steps is summarized in Figure 5.5. This line, however, becomes more jagged if every one of the four intervals became a miniature of the whole curve; that is, if the process of drawing four lines were made for each of these *side*/3 long intervals. As a result, 16 intervals *side*/9 long would be drawn. The process may be continued indefinitely—at least in theory. Computer graphics resolution prevents us from going too far because if lines are smaller than the diameter of a pixel, we just see one dot on the screen.

FIGURE 5.4 Examples of von Koch snowflakes.

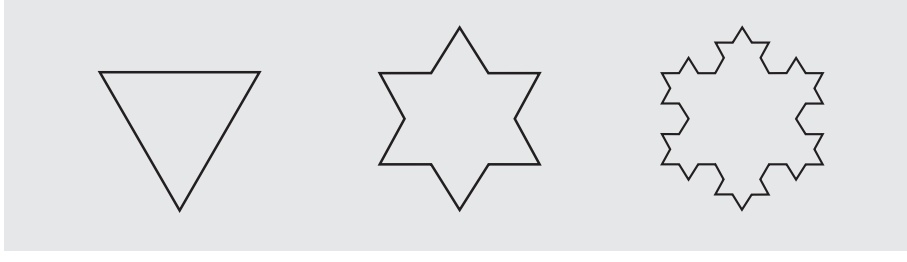
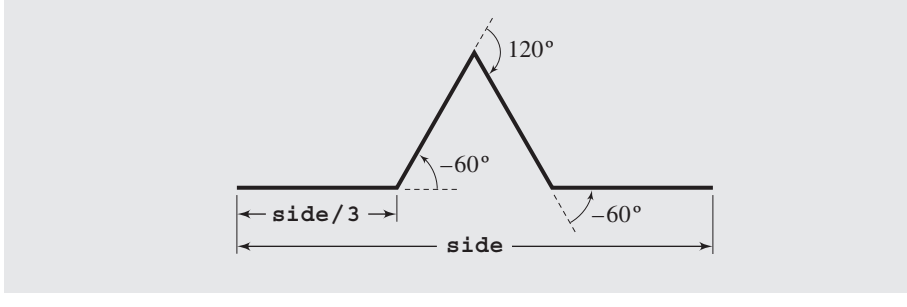


FIGURE 5.5 The process of drawing four sides of one segment of the von Koch snowflake.



The five steps that, instead of drawing one line of length *side,* draw four lines each of length one-third of *side* form one cycle only. Each of these four lines can also be compound lines drawn by the use of the described cycle. This is a situation in which recursion is well suited, which is reflected by the following pseudocode:

```
drawFourLines (side, level)
   if (level == 0)
     draw a line;
   else
     drawFourLines(side/3, level-1);
     turn left 60°;
     drawFourLines(side/3, level-1);
     turn right 120°;
     drawFourLines(side/3, level-1);
     turn left 60°;
     drawFourLines(side/3, level-1);
```

This pseudocode can be rendered almost without change into C++ code (Figure 5.6).

FIGURE 5.6 Recursive implementation of the von Koch snowflake.

```
//************************  vonKoch.h  ***************************
//                       Visual C++ program
//           A header file in a project of type MFC Application.

#define _USE_MATH_DEFINES
#include <cmath>

class vonKoch {
public:
    vonKoch(int,int,CDC*);
    void snowflake();
private:
    double side, angle;
    int level;
    CPoint currPt, pt;
    CDC *pen;
    void right(double x) {
        angle += x;
    }
    void left (double x) {
        angle -= x;
    }
    void drawFourLines(double side, int level);
};
```

FIGURE 5.6 (*continued*)

```
vonKoch::vonKoch(int s, int lvl, CDC *pDC) {
    pen = pDC;
    currPt.x = 200;
    currPt.y = 100;
    pen->MoveTo(currPt);
    angle = 0.0;
    side = s;
    level = lvl;
}

void vonKoch::drawFourLines(double side, int level) {
    // arguments to sin() and cos() are angles
    // specified in radians, i.e., the coefficient
    // PI/180 is necessary;
    if (level == 0) {
         pt.x = int(cos(angle*M_PI/180)*side) + currPt.x;
         pt.y = int(sin(angle*M_PI/180)*side) + currPt.y;
         pen->LineTo(pt);
         currPt.x = pt.x;
         currPt.y = pt.y;
    }
    else {
         drawFourLines(side/3,level-1);
         left (60);
         drawFourLines(side/3,level-1);
         right(120);
         drawFourLines(side/3,level-1);
         left (60);
         drawFourLines(side/3,level-1);
    }
}

void vonKoch::snowflake() {
    for (int i = 1; i <= 3; i++) {
        drawFourLines(side,level);
            right(120);
    }
}

// The function OnDraw() is generated by Visual C++ in snowflakeView.cpp
// when creating a snowflake project of type MFC Application;

#include "vonKoch.h"
```

Continues

FIGURE 5.6 (*continued*)

```
void CSnowflakeView::OnDraw(CDC* pDC)
{
    CSnowflakeDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
          return;

    // TODO: add draw code for native data here

    vonKoch(200,4,pDC).snowflake();
}
```

5.6 Indirect Recursion

The preceding sections discussed only direct recursion, where a function `f()` called itself. However, `f()` can call itself indirectly via a chain of other calls. For example, `f()` can call `g()`, and `g()` can call `f()`. This is the simplest case of indirect recursion.

The chain of intermediate calls can be of an arbitrary length, as in:

```
f() -> f1() -> f2() -> ··· -> fn() -> f()
```

There is also the situation when `f()` can call itself indirectly through different chains. Thus, in addition to the chain just given, another chain might also be possible. For instance

```
f() -> g1() -> g2() -> ··· -> gm() -> f()
```

This situation can be exemplified by three functions used for decoding information. `receive()` stores the incoming information in a buffer, `decode()` converts it into legible form, and `store()` stores it in a file. `receive()` fills the buffer and calls `decode()`, which in turn, after finishing its job, submits the buffer with decoded information to `store()`. After `store()` accomplishes its tasks, it calls `receive()` to intercept more encoded information using the same buffer. Therefore, we have the chain of calls

```
receive() -> decode() -> store() -> receive() -> decode() -> ···
```

which is finished when no new information arrives. These three functions work in the following manner:

```
receive(buffer)
   while buffer is not filled up
      if information is still incoming
         get a character and store it in buffer;
      else exit();
      decode(buffer);
```

```
decode(buffer)
    decode information in buffer;
    store(buffer);

store(buffer)
    transfer information from buffer to file;
    receive(buffer);
```

A more mathematically oriented example concerns formulas calculating the trigonometric functions sine, cosine, and tangent:

$$\sin(x) = \sin\left(\frac{x}{3}\right) \cdot \frac{(3 - \tan^2(\frac{x}{3}))}{(1 + \tan^2(\frac{x}{3}))}$$

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

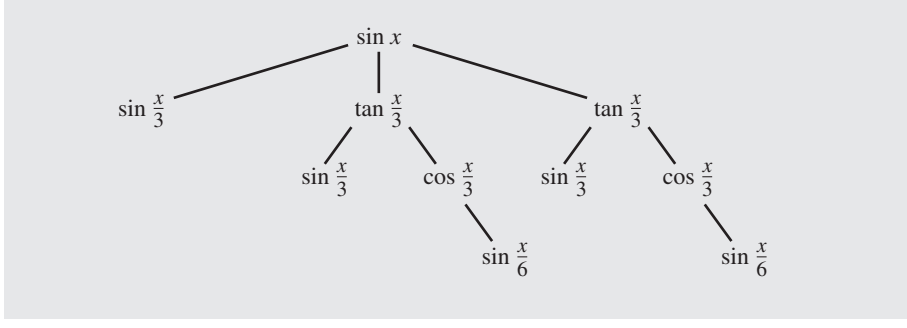
$$\cos(x) = 1 - \sin\left(\frac{x}{2}\right)$$

As usual in the case of recursion, there has to be an anchor in order to avoid falling into an infinite loop of recursive calls. In the case of sine, we can use the following approximation:

$$\sin(x) \approx x - \frac{x^3}{6}$$

where small values of x give a better approximation. To compute the sine of a number x such that its absolute value is greater than an assumed tolerance, we have to compute $\sin(\frac{x}{3})$ directly, $\sin(\frac{x}{3})$ indirectly through tangent, and also indirectly, $\sin(\frac{x}{6})$ through tangent and cosine. If the absolute value of $\frac{x}{3}$ is sufficiently small, which does not require other recursive calls, we can represent all the calls as a tree, as in Figure 5.7.

FIGURE 5.7 A tree of recursive calls for sin (x).



5.7 Nested Recursion

A more complicated case of recursion is found in definitions in which a function is not only defined in terms of itself, but also is used as one of the parameters. The following definition is an example of such a nesting:

$$h(n) = \begin{cases} 0 & \text{if } n = 0 \\ n & \text{if } n > 4 \\ h(2 + h(n)) & \text{if } n \le 4 \end{cases}$$

Function h has a solution for all $n \ge 0$. This fact is obvious for all $n > 4$ and $n = 0$, but it has to be proven for $n = 1, 2, 3$, and 4. Thus, $h(2) = h(2 + h(4)) = h(2 + h(2 + h(8))) = 12$. (What are the values of $h(n)$ for $n = 1, 3$, and 4?)

Another example of nested recursion is a very important function originally suggested by Wilhelm Ackermann in 1928 and later modified by Rozsa Peter:

$$A(n,m) = \begin{cases} m+1 & \text{if } n = 0 \\ A(n-1,1) & \text{if } n > 0,\ m = 0 \\ A(n-1,A(n,m-1)) & \text{otherwise} \end{cases}$$

This function is interesting because of its remarkably rapid growth. It grows so fast that it is guaranteed not to have a representation by a formula that uses arithmetic operations such as addition, multiplication, and exponentiation. To illustrate the rate of growth of the Ackermann function, we need only show that

$$A(3,m) = 2^{m+3} - 3$$

$$A(4,m) = 2^{2^{2^{\cdot^{\cdot^{\cdot^{2^{16}}}}}}} - 3$$

with a stack of m 2s in the exponent; $A(4,1) = 2^{2^{16}} - 3 = 2^{65536} - 3$, which exceeds even the number of atoms in the universe (which is 10^{80} according to current theories).

The definition translates very nicely into C++, but the task of expressing it in a nonrecursive form is truly troublesome.

5.8 Excessive Recursion

Logical simplicity and readability are used as an argument supporting the use of recursion. The price for using recursion is slowing down execution time and storing on the run-time stack more things than required in a nonrecursive approach. If recursion is too deep (for example, computing $5.6^{100,000}$), then we can run out of space on the stack and our program crashes. But usually, the number of recursive calls is much smaller than 100,000, so the danger of overflowing the stack may not be imminent.[2]

[2]Even if we try to compute the value of $5.6^{100,000}$ using an iterative algorithm, we are not completely free from a troublesome situation because the number is much too large to fit even a variable of double length. Thus, although the program would not crash, the computed value would be incorrect, which may be even more dangerous than a program crash.

However, if some recursive function repeats the computations for some parameters, the run time can be prohibitively long even for very simple cases.

Consider Fibonacci numbers. A sequence of Fibonacci numbers is defined as follows:

$$\text{Fib}(n) = \begin{cases} n & \text{if } n < 2 \\ \text{Fib}(n-2) + \text{Fib}(n-1) & \text{otherwise} \end{cases}$$

The definition states that if the first two numbers are 0 and 1, then any number in the sequence is the sum of its two predecessors. But these predecessors are in turn sums of their predecessors, and so on, to the beginning of the sequence. The sequence produced by the definition is

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

How can this definition be implemented in C++? It takes almost term-by-term translation to have a recursive version, which is

```
unsigned long Fib(unsigned long n) {
  if (n < 2)
    return n;
// else
    return Fib(n-2) + Fib(n-1);
}
```

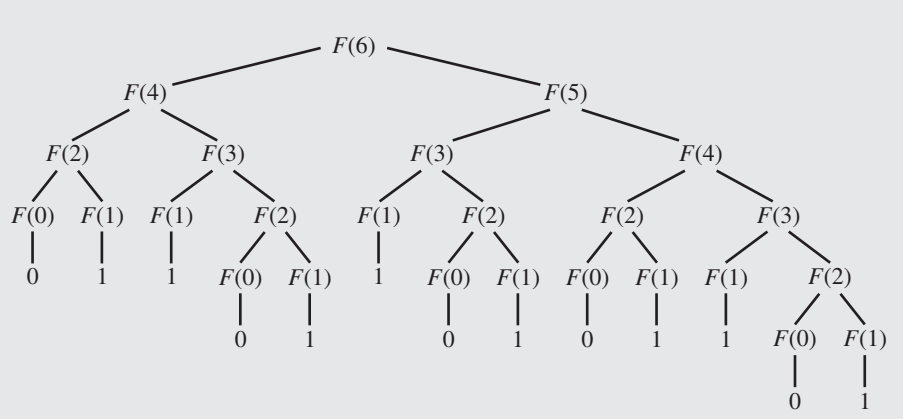
The function is simple and easy to understand but extremely inefficient. To see it, compute `Fib(6)`, the seventh number of the sequence, which is 8. Based on the definition, the computation runs as follows:

```
Fib(6) =                Fib(4)                    + Fib(5)
       =    Fib(2)        +        Fib(3)         + Fib(5)
       = Fib(0)+Fib(1)    +        Fib(3)         + Fib(5)
       =   0   +   1      +        Fib(3)         + Fib(5)
       =       1          + Fib(1)+   Fib(2)      + Fib(5)
       =       1          + Fib(1)+Fib(0)+Fib(1) + Fib(5)
```

etc.

This is just the beginning of our calculation process, and even here there are certain shortcuts. All these calculations can be expressed more concisely in the form of the tree shown in Figure 5.8. Tremendous inefficiency results because `Fib()` is called 25 times to determine the seventh element of the Fibonacci sequence. The source of this inefficiency is the repetition of the same calculations because the system forgets what has already been calculated. For example, `Fib()` is called eight times with parameter `n` = 1 to decide that 1 can be returned. For each number of the sequence, the function computes all its predecessors without taking into account that it suffices to do this only once. To find `Fib(6)` = 8, it computes `Fib(5)`, `Fib(4)`, `Fib(3)`, `Fib(2)`, `Fib(1)`, and `Fib(0)` first. To determine these values, `Fib(4)`, . . . , `Fib(0)` have to be computed to know the value of `Fib(5)`. Independently of this, the chain of computations `Fib(3)`, . . . , `Fib(0)` is executed to find `Fib(4)`.

FIGURE 5.8 The tree of calls for `Fib(6)`.



We can prove that the number of additions required to find `Fib(n)` using a recursive definition is equal to Fib$(n + 1) - 1$. Counting two calls per one addition plus the very first call means that `Fib()` is called $2 \cdot \text{Fib}(n + 1) - 1$ times to compute `Fib(n)`. This number can be exceedingly large for fairly small *n*s, as the table in Figure 5.9 indicates.

It takes almost a quarter of a million calls to find the twenty-sixth Fibonacci number, and nearly 3 million calls to determine the thirty-first! This is too heavy a price for the simplicity of the recursive algorithm. As the number of calls and the run time grow exponentially with *n,* the algorithm has to be abandoned except for very small numbers.

FIGURE 5.9 Number of addition operations and number of recursive calls to calculate Fibonacci numbers.

`n`	`Fib(n+1)`	Number of Additions	Number of Calls
6	13	12	25
10	89	88	177
15	987	986	1,973
20	10,946	10,945	21,891
25	121,393	121,392	242,785
30	1,346,269	1,346,268	2,692,537

An iterative algorithm may be produced rather easily as follows:

```
unsigned long iterativeFib(unsigned long n) {
    if (n < 2)
         return n;
    else {
         register long i = 2, tmp, current = 1, last = 0;
         for ( ; i <= n; ++i) {
             tmp = current;
             current += last;
             last = tmp;
         }
         return current;
    }
}
```

For each `n > 1`, the function loops `n - 1` times making three assignments per iteration and only one addition, disregarding the autoincrement of `i` (see Figure 5.10).

However, there is another, numerical method for computing Fib(n), using a formula discovered by Abraham de Moivre:

$$\text{Fib}(n) = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}$$

where $\phi = \frac{1}{2}(1+\sqrt{5})$ and $\hat{\phi} = 1 - \phi = \frac{1}{2}(1-\sqrt{5}) \approx -0.618034$. Because $-1 < \hat{\phi} < 0$, $\hat{\phi}^n$ becomes very small when n grows. Therefore, it can be omitted from the formula and

$$\text{Fib}(n) = \frac{\phi^n}{\sqrt{5}}$$

approximated to the nearest integer. This leads us to the third implementation for computing a Fibonacci number. To round the result to the nearest integer, we use the function `ceil` (for ceiling):

FIGURE 5.10 Comparison of iterative and recursive algorithms for calculating Fibonacci numbers.

		Assignments	
n	Number of Additions	Iterative Algorithm	Recursive Algorithm
6	5	15	25
10	9	27	177
15	14	42	1,973
20	19	57	21,891
25	24	72	242,785
30	29	87	2,692,537

```
unsigned long deMoivreFib(unsigned long n) {
    return ceil(exp(n*log(1.6180339897) - log(2.2360679775)) - .5);
}
```

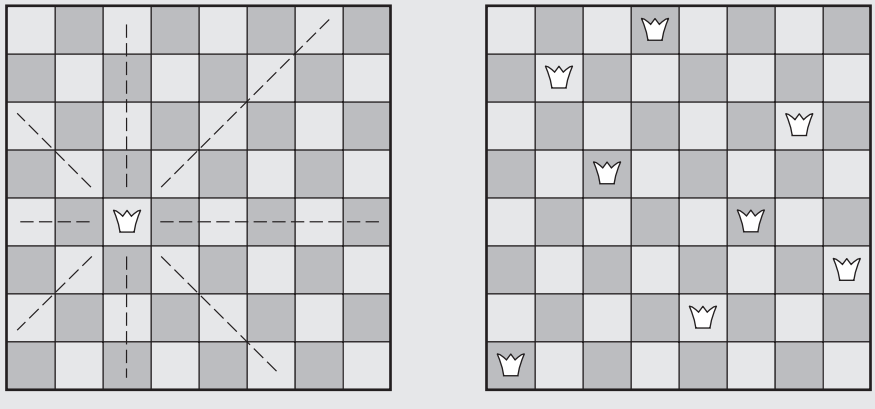
Try to justify this implementation using the definition of logarithm.

5.9 Backtracking

In solving some problems, a situation arises where there are different ways leading from a given position, none of them known to lead to a solution. After trying one path unsuccessfully, we return to this crossroads and try to find a solution using another path. However, we must ensure that such a return is possible and that all paths can be tried. This technique is called *backtracking,* and it allows us to systematically try all available avenues from a certain point after some of them lead to nowhere. Using backtracking, we can always return to a position that offers other possibilities for successfully solving the problem. This technique is used in artificial intelligence, and one of the problems in which backtracking is very useful is the eight queens problem.

The eight queens problem attempts to place eight queens on a chessboard in such a way that no queen is attacking any other. The rules of chess say that a queen can take another piece if it lies on the same row, on the same column, or on the same diagonal as the queen (see Figure 5.11). To solve this problem, we try to put the first queen on the board, then the second so that it cannot take the first, then the third so that it is not in conflict with the two already placed, and so on, until all of the queens are placed. What happens if, for instance, the sixth queen cannot be placed in a nonconflicting position? We choose another position for the fifth queen and try again with the sixth. If this does not work, the fifth queen is moved again. If all the possible positions for the fifth queen have been tried, the fourth queen is moved and then the

FIGURE 5.11 The eight queens problem.



(a) (b)

process restarts. This process requires a great deal of effort, most of which is spent backtracking to the first crossroads offering some untried avenues. In terms of code, however, the process is rather simple due to the power of recursion, which is a natural implementation of backtracking. Pseudocode for this backtracking algorithm is as follows (the last line pertains to backtracking):

```
putQueen(row)
   for every position col on the same row
     if position col is available
        place the next queen in position col;
        if (row < 8)
          putQueen(row+1);
        else success;
        remove the queen from position col;
```

This algorithm finds all possible solutions without regard to the fact that some of them are symmetrical.

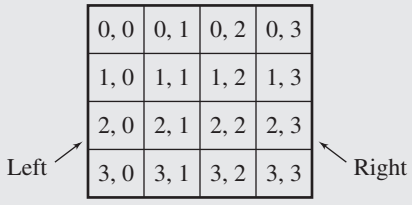
The most natural approach for implementing this algorithm is to declare an 8×8 array `board` of 1s and 0s representing a chessboard. The array is initialized to 1s, and each time a queen is put in a position (r, c), `board[r][c]` is set to 0. Also, a function must set to 0, as not available, all positions on row r, in column c, and on both diagonals that cross each other in position (r, c). When backtracking, the same positions (that is, positions on corresponding row, column, and diagonals) have to be set back to 1, as again available. Because we can expect hundreds of attempts to find available positions for queens, the setting and resetting process is the most time-consuming part of the implementation; for each queen, between 22 and 28 positions have to be set and then reset, 15 for row and column, and between 7 and 13 for diagonals.

In this approach, the board is viewed from the perspective of the player who sees the entire board along with all the pieces at the same time. However, if we focus solely on the queens, we can consider the chessboard from their perspective. For the queens, the board is not divided into squares, but into rows, columns, and diagonals. If a queen is placed on a single square, it resides not only on this square, but on the entire row, column, and diagonal, treating them as its own temporary property. A different data structure can be utilized to represent this.

To simplify the problem for the first solution, we use a 4×4 chessboard instead of the regular 8×8 board. Later, we can make the rather obvious changes in the program to accommodate a regular board.

Figure 5.12 contains the 4×4 chessboard. Notice that indexes in all fields in the indicated left diagonal all add up to two, $r + c = 2$; this number is associated with this diagonal. There are seven left diagonals, 0 through 6. Indexes in the fields of the indicated right diagonal all have the same difference, $r - c = -1$, and this number is unique among all right diagonals. Therefore, right diagonals are assigned numbers –3 through 3. The data structure used for all left diagonals is simply an array indexed by numbers 0 through 6. For right diagonals, it is also an array, but it cannot be indexed by negative numbers. Therefore, it is an array of seven cells, but to account for negative values obtained from the formula $r - c$, the same number is always added to it so as not to cross the bounds of this array.

FIGURE 5.12 A 4 × 4 chessboard.



An analogous array is also needed for columns, but not for rows, because a queen *i* is moved along row *i* and all queens < *i* have already been placed in rows < *i*. Figure 5.13 contains the code to implement these arrays. The program is short due to recursion, which hides some of the goings-on from the user's sight.

FIGURE 5.13 Eight queens problem implementation.

```
class ChessBoard {
public:
    ChessBoard();    // 8 x 8 chessboard;
    ChessBoard(int); // n x n chessboard;
    void findSolutions();
private:
    const bool available;
    const int squares, norm;
    bool *column, *leftDiagonal, *rightDiagonal;
    int  *positionInRow, howMany;
    void putQueen(int);
    void printBoard(ostream&);
    void initializeBoard();
};

ChessBoard::ChessBoard() : available(true), squares(8), norm(squares-1)
{
    initializeBoard();
}
ChessBoard::ChessBoard(int n) : available(true), squares(n),
norm(squares-1) {
    initializeBoard();
}
```

FIGURE 5.13 (*continued*)

```
void ChessBoard::initializeBoard() {
    register int i;
    column = new bool[squares];
    positionInRow = new int[squares];
    leftDiagonal  = new bool[squares*2 - 1];
    rightDiagonal = new bool[squares*2 - 1];
    for (i = 0; i < squares; i++)
        positionInRow[i] = -1;
    for (i = 0; i < squares; i++)
        column[i] = available;
    for (i = 0; i < squares*2 - 1; i++)
        leftDiagonal[i] = rightDiagonal[i] = available;
    howMany = 0;
}
void ChessBoard::putQueen(int row) {
    for (int col = 0; col < squares; col++)
        if (column[col] == available &&
            leftDiagonal [row+col] == available &&
            rightDiagonal[row-col+norm] == available) {
            positionInRow[row] = col;
            column[col] = !available;
            leftDiagonal[row+col] = !available;
            rightDiagonal[row-col+norm] = !available;
            if (row < squares-1)
                 putQueen(row+1);
            else printBoard(cout);
            column[col] = available;
            leftDiagonal[row+col] = available;
            rightDiagonal[row-col+norm] = available;
        }
}
void ChessBoard::findSolutions() {
    putQueen(0);
    cout << howMany << " solutions found.\n";
}
```

Figures 5.14 through 5.17 document the steps taken by `putQueen()` to place four queens on the chessboard. Figure 5.14 contains the move number, queen number, and row and column number for each attempt to place a queen. Figure 5.15 contains the changes to the arrays `positionInRow`, `column`, `leftDiagonal`, and

FIGURE 5.14 Steps leading to the first successful configuration of four queens as found by the function `putQueen()`.

Move	Queen	row	col	
{1}	1	0	0	
{2}	2	1	2	failure
{3}	2	1	3	
{4}	3	2	1	failure
{5}	1	0	1	
{6}	2	1	3	
{7}	3	2	0	
{8}	4	3	2	

FIGURE 5.15 Changes in the four arrays used by function `putQueen()`.

positionInRow	column	leftDiagonal	rightDiagonal	row
(0 , 2 , ,)	(!a , a , !a , a)	(!a, a , a , !a , a , a , a)	(a, a , !a , !a , a , a , a)	0 , 1
{1}{2}	{1} {2}	{1} {2}	{2}{1}	{1}{2}
(0 , 3 , 1 ,)	(!a , !a , a , !a)	(!a, a , a , !a , !a , a , a)	(a, !a , a , !a , !a , a , a)	1 , 2
{1}{3}{4}	{1} {4} {3}	{1} {4}{3}	{3} {1} {4}	{3}{4}
(1 , 3 , 0 , 2)	(!a , !a , !a , !a)	(a, !a , !a , a , !a , !a , a)	(a, !a , !a , a , !a , !a , a)	0 , 1 , 2 , 3
{5} {6} {7} {8}	{7} {5} {8} {6}	{5} {7} {6} {8}	{6} {5} {8} {7}	{5}{6}{7}{8}

`rightDiagonal`. Figure 5.16 shows the changes to the run-time stack during the eight steps. All changes to the run-time stack are depicted by an activation record for each iteration of the `for` loop, which mostly lead to a new invocation of `putQueen()`. Each activation record stores a return address and the values of `row` and `col`. Figure 5.17 illustrates the changes to the chessboard. A detailed description of each step follows.

{1} We start by trying to put the first queen in the upper left corner (0, 0). Because it is the very first move, the condition in the `if` statement is met, and the queen is placed in this square. After the queen is placed, the column 0, the main right diagonal, and the leftmost diagonal are marked as unavailable. In Figure 5.15, {1} is put underneath cells reset to `!available` in this step.

FIGURE 5.16 Changes on the run-time stack for the first successful completion of `putQueen()`.

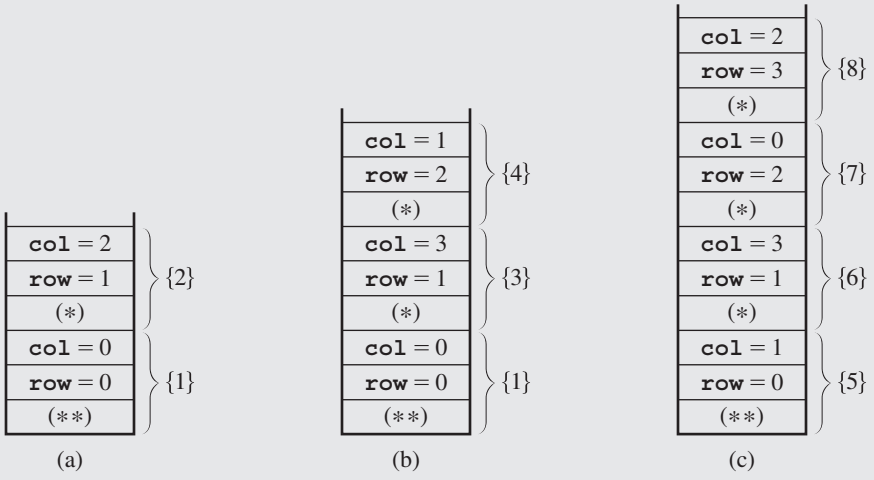
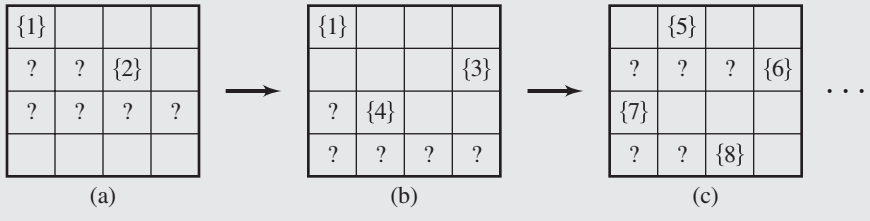


FIGURE 5.17 Changes to the chessboard leading to the first successful configuration.



{2} Since `row<3`, `putQueen()` calls itself with `row+1`, but before its execution, an activation record is created on the run-time stack (see Figure 5.16a). Now we check the availability of a field on the second row (i.e., `row==1`). For `col==0`, column 0 is guarded, for `col==1`, the main right diagonal is checked, and for `col==2`, all three parts of the `if` statement condition are true. Therefore, the second queen is placed in position (1, 2), and this fact is immediately reflected in the proper cells of all four arrays. Again, `row<3`. `putQueen()` is called trying to locate the third queen in row 2. After all the positions

in this row, 0 through 3, are tested, no available position is found, the `for` loop is exited without executing the body of the `if` statement, and this call to `putQueen()` is complete. But this call was executed by `putQueen()` dealing with the second row, to which control is now returned.

{3} Values of `col` and `row` are restored and the execution of the second call of `putQueen()` continues by resetting some fields in three arrays back to `available`, and since `col==2`, the `for` loop can continue iteration. The test in the `if` statement allows the second queen to be placed on the board, this time in position (1, 3).

{4} Afterward, `putQueen()` is called again with `row==2`, the third queen is put in (2, 1), and after the next call to `putQueen()`, an attempt to place the fourth queen is unsuccessful (see Figure 5.17b). No calls are made, the call from step {3} is resumed, and the third queen is once again moved, but no position can be found for it. At the same time, `col` becomes 3, and the `for` loop is finished.

{5} As a result, the first call of `putQueen()` resumes execution by placing the first queen in position (0, 1).

{6–8} This time execution continues smoothly and we obtain a complete solution.

Figure 5.18 contains a trace of all calls leading to the first successful placement of four queens on a 4×4 chessboard.

FIGURE 5.18 Trace of calls to `putQueen()` to place four queens.

```
putQueen(0)
  col = 0;
  putQueen(1)
    col = 0;
    col = 1;
    col = 2;
    putQueen(2)
      col = 0;
      col = 1;
      col = 2;
      col = 3;
    col = 3;
    putQueen(2)
      col = 0;
      col = 1;
      putQueen(3)
        col = 0;
        col = 1;
        col = 2;
        col = 3;
```

FIGURE 5.18 (*continued*)

```
      col = 2;
      col = 3;
  col = 1;
  putQueen(1)
    col = 0;
    col = 1;
    col = 2;
    col = 3;
    putQueen(2)
      col = 0;
      putQueen(3)
        col = 0;
        col = 1;
        col = 2;
        success
```

5.10 Concluding Remarks

After looking at all these examples (and one more to follow), what can be said about recursion as a programming tool? Like any other topic in data structures, it should be used with good judgment. There are no general rules for when to use it and when not to use it. Each particular problem decides. Recursion is usually less efficient than its iterative equivalent. But if a recursive program takes 100 milliseconds (ms) for execution, for example, and the iterative version only 10 ms, then although the latter is 10 times faster, the difference is hardly perceivable. If there is an advantage in the clarity, readability, and simplicity of the code, the difference in the execution time between these two versions can be disregarded. Recursion is often simpler than the iterative solution and more consistent with the logic of the original algorithm. The factorial and power functions are such examples, and we will see more interesting cases in chapters to follow.

Although every recursive procedure can be converted into an iterative version, the conversion is not always a trivial task. In particular, it may involve explicitly manipulating a stack. That is where the time–space trade-off comes into play: using iteration often necessitates the introduction of a new data structure to implement a stack, whereas recursion relieves the programmer of this task by handing it over to the system. One way or the other, if nontail recursion is involved, very often a stack has to be maintained by the programmer or by the system. But the programmer decides who carries the load.

Two situations can be presented in which a nonrecursive implementation is preferable even if recursion is a more natural solution. First, iteration should be used in the so-called real-time systems where an immediate response is vital for proper

functioning of the program. For example, in military environments, in the space shuttle, or in certain types of scientific experiments, it may matter whether the response time is 10 ms or 100 ms. Second, the programmer is encouraged to avoid recursion in programs that are executed hundreds of times. The best example of this kind of program is a compiler.

But these remarks should not be treated too stringently, because sometimes a recursive version is faster than a nonrecursive implementation. Hardware may have built-in stack operations that considerably speed up functions operating on the run-time stack, such as recursive functions. Running a simple routine implemented recursively and iteratively and comparing the two run times can help to decide if recursion is advisable—in fact, recursion can execute faster than iteration. Such a test is especially important if tail recursion comes into play. However, when a stack cannot be eliminated from the iterative version, the use of recursion is usually recommended, because the execution time for both versions does not differ substantially—certainly not by a factor of 10.

Recursion should be eliminated if some part of the work is unnecessarily repeated to compute the answer. The Fibonacci series computation is a good example of such a situation. It shows that the ease of using recursion can sometimes be deceptive, and this is where iteration can grapple effectively with run-time limitations and inefficiencies. Whether a recursive implementation leads to unnecessary repetitions may not be immediately apparent; therefore, drawing a tree of calls similar to Figure 5.8 can be very helpful. This tree shows that `Fib(n)` is called many times with the same argument `n`. A tree drawn for power or factorial functions is reduced to a linked list with no repetitions in it. If such a tree is very deep (that is, it has many levels), then the program can endanger the run-time stack with an overflow. If the tree is shallow and bushy, with many nodes on the same level, then recursion seems to be a good approach—but only if the number of repetitions is very moderate.

5.11 Case Study: A Recursive Descent Interpreter

All programs written in any programming language have to be translated into a representation that the computer system can execute. However, this is not a simple process. Depending on the system and programming language, the process may consist of translating one executable statement at a time and immediately executing it, which is called *interpretation,* or translating the entire program first and then executing it, which is called *compilation.* Whichever strategy is used, the program should not contain sentences or formulas that violate the formal specification of the programming language in which the program is written. For example, if we want to assign a value to a variable, we must put the variable first, then the equal sign, and then a value after it.

Writing an interpreter is by no means a trivial task. As an example, this case study is a sample interpreter for a limited programming language. Our language consists only of assignment statements; it contains no declarations, `if-else` statements, loops, functions, or the like. For this limited language, we would like to write a program that accepts any input and

- determines if it contains valid assignment statements (this process is known as parsing); and simultaneously,
- evaluates all expressions.

Our program is an interpreter in that it not only checks whether the assignment statements are syntactically correct, but also executes the assignments.

The program is to work in the following way. If we enter the assignment statements

```
var1 = 5;
var2 = var1;
var3 = 44/2.0 * (var2 + var1);
```

then the system can be prompted for the value of each variable separately. For instance, after entering

```
print var3
```

the system should respond by printing

```
var3 = 220
```

Evaluation of all variables stored so far may be requested by entering

```
status
```

and the following values should be printed in our example:

```
var3 = 220
var2 = 5
var1 = 5
```

All current values are to be stored on `idList` and updated if necessary. Thus, if

```
var2 = var2 * 5;
```

is entered, then

```
print var2
```

should return

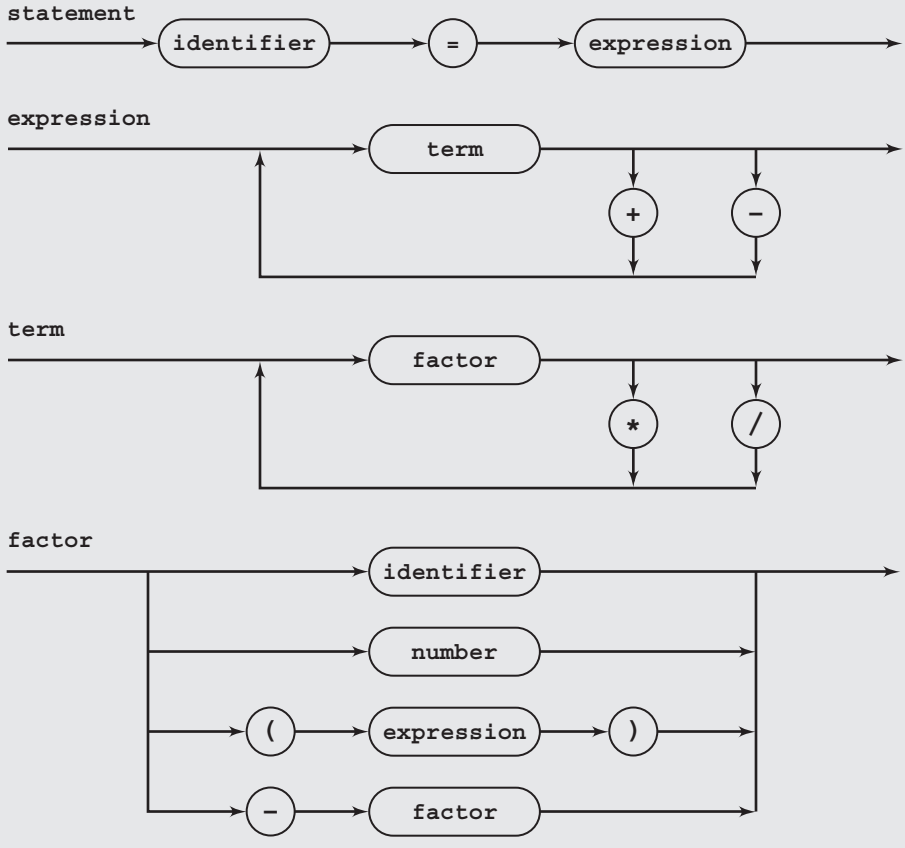
```
var2 = 25
```

The interpreter prints a message if any undefined identifier is used and if statements and expressions do not conform to common grammatical rules such as unmatched parentheses, two identifiers in a row, and so on.

The program can be written in a variety of ways, but to illustrate recursion, we chose a method known as *recursive descent*. This consists of several mutually recursive functions according to the diagrams in Figure 5.19.

These diagrams serve to define a statement and its parts. For example, a term is a factor or a factor followed by either the multiplication symbol "*" or the division symbol "/" and then another factor. A factor, in turn, is either an identifier, a number, an expression enclosed in a pair of matching parentheses, or a negated factor. In this

FIGURE 5.19 Diagrams of functions used by the recursive descent interpreter.



method, a statement is looked at in more and more detail. It is broken down into its components, and if the components are compound, they are separated into their constituent parts until the simplest language elements are found: numbers, variable names, operators, and parentheses. Thus, the program recursively descends from a global overview of the statement to more detailed elements.

The diagrams in Figure 5.19 indicate that recursive descent is a combination of direct and indirect recursion. For example, a factor can be a factor preceded by a minus, an expression can be a term, a term can be a factor, a factor can be an expression that, in turn, can be a term, until the level of identifiers or numbers is found. Thus, an expression can be composed of expressions, a term of terms, and a factor of factors.

How can the recursive descent interpreter be implemented? The simplest approach is to treat every word in the diagrams as a function name. For instance, `term()` is a function returning a double number. This function always calls `factor()` first, and if the nonblank character currently being looked at is either "*" or "/", then `term()` calls `factor()` again. Each time, the value already accumulated by `term()` is either multiplied or divided by the value returned by the subsequent call of `term()` to `factor()`. Every call of `term()` can invoke another call to `term()` indirectly through the chain `term() -> factor() -> expression() -> term()`. Pseudocode for the function `term()` looks like the following:

```
term()
   f1 = factor();
   check current character ch;
   while ch is either / or *
      f2 = factor();
      f1 = f1 * f2 or f1 / f2;
   return f1;
```

The function `expression()` has exactly the same structure, and the pseudocode for `factor()` is:

```
factor()
   process all +s and – s preceding a factor;
   if current character ch is a letter
      store in id all consecutive letters and/or digits starting with ch;
      return value assigned to id;
   else if ch is a digit
      store in id all consecutive digits starting from ch;
      return number represented by string id;
   else if ch is (
      e = expression();
      if ch is )
         return e;
```

We have tacitly assumed that `ch` is a global variable that is used for scanning an input character by character.

However, in the pseudocode, we assumed that only valid statements are entered for evaluation. What happens if a mistake is made, such as entering two equal signs, mistyping a variable name, or forgetting an operator? In the interpreter, parsing is simply discontinued after printing an error message. Figure 5.20 contains the complete code for our interpreter.

FIGURE 5.20 Implementation of a simple language interpreter.

```
//*************************  interpreter.h   ***********************

#ifndef INTERPRETER
#define INTERPRETER

#include <iostream>
#include <list>
#include <algorithm> // find()

using namespace std;

class IdNode {
public:
    IdNode(char *s = "", double e = 0) {
        id = strdup(s);
        value = e;
    }
    bool operator== (const IdNode& node) const {
        return strcmp(id,node.id) == 0;
    }
private:
    char *id;
    double value;
    friend class Statement;
    friend ostream& operator<< (ostream&, const IdNode&);
};

class Statement {
public:
    Statement() {
    }
    void getStatement();
private:
    list<IdNode> idList;
    char  ch;
    double factor();
    double term();
    double expression();
    void readId(char*);
    void issueError(char *s) {
```

FIGURE 5.20 (*continued*)

```
        cerr << s << endl; exit(1);
    }
    double findValue(char*);
    void  processNode(char*, double);
    friend ostream& operator<< (ostream&, const Statement&);
};

#endif

//*************************  interpreter.cpp   ***********************

#include <cctype>
#include "interpreter.h"

double Statement::findValue(char *id) {
    IdNode tmp(id);
    list<IdNode>::iterator i = find(idList.begin(),idList.end(),tmp);
    if (i != idList.end())
         return i->value;
    else issueError("Unknown variable");
    return 0;  // this statement will never be reached;
}

void Statement::processNode(char* id,double e) {
    IdNode tmp(id,e);
    list<IdNode>::iterator i = find(idList.begin(),idList.end(),tmp);
    if (i != idList.end())
         i->value = e;
    else idList.push_front(tmp);
}

// readId() reads strings of letters and digits that start with
// a letter, and stores them in array passed to it as an actual
// parameter.
// Examples of identifiers are: var1, x, pqr123xyz, aName, etc.

void Statement::readId(char *id) {
    int i = 0;
    if (isspace(ch))
         cin >> ch;       // skip blanks;
    if (isalpha(ch)) {
         while (isalnum(ch)) {
```

Continues

FIGURE 5.20 (*continued*)

```
            id[i++] = ch;
            cin.get(ch); // don't skip blanks;
        }
        id[i] = '\0';
    }
    else issueError("Identifier expected");
}

double Statement::factor() {
    double var, minus = 1.0;
    static char id[200];
    cin >> ch;
    while (ch == '+' || ch == '-') {      // take all '+'s and '-'s.
        if (ch == '-')
            minus *= -1.0;
        cin >> ch;
    }
    if (isdigit(ch) || ch == '.') {       // Factor can be a number
         cin.putback(ch);
         cin >> var >> ch;
    }
    else if (ch == '(') {                  // or a parenthesized
                                           //  expression,
         var = expression();
         if (ch == ')')
              cin >> ch;
         else issueError("Right paren left out");
    }
    else {
         readId(id);                       // or an identifier.
         if (isspace(ch))
             cin >> ch;
         var = findValue(id);
    }
    return minus * var;
}

double Statement::term() {
    double f = factor();
    while (true) {
        switch (ch) {
```

FIGURE 5.20 (*continued*)

```
            case '*' : f *= factor(); break;
            case '/' : f /= factor(); break;
            default  : return f;
        }
    }
}

double Statement::expression() {
    double t = term();
    while (true) {
        switch (ch) {
            case '+' : t += term(); break;
            case '-' : t -= term(); break;
            default  : return t;
        }
    }
}

void Statement::getStatement() {
    char id[20], command[20];
    double e;
    cout << "Enter a statement: ";
    cin  >> ch;
    readId(id);
    strupr(strcpy(command,id));
    if (strcmp(command,"STATUS") == 0)
         cout << *this;
    else if (strcmp(command,"PRINT") == 0) {
         readId(id);
         cout << id << " = " << findValue(id) << endl;
    }
    else if (strcmp(command,"END") == 0)
         exit(0);
    else {
         if (isspace(ch))
             cin >> ch;
         if (ch == '=') {
              e = expression();
              if (ch != ';')
                   issueError("There are some extras in the statement");
              else processNode(id,e);
         }
```

Continues

FIGURE 5.20 (*continued*)

```
          else issueError("'=' is missing");
    }
}

ostream& operator<< (ostream& out, const Statement& s) {
    list<IdNode>::iterator i = s.idList.begin();
    for ( ; i != s.idList.end(); i++)
        out << *i;
    out << endl;
    return out;
}

ostream& operator<< (ostream& out, const IdNode& r) {
    out << r.id << " = " << r.value << endl;
    return out;
}

//************************  useInterpreter.cpp   ************************

#include "interpreter.h"

using namespace std;

int main() {
    Statement statement;
    cout << "The program processes statements of the following format:\n"
         << "\t<id> = <expr>;\n\tprint <id>\n\tstatus\n\tend\n\n";
    while (true)               // This infinite loop is broken by exit(1)
        statement.getStatement(); // in getStatement() or upon finding an
                                  // error.
    return 0;
}
```

5.12 EXERCISES

1. The set of natural numbers **N** defined at the beginning of this chapter includes the numbers 10, 11, . . . , 20, 21, . . . , and also the numbers 00, 000, 01, 001, ... Modify this definition to allow only numbers with no leading zeros.
2. Write a recursive function that calculates and returns the length of a linked list.
3. What is the output for the following version of `reverse()`:

```
void reverse() {
  int ch;
  cin.get(ch);
  if (ch != '\n')
      reverse();
  cout.put(ch);
}
```

4. What is the output of the same function if `ch` is declared as

```
static char ch;
```

5. Write a recursive method that for a positive integer n prints odd numbers
   a. between 1 and n
   b. between n and 1
6. Write a recursive method that for a positive integer returns a string with commas in the appropriate places, for example, `putCommas(1234567)` returns the string "1,234,567."
7. Write a recursive method to print a *Syracuse sequence* that begins with a number n_0 and each element n_i of the sequence is $n_{i-1}/2$ if n_{i-1} is even and $3n_{i-1} + 1$ otherwise. The sequence ends with 1.
8. Write a recursive method that uses only addition, subtraction, and comparison to multiply two numbers.
9. Write a recursive function to compute the binomial coefficient according to the definition

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{otherwise} \end{cases}$$

10. Write a recursive function to add the first n terms of the series

$$1 + \frac{1}{2} - \frac{1}{3} + \frac{1}{4} - \frac{1}{5} \cdots$$

11. Write a recursive function `GCD(n,m)` that returns the greatest common divisor of two integers `n` and `m` according to the following definition:

$$\text{GCD}(n, m) = \begin{cases} m & \text{if } m \le n \text{ and } n \bmod m = 0 \\ \text{GCD}(m, n) & \text{if } n < m \\ \text{GCD}(m, n \bmod m) & \text{otherwise} \end{cases}$$

12. Give a recursive version of the following function:

```
void cubes(int n) {
  for (int i = 1; i <= n; i++)
       cout << i * i * i << '';
}
```

13. An early application of recursion can be found in the seventeenth century in John Napier's method of finding logarithms. The method was as follows:

start with two numbers `n, m` *and their logarithms* `logn, logm` *if they are known;*
`while` *not done*
for a geometric mean of two earlier numbers find a logarithm which is an arithmetic mean of two earlier logarithms, that is, `logk = (logn+logm)/2` *for* `k =` $\sqrt{\texttt{nm}}$`;`
proceed recursively for pairs `(n,` $\sqrt{\texttt{nm}}$`)` *and* `(`$\sqrt{\texttt{nm}}$`,m);`

For example, the 10-based logarithms of 100 and 1,000 are numbers 2 and 3, the geometric mean of 100 and 1,000 is 316.23, and the arithmetic mean of their logarithms, 2 and 3, is 2.5. thus, the logarithm of 316.23 equals 2.5. The process can be continued: the geometric mean of 100 and 316.23 is 177.83, whose logarithm is equal to (2 + 2.5)/2 = 2.25.

a. Write a recursive function `logarithm()` that outputs logarithms until the difference between adjacent logarithms is smaller than a certain small number.

b. Modify this function so that a new function `logarithmOf()` finds a logarithm of a specific number x between 100 and 1,000. Stop processing if you reach a number y such that $y - x < \epsilon$ for some ϵ.

c. Add a function that calls `logarithmOf()` after determining between what powers of 10 a number x falls so that it does not have to be a number between 100 and 1,000.

14. The algorithms for both versions of the power function given in this chapter are rather simpleminded. Is it really necessary to make eight multiplications to compute x^8? It can be observed that $x^8 = (x^4)^2$, $x^4 = (x^2)^2$, and $x^2 = x \cdot x$; that is, only three multiplications are needed to find the value of x^8. Using this observation, improve both algorithms for computing x^n. Hint: A special case is needed for odd exponents.

15. Execute by hand the functions `tail()` and `nonTail()` for the parameter values of 0, 2, and 4. Definitions of these functions are given in Section 5.4.

16. Check recursively if the following objects are palindromes:

a. a word

b. a sentence (ignoring blanks, lower- and uppercase differences, and punctuation marks so that "Madam, I'm Adam" is accepted as a palindrome)

17. For a given character recursively, without using `strchr()` or `strrchr()`,

a. Check if it is in a string.

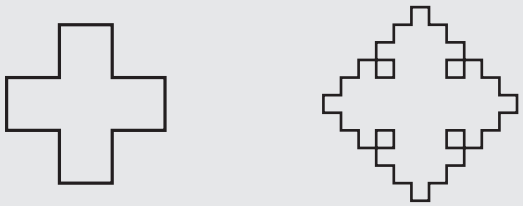
b. Count all of its occurrences in a string.

c. Remove all of its occurrences from a string.

18. Write equivalents of the last three functions for substrings (do not use `strstr()`).

19. What changes have to be made in the program in Figure 5.6 to draw a line as in Figure 5.21? Try it and experiment with other possibilities to generate other curves.

FIGURE 5.21 Lines to be drawn with modified program in Figure 5.6.



20. Create a tree of calls for $\sin(x)$ assuming that only $\frac{x}{18}$ (and smaller values) do not trigger other calls.

21. Write recursive and nonrecursive functions to print out a nonnegative integer in binary. The functions should not use bitwise operations.

22. The nonrecursive version of the function for computing Fibonacci numbers uses information accumulated during computation, whereas the recursive version does not. However, it does not mean that no recursive implementation can be given that can collect the same information as the nonrecursive counterpart. In fact, such an implementation can be obtained directly from the nonrecursive version. What would it be? Consider using two functions instead of one; one would do all the work, and the other would only invoke it with the proper parameters.

23. The function `putQueen()` does not recognize that certain configurations are symmetric. Adapt function `putQueen()` for a full 8×8 chessboard, write the function `printBoard()`, and run a program for solving the eight queens problem so that it does not print symmetric solutions.

24. Finish the trace of execution of `putQueen()` shown in Figure 5.18.

25. Execute the following program by hand from the case study, using these two entries:

a. `v = x + y*w - z`

b. `v = x * (y - w) --z`

Indicate clearly which functions are called at which stage of parsing these sentences.

26. Extend our interpreter so that it can also process exponentiation, ^. Remember that exponentiation has precedence over all other operations so that `2 - 3^4 * 5` is the same as `2 - ((3^4) * 5)`. Notice also that exponentiation is a right-associative operator (unlike addition or multiplication); that is, `2^3^4` is the same as `2^(3^4)` and not `(2^3)^4`.

27. In C++, the division operator, /, returns an integer result when it is applied to two integers; for instance, `11/5` equals `2`. However, in our interpreter, the result is `2.2`. Modify this interpreter so that division works the same way as in C++.

28. Our interpreter is unforgiving when a mistake is made by the user, because it finishes execution if a problem is detected. For example, when the name of a variable is mistyped when requesting its value, the program notifies the user and exits and destroys the list of identifiers. Modify the program so that it continues execution after finding an error.

29. Write the shortest program you can that uses recursion.

5.13 PROGRAMMING ASSIGNMENTS

1. Compute the standard deviation σ for n values x_k stored in an array `data` and for the equal probabilities $\frac{1}{n}$ associated with them. The standard deviation is defined as

$$\sigma = \sqrt{V}$$

where the variance, V, is defined by

$$V = \frac{1}{n-1}\Sigma_k(x_k - \bar{x})^2$$

and the mean, $\bar{x}$, by

$$\bar{x} = \frac{1}{n}\Sigma_k x_k$$

Write recursive and iterative versions of both V and $\bar{x}$ and compute the standard deviation using both versions of the mean and variance. Run your program for n = 500, 1,000, 1,500, and 2,000 and compare the run times.

2. Write a program to do symbolic differentiation. Use the following formulas:

$$\text{Rule 1: } (fg)' = fg' + f'g$$

$$\text{Rule 2: } (f+g)' = f' + g'$$

$$\text{Rule 3: } \left(\frac{f}{g}\right)' = \frac{f'g - fg'}{g^2}$$

$$\text{Rule 4: } (ax^n)' = nax^{n-1}$$

An example of application of these rules is given below for differentiation with respect to *x:*

$$\left(5x^3 + \frac{6x}{y} - 10x^2y + 100\right)'$$

$$= (5x^3)' + \left(\frac{6x}{y}\right)' + (-10x^2y)' + (100)' \qquad \text{by Rule 2}$$

$$= 15x^2 + \left(\frac{6x}{y}\right)' + (-10x^2y)' \qquad \text{by Rule 4}$$

$$= 15x^2 + \frac{(6x)'y - (6x)y'}{y^2} + (-10x^2y)' \qquad \text{by Rule 3}$$

$$= 15x^2 + \frac{6y}{y^2} + (-10x^2y)' \qquad \text{by Rule 4}$$

$$= 15x^2 + \frac{6y}{y^2} + (-10x^2)y' + (-10x^2)'y \qquad \text{by Rule 1}$$

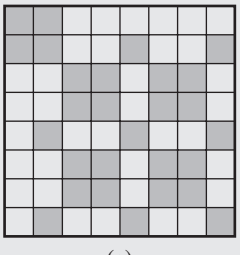
$$= 15x^2 + \frac{6}{y} - 20xy \qquad \text{by Rule 4}$$

First, run your program for polynomials only, and then add formulas for derivatives for trigonometric functions, logarithms, and so on, that extend the range of functions handled by your program.

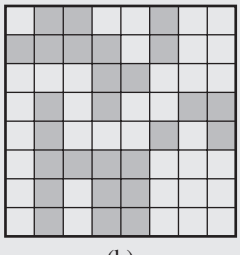
3. An $n \times n$ square consists of black and white cells arranged in a certain way. The problem is to determine the number of white areas and the number of white cells in each area. For example, a regular 8×8 chessboard has 32 one-cell white areas; the square in Figure 5.22a consists of 10 areas, 2 of them of 10 cells, and 8 of 2 cells; the square in Figure 5.22b has 5 white areas of 1, 3, 21, 10, and 2 cells.

Write a program that, for a given $n \times n$ square, outputs the number of white areas and their sizes. Use an $(n + 2) \times (n + 2)$ array with properly marked cells. Two additional rows and columns constitute a frame of black cells surrounding the entered square to simplify your implementation. For instance, the square in Figure 5.22b is stored as the square in Figure 5.22c.

FIGURE 5.22 (a–b) Two $n \times n$ squares of black and white cells and (c) an $(n + 2) \times (n + 2)$ array implementing square (b).



(a)



(b)

```
b b b b b b b b b b
b w b b w w b w w b
b b b b b w b w w b
b w w w b b w w w b
b w b w b w w b b b
b w b w w w b w b b
b w b b b b w w w b
b w b w b b w w w b
b w b w b b w w w b
b b b b b b b b b b
```

(c)

Traverse the square row by row and, for the first unvisited cell encountered, invoke a function that processes one area. The secret is in using four recursive calls in this function for each unvisited white cell and marking it with a special symbol as visited (counted).

4. Write a program for *pretty printing* C++ programs; that is, for printing programs with consistent use of indentation, the number of spaces between tokens such as key words, parentheses, brackets, operators, the number of blank lines between blocks of code (classes, functions), aligning braces with key words, aligning `else` statements with the corresponding `if` statements, and so on. The program takes as input a C++ file and prints code in this file according to the rules incorporated in the pretty printing program. For example, the code

```
if (n == 1) { n = 2 * m;
if (m < 10)
f(n,m-1); else f(n,m-2); } else n = 3 * m;
```

should be transformed into

```
if (n == 1) {
     n = 2 * m;
     if (m < 10)
          f(n,m-1);
     else f(n,m-2);
}
else n = 3 * m;
```

5. An excellent example of a program that can be greatly simplified by the use of recursion is the Chapter 4 case study, escaping a maze. As already explained, in each maze cell the mouse stores on the maze stack up to four cells neighboring the cell in which it is currently located. The cells put on the stack are the ones that should be investigated after reaching a dead end. It does the same for each visited cell. Write a program that uses recursion to solve the maze problem. Use the following pseudocode:

```
exitCell(currentCell)
    if currentCell is the exit
         success;
    else exitCell(the passage above currentCell);
         exitCell(the passage below currentCell);
         exitCell(the passage left to currentCell);
         exitCell(the passage right to currentCell);
```

6 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 so 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* (*terminal nodes*) 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 disjointed 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 6.2 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.

FIGURE 6.1 Examples of trees.

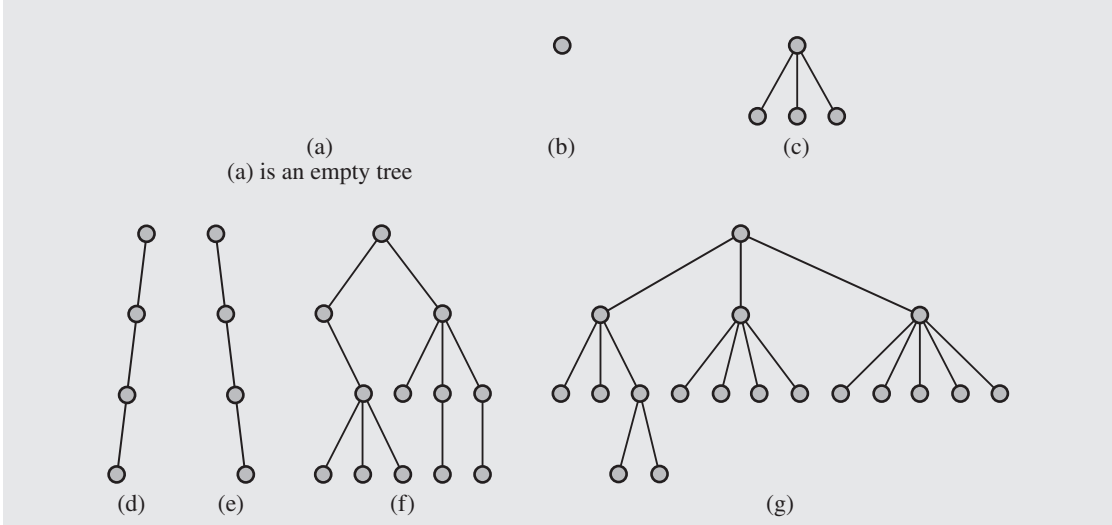
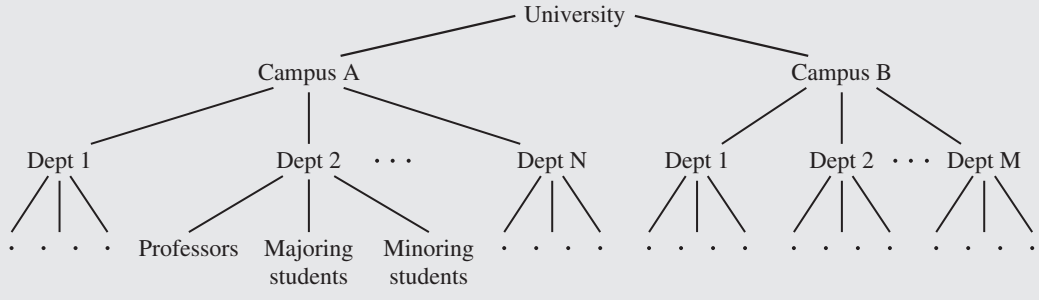


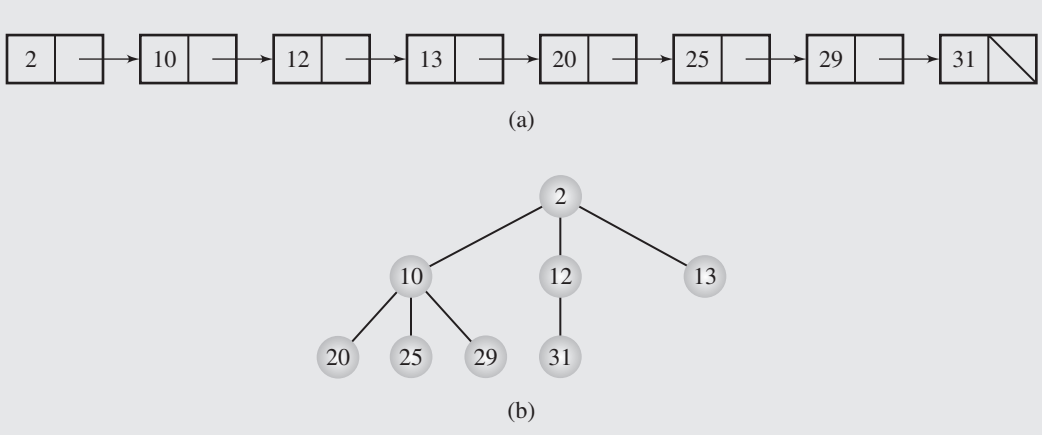
FIGURE 6.2 Hierarchical structure of a university shown as a tree.



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. 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 9,999 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 farthest away. For example, the linked list in Figure 6.3a can be transformed into the tree in Figure 6.3b.

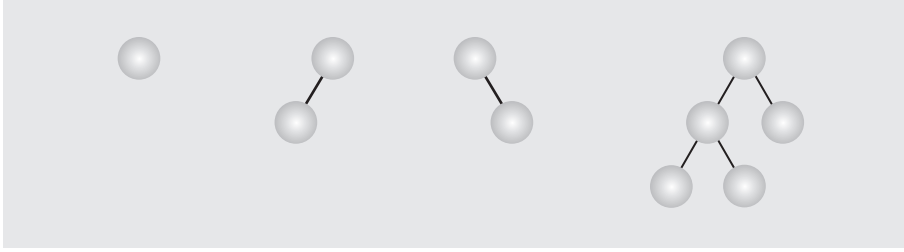
FIGURE 6.3 Transforming (a) a linked list into (b) a tree.



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 a right child. For example, the trees in Figure 6.4 are binary trees,

FIGURE 6.4 Examples of 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.

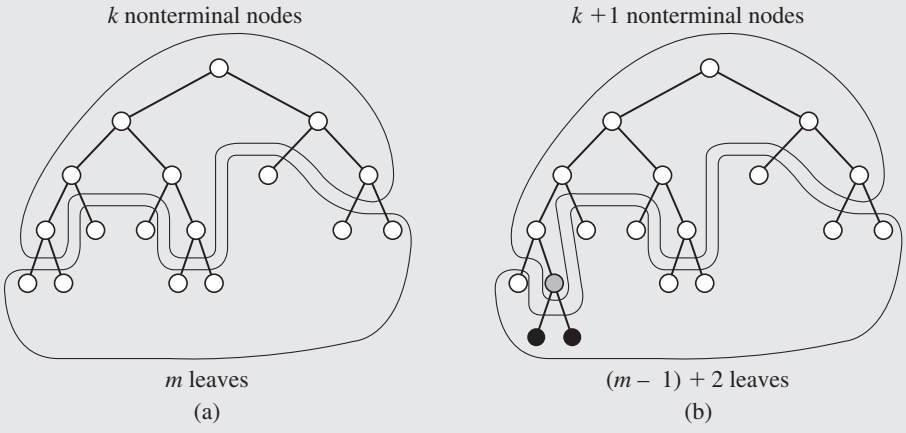
As already defined, the level of a node is the number of arcs traversed from the root to the node plus one. According to this definition, the root is at level 1, its nonempty children are at level 2, and so on. If all the nodes at all levels except the last had two children, then there would be $1 = 2^0$ 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 due to 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).

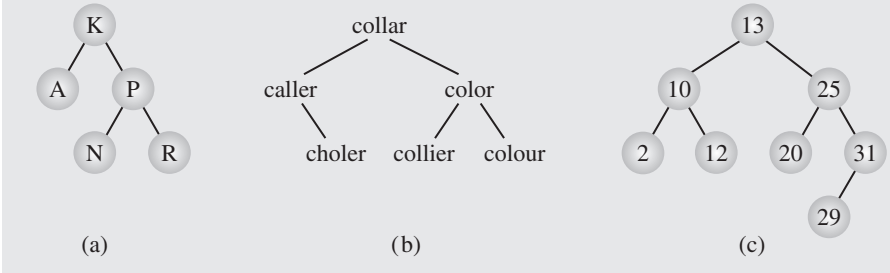
In this chapter, the *binary search trees,* also called *ordered binary trees,* are of particular interest. A binary search tree has the following property: for each node n of the tree, all values stored in its left subtree (the tree whose root is the left child) are less than value v stored in n, and all values stored in the right subtree are greater than or equal to v. For reasons to be discussed later, storing multiple copies of the same

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



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.

FIGURE 6.6 Examples of binary search trees.



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.

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

However, this implementation may be inconvenient, even if the array is flexible—that is, a vector. Locations of children must be known to insert a new node, and these locations may need to be located sequentially. After deleting a node from the tree, a hole in the array would have to be eliminated. This can be done either by using a special marker for an unused cell, which may lead to populating the array with many unused cells, or by moving elements by one position, which also requires updating references to the elements that have been moved. Sometimes an array implementation is convenient and desirable, and it will be used when discussing the heap sort. But usually, another approach needs to be used.

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 6.8 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& e, BSTNode<T> *l = 0, BSTNode<T> *r = 0) {
        el = e; left = l; right = r;
    }
    T el;
    BSTNode<T> *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->el << ' ';
    }
    . . . . . .
};

#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 element to be located with the value stored in the node currently pointed at. If the element 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 element 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, because 31 is again

FIGURE 6.9 A function for searching a binary search tree.

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

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 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 depth of a node in the tree is determined by the path length. An average depth, 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 can also 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, because $path_{best}/path_n \approx .7215$, 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 traversals 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 met.

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

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);
        }
    }
}
```

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

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 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 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

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);
    }
}
```

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.

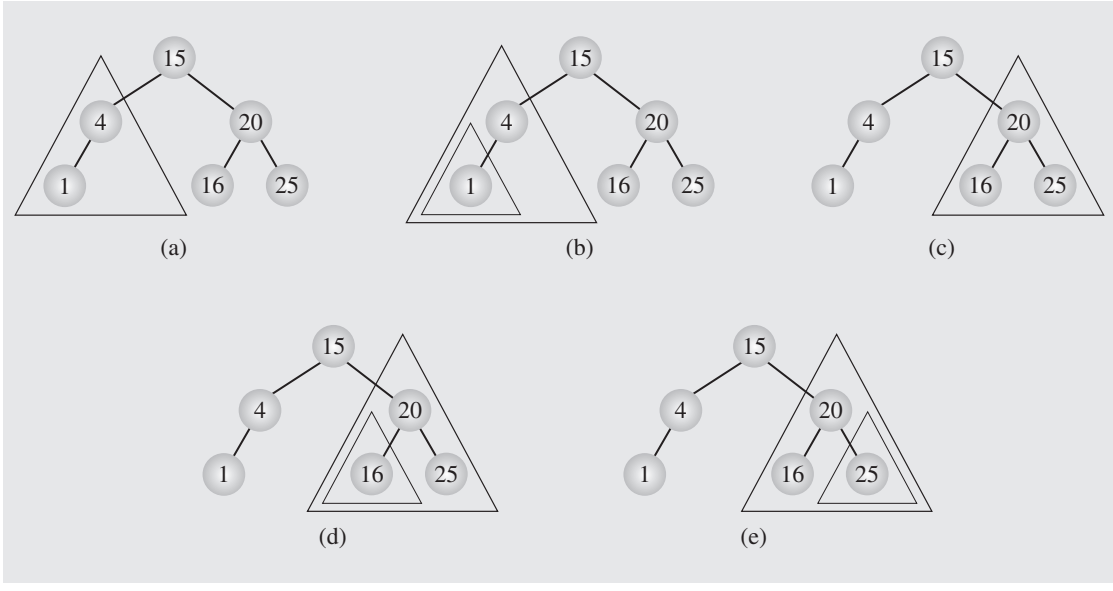
(d) `inorder()` is called for node 16, the node is visited, and then on its null left subtree, which is followed by visiting node 16. After a quick call to `inorder()` on the null right subtree of node 16 and return to the call on node 20, node 20 is also visited.

(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.

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

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

FIGURE 6.12 Inorder tree traversal.



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.

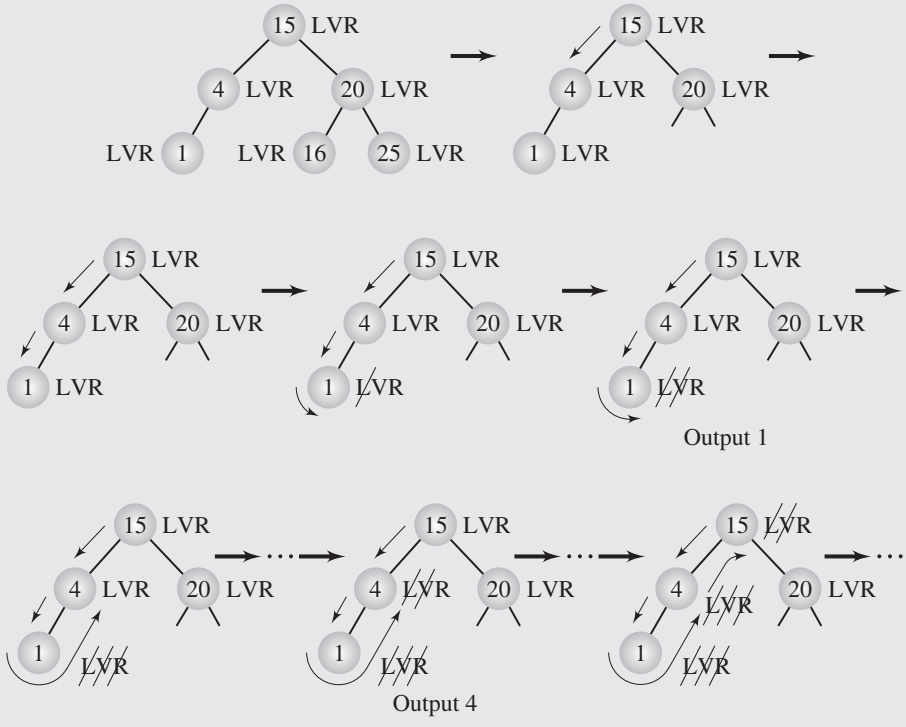
To present the way `inorder()` works, the behavior of the run-time stack is observed. The numbers in parentheses in Figure 6.14 indicate return addresses shown on the left-hand side of the code for `inorder()`.

```
          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.

(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()`).

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



(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. Because `node` is not null, `inorder()` is about to be invoked for `node`'s left child, ↑1.

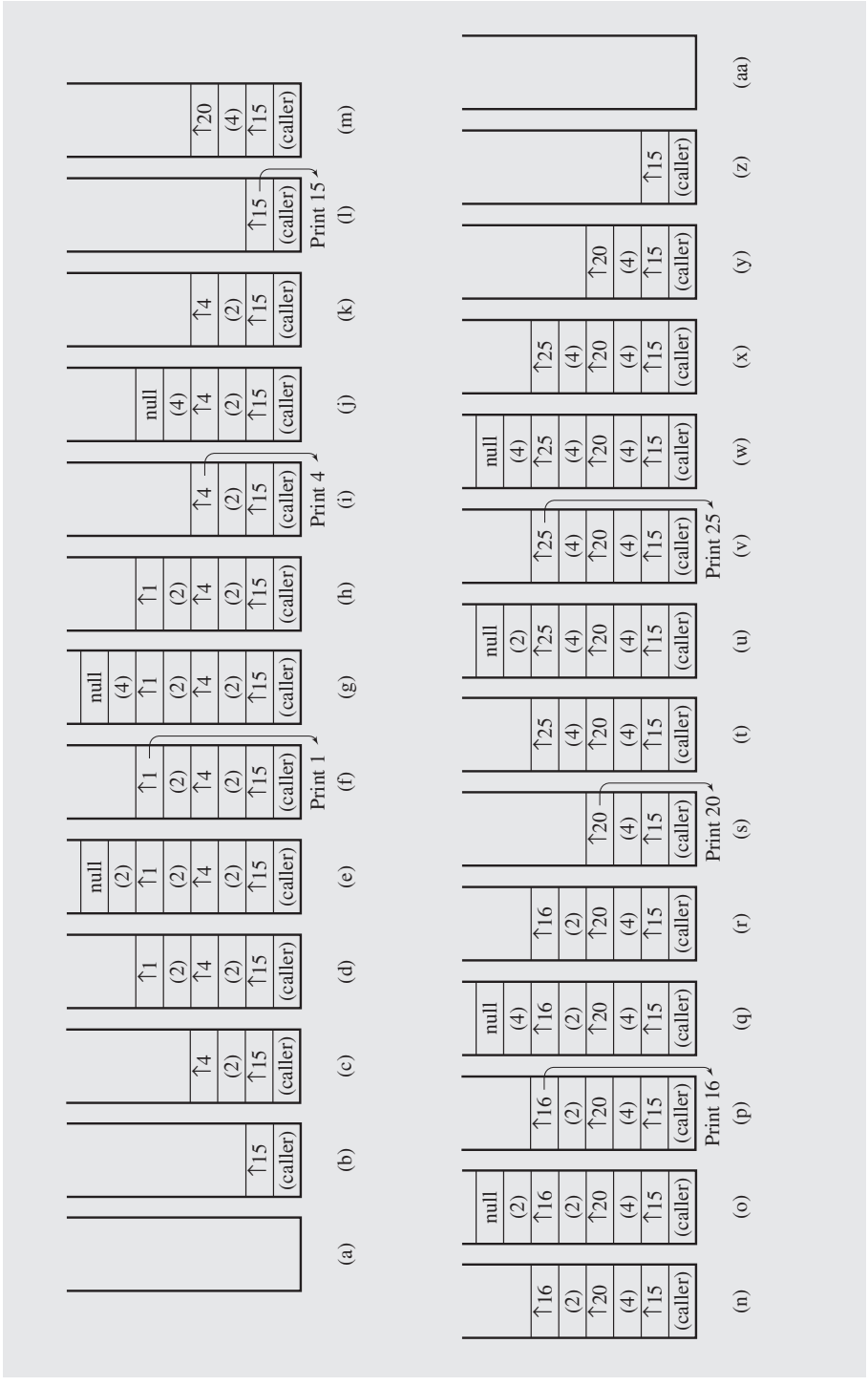
(d) First, the return address, (2), and the `node`'s value are stored on the stack.

(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. Because `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. Because `node` is not completely processed, the value of `node` and address (2) are still on 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 updated.

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



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

(i) Because 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 the `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

FIGURE 6.15 A nonrecursive 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;
    }
}
```

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 (an 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 `q` 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).

FIGURE 6.16 A nonrecursive 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->right == 0 || p->right == q) {
            visit(p);
            q = p;
            if (travStack.empty())
                 return;
            p = travStack.pop();
        }
        travStack.push(p);
        p = p->right;
    }
}
```

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.

FIGURE 6.17 A nonrecursive 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;
    }
}
```

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.

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

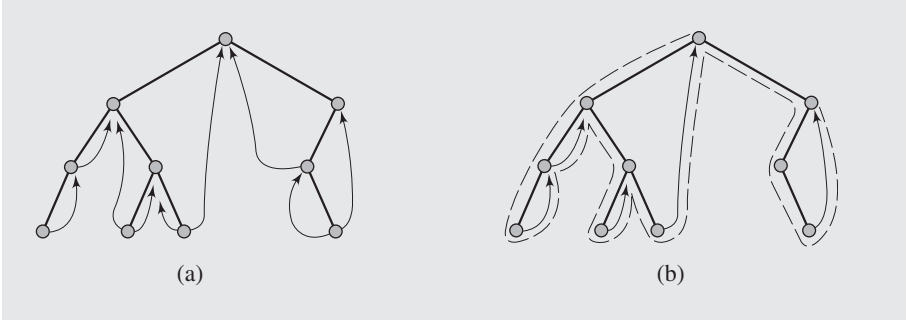


Figure 6.18a suggests that pointers to both predecessors and successors have to be maintained, which is not always the case. It may be sufficient to use only one thread, as shown in the implementation of the inorder traversal of a threaded tree, which requires only pointers to successors (Figure 6.18b).

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 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.

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& e, ThreadedNode *l = 0, ThreadedNode *r = 0) {
        el = e; left = l; right = r; successor = 0;
    }
    T el;
    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
```

Continues

FIGURE 6.19 (*continued*)

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

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 the right child of the rightmost node in its left descendant;
            go to this left descendant;
```

This algorithm successfully traverses the tree, but only once, because 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). Because 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, because 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, 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

FIGURE 6.20 Implementation of the Morris algorithm for inorder traversal.

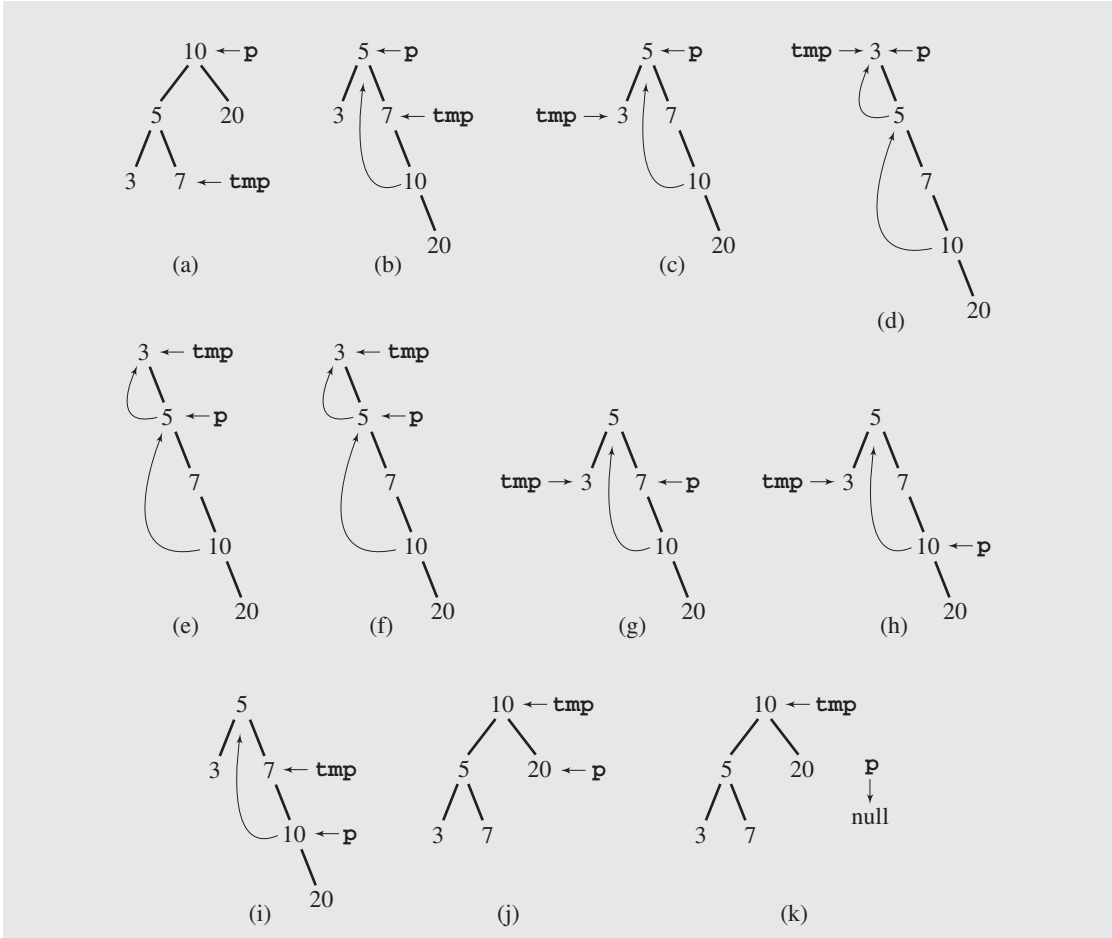
```
template<class T>
void BST<T>::MorrisInorder() {
    BSTNode<T> *p = root, *tmp;
    while (p != 0)
        if (p->left == 0) {
             visit(p);
             p = p->right;
        }
        else {
             tmp = p->left;
             while (tmp->right != 0 && // go to the rightmost node
                    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;
        }
}
```

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 (Figure 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, because it has no left child, nor has its position been altered.

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.

FIGURE 6.21 Tree traversal with the Morris method.



Preorder traversal is easily obtainable from inorder traversal by moving `visit()` from the 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 the 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 5,000 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 with key `el`, a tree node with a dead end has to be reached, and the new node has to be attached to it. Such a tree node is found using the same technique that tree searching used: the key `el` is compared to the key of a node currently being examined during a tree scan. If `el` is less than that key, the left child (if any) of `p` is tried; otherwise, the right child (if any) is tested. If the child of `p` to be tested is empty, the scanning is discontinued and the new node becomes this child. The procedure is illustrated in Figure 6.22. Figure 6.23 contains an implementation of 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 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 6.22 Inserting nodes into binary search trees.

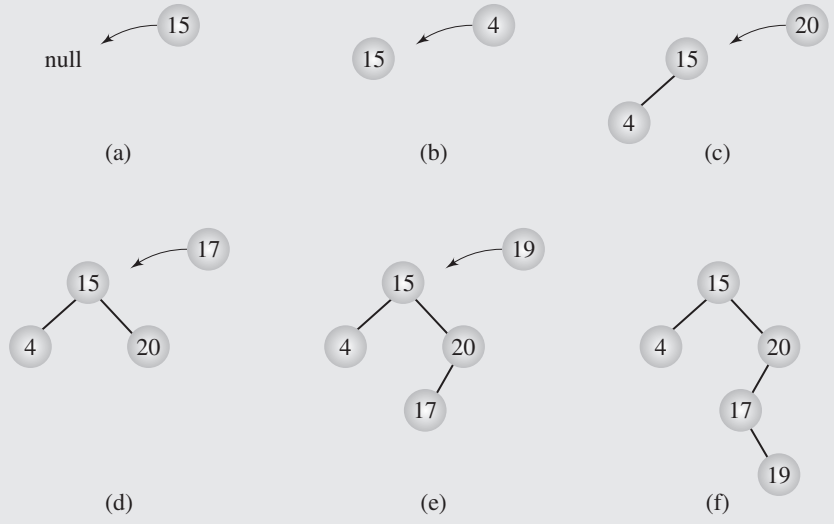


FIGURE 6.23 Implementation of the insertion algorithm.

```
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 (el < p->el)
             p = p->left;
        else p = p->right;
    }
    if (root == 0)    // tree is empty;
         root = new BSTNode<T>(el);
    else if (el < prev->el)
         prev->left   = new BSTNode<T>(el);
    else prev->right  = new BSTNode<T>(el);
}
```

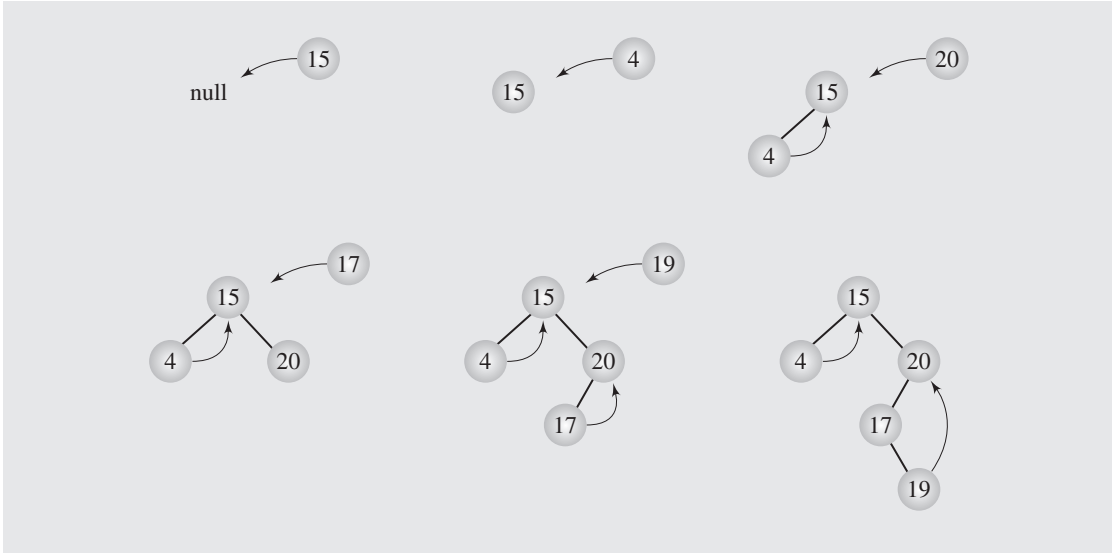
The function for inserting a node in a threaded tree is a simple extension of `insert()` for regular binary search trees to adjust threads whenever applicable. This function is for inorder tree traversal and it only takes care of successors, not predecessors.

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.

FIGURE 6.24 Implementation of the algorithm to insert a node into 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->el > 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->el > 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;
}
```

FIGURE 6.25 Inserting nodes into a threaded tree.

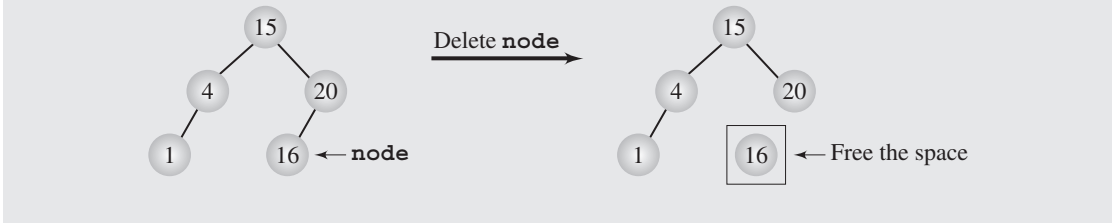


6.6 Deletion

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.

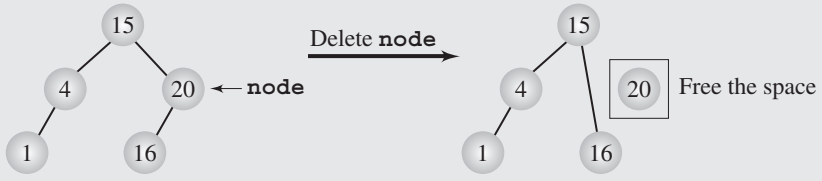
FIGURE 6.26 Deleting a leaf.



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.

FIGURE 6.27 Deleting a node with one child.



3. The node has two children. In this case, no one-step operation can be performed because the parent's right or left pointer cannot point to both the 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 key of the right subtree is greater than every key of the left subtree, so the best thing to do is to find in the left subtree the node with the greatest key and make it a parent of the right subtree. Symmetrically, the node with the lowest key 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.

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 `el`. 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 from either its parent's left or right pointer data member. Otherwise, access to the entire subtree having this node as its root would be lost. One reason for this is the fact that `search()` focuses on the

FIGURE 6.28 Summary of deleting by merging.

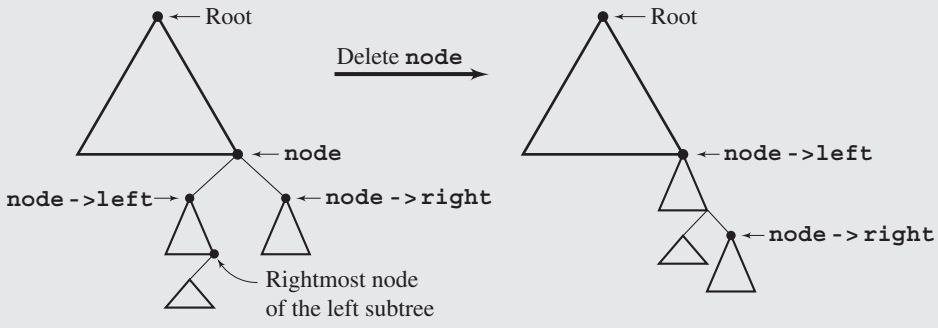


FIGURE 6.29 Implementation of an algorithm for deleting by merging.

```
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
                                    //    subtree and the right subtree;
             tmp = node;            // 4.
             node = node->left;     // 5.
        }
        delete tmp;                 // 6.
     }
}
```

Continues

FIGURE 6.29 *(continued)*

```
template<class T>
void BST<T>::findAndDeleteByMerging(const T& el) {
    BSTNode<T> *node = root, *prev = 0;
    while (node != 0) {
        if (node->el == el)
             break;
        prev = node;
        if (el < node->el)
             node = node->left;
        else node = node->right;
    }
    if (node != 0 && node->el == el)
         if (node == root)
              deleteByMerging(root);
         else if (prev->left == node)
              deleteByMerging(prev->left);
         else deleteByMerging(prev->right);
    else if (root != 0)
         cout << "element" << el << "is not in the tree\n";
    else cout << "the tree is empty\n";
}
```

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, the problem 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

FIGURE 6.30 Details of deleting by merging.

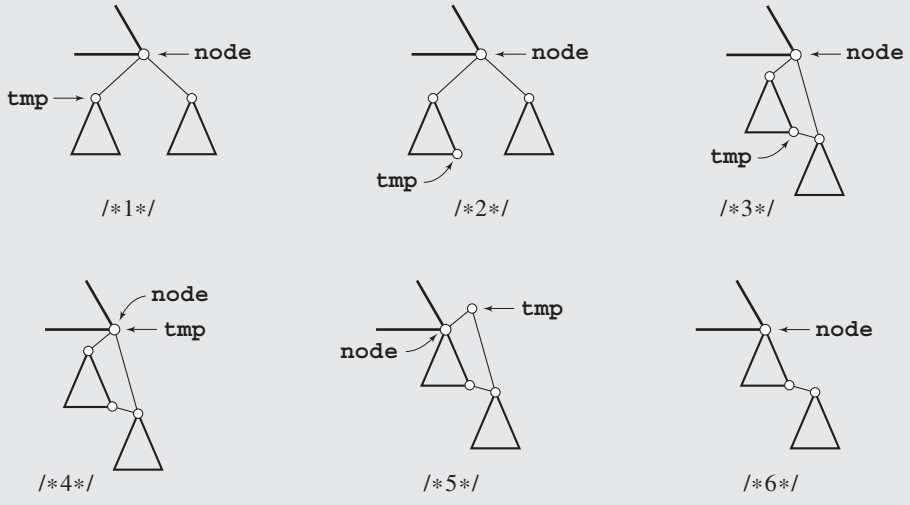
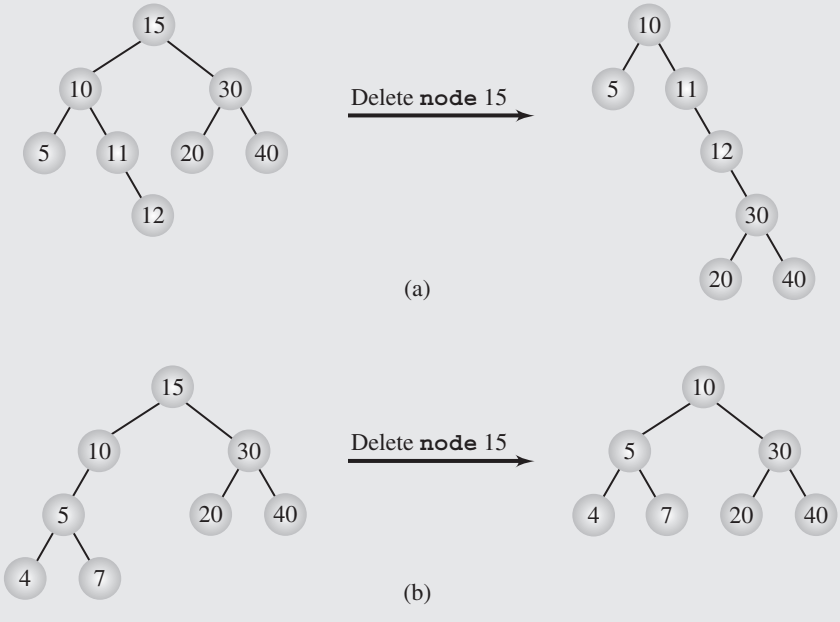


FIGURE 6.31 The height of a tree can be (a) extended or (b) reduced after deleting by merging.



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 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.

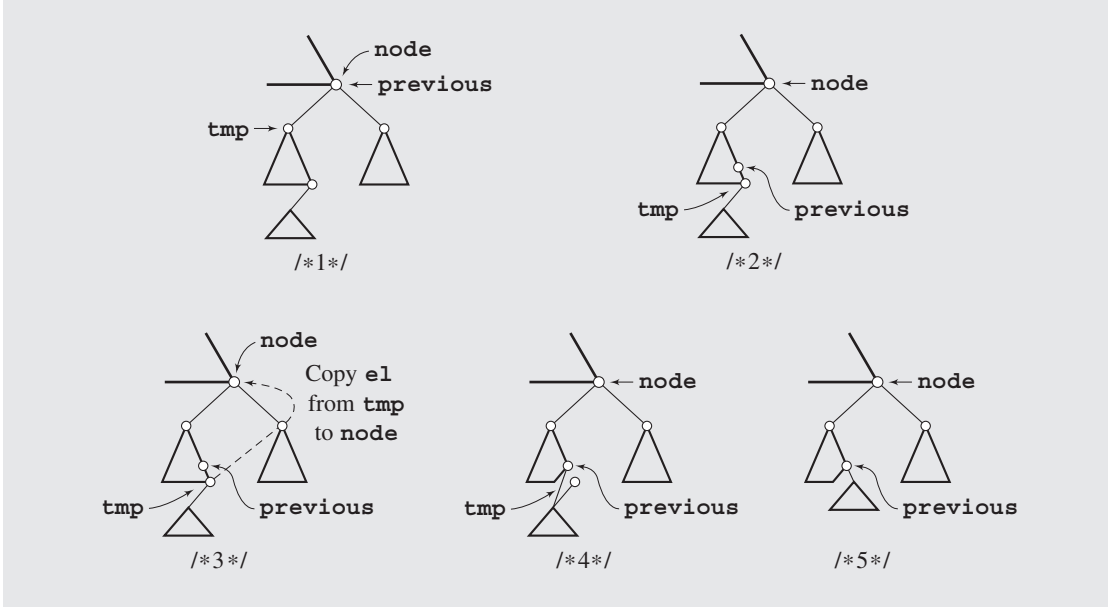
To implement the algorithm, two functions can be used. One function, `deleteByCopying()`, is illustrated in Figure 6.32. The second function, `findAndDeleteByCopying()`, is just like `findAndDeleteByMerging()`, but it calls `deleteByCopying()` instead of `deleteByMerging()`. A step-by-step trace is shown in Figure 6.33, and the numbers under the diagrams refer to the numbers indicated in comments included in the implementation of `deleteByCopying()`.

FIGURE 6.32 Implementation of an algorithm for deleting by copying.

```
template<class T>
void BST<T>::deleteByCopying(BSTNode<T>*& node) {
    BSTNode<T> *previous, *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;
          previous = node;                    // 1.
          while (tmp->right != 0) {           // 2.
              previous = tmp;
              tmp = tmp->right;
          }
          node->el = tmp->el;                 // 3.
          if (previous == node)
               previous ->left  = tmp->left;
          else previous ->right = tmp->left;  // 4.
     }
     delete tmp;                              // 5.
}
```

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 the key 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 key

FIGURE 6.33 Deleting by copying.



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 subtree 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 key 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 $\Theta(n \lg^3 n)$ 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.

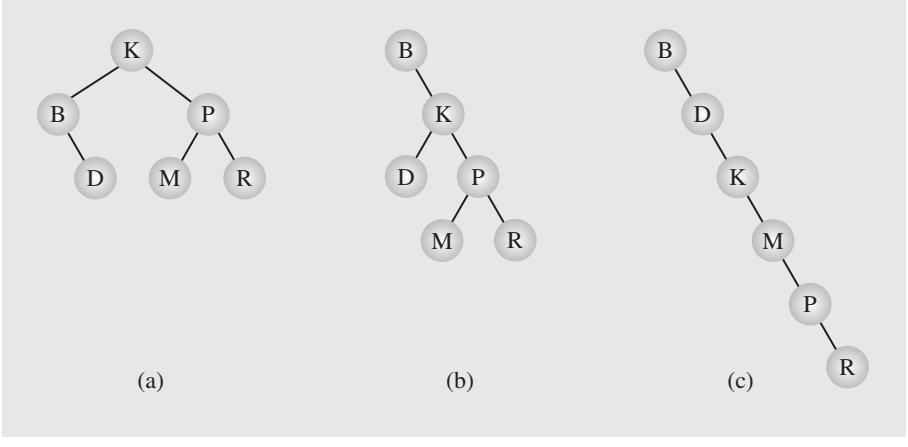
These results may be of moderate importance for practical applications. Experiments show that for a 2,048-node binary tree, only after 1.5 million insertions and asymmetric deletions does the IPL become worse than in a randomly generated tree.

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.

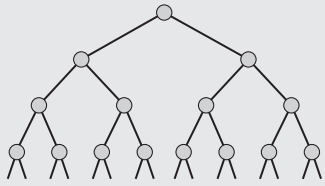
FIGURE 6.34 Different binary search trees with the same information.



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. Because 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 be checked to locate a particular element. This is a substantial difference compared

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	$2^{10} = 1{,}024$	$2{,}047 = 2^{11} - 1$
⋮		
14	$2^{13} = 8{,}192$	$16{,}383 = 2^{14} - 1$
⋮		
h	2^{h-1}	$n = 2^h - 1$
⋮		

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, because 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. After all the data arrive, 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 subarray. 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.

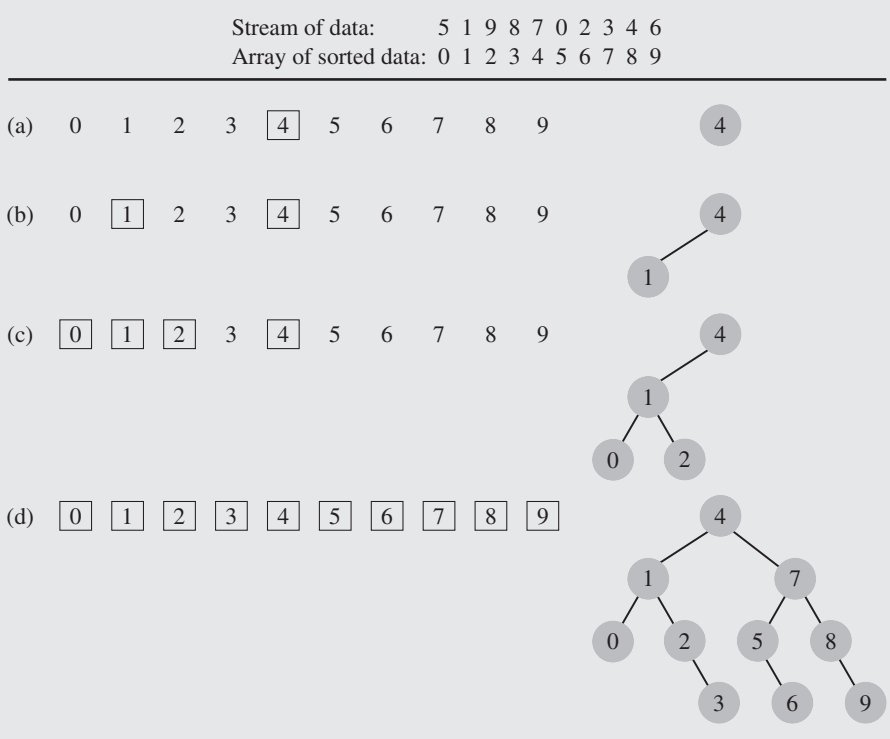
In this description, first the root is inserted into an initially empty tree, then its left child, then its right child, and so on level by level. An implementation of this algorithm is greatly simplified if the order of insertion is changed: first insert the root, then its left child, then the left 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 the 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, 3, 7, 5, 6, 8, and 9 (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 that needed to be sorted before the construction of a perfectly balanced tree began. To avoid sorting, it required deconstructing the tree after placing elements in the array using the inorder traversal 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* introduced by Adel'son-Vel'skii and Landis (1962). 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;
  right subtree of Ch becomes left subtree of Ch's parent Par;
  node Ch acquires Par as its right child;
```

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.

FIGURE 6.37 Right rotation of child `Ch` about parent `Par`.

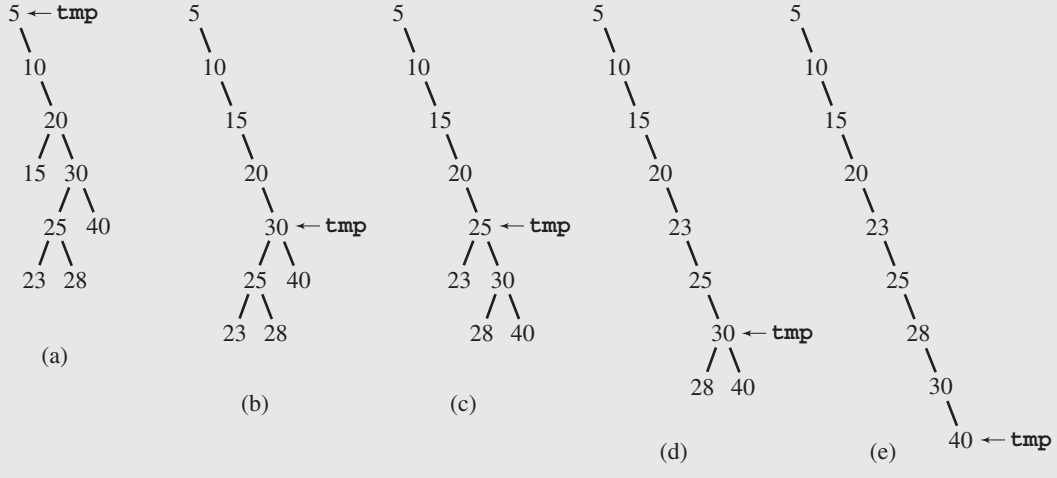
Basically, the DSW algorithm transfigures an arbitrary binary search tree into a linked listlike tree called a *backbone* or *vine.* Then this elongated tree is transformed in a series of passes into a perfectly balanced tree by repeatedly rotating every second node of the backbone about its parent.

In the first phase, a backbone is created using the following routine:

```
createBackbone(root)
  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 that 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.

FIGURE 6.38 Transforming a binary search tree into a backbone.



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.

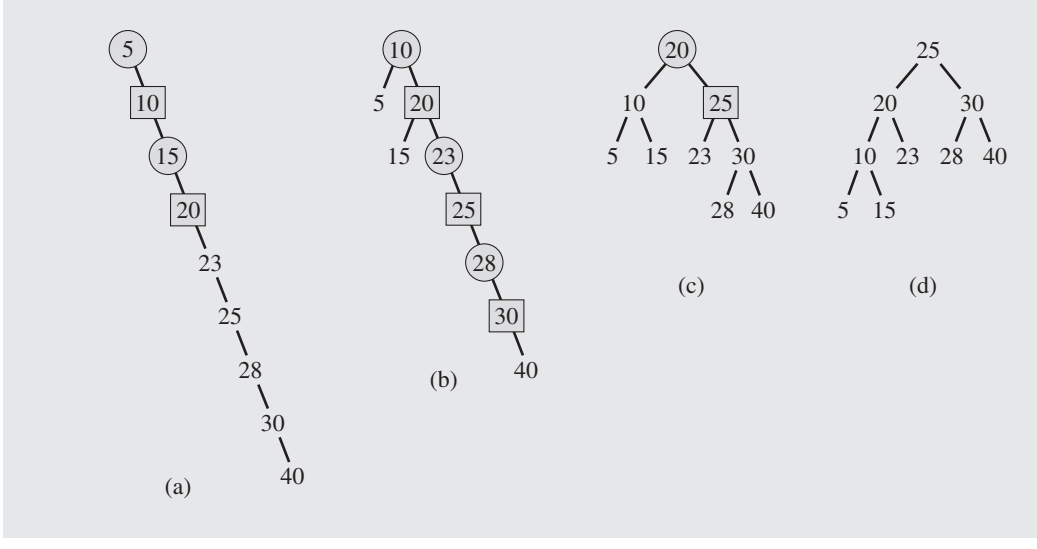
In the second phase, the backbone is transformed into a tree, but this time, the tree is perfectly balanced by having leaves only on two adjacent levels. In each pass down the backbone, every second node down to a certain point is rotated about its

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 = number of nodes;
  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.

FIGURE 6.39 Transforming a backbone into a perfectly balanced tree.



To compute the complexity of the tree building phase, observe that the number of iterations performed by the `while` loop equals

$$(2^{\lg(m+1)-1} - 1) + \cdots + 15 + 7 + 3 + 1 = \sum_{i=1}^{\lg(m+1)-1} (2^i - 1) = m - \lg(m+1)$$

The number of rotations can now be given by the formula

$$n - m + (m - \lg(m+1)) = n - \lg(m+1) = n - \lfloor \lg(n+1) \rfloor$$

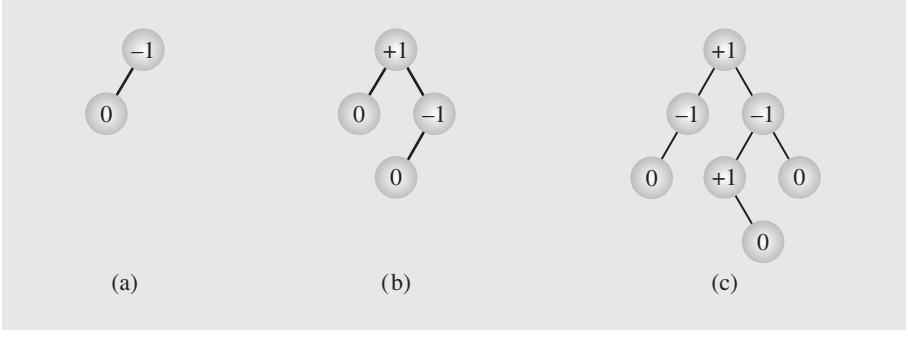
that is, the number of rotations is $O(n)$. Because creating a backbone also required at most $O(n)$ rotations, the cost of global rebalancing with the DSW algorithm is optimal in terms of time because it grows linearly with n and requires a very small and fixed amount of additional storage.

6.7.2 AVL Trees

The previous two sections discussed algorithms that 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 the 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* that are the differences between the heights of the left and right subtrees. A balance factor is 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.

FIGURE 6.40 Examples of AVL trees.



The definition of an AVL tree indicates that the minimum number of nodes in a tree is determined by the recurrence equation

$$AVL_h = AVL_{h-1} + AVL_{h-2} + 1$$

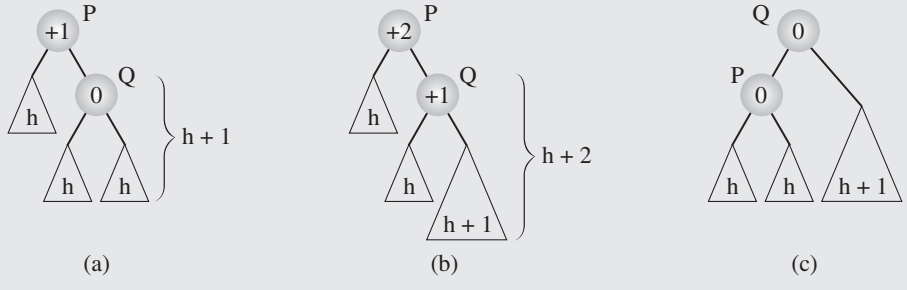
where $AVL_0 = 0$ and $AVL_1 = 1$ are the initial conditions.[1] This formula leads to the following bounds on the height h of an AVL tree depending on the number of nodes n (see Appendix A.5):

$$\lg(n + 1) \leq h < 1.44\lg(n + 2) - 0.328$$

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 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.

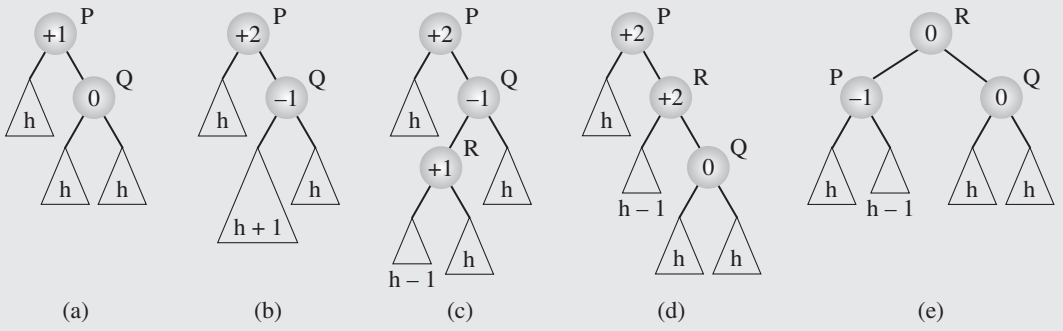
FIGURE 6.41 Balancing a tree after insertion of a node in the right subtree of node Q.



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. Note that R's balance factor can also be –1. 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).

[1]Numbers generated by this recurrence formula are called *Leonardo numbers.*

FIGURE 6.42 Balancing a tree after insertion of a node in the left subtree of node Q.



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 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 because they remain the same.

To update the balance factors, the following algorithm can be used:

```
updateBalanceFactors()
   Q = the node just inserted;
   P = parent of Q;
   if Q is the left child of P
        P->balanceFactor--;
   else P->balanceFactor++;
   while P is not the root and P->balanceFactor ≠ ±2
       Q = P;
       P = P's parent;
       if Q->balanceFactor is 0
            return;
```

```
        if Q is the left child of P
            P->balanceFactor--;
        else P->balanceFactor++;
    if P->balanceFactor is ±2
        rebalance the subtree rooted at P;
```

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.

FIGURE 6.43 An example of inserting (b) a new node in (a) an AVL tree, which requires one rotation (c) to restore the height balance.

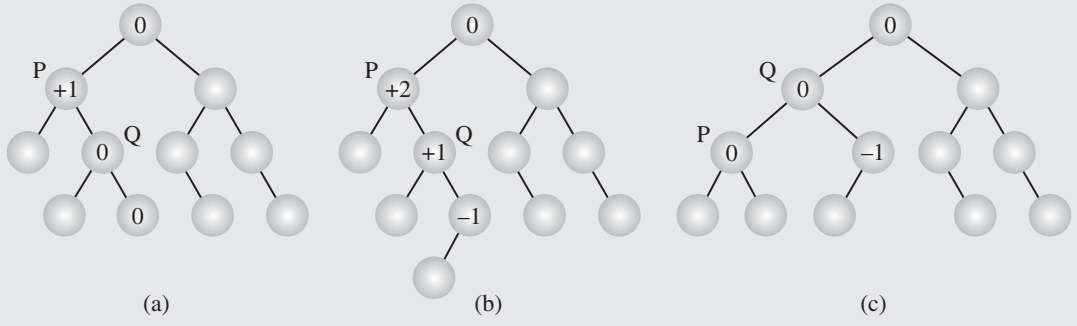
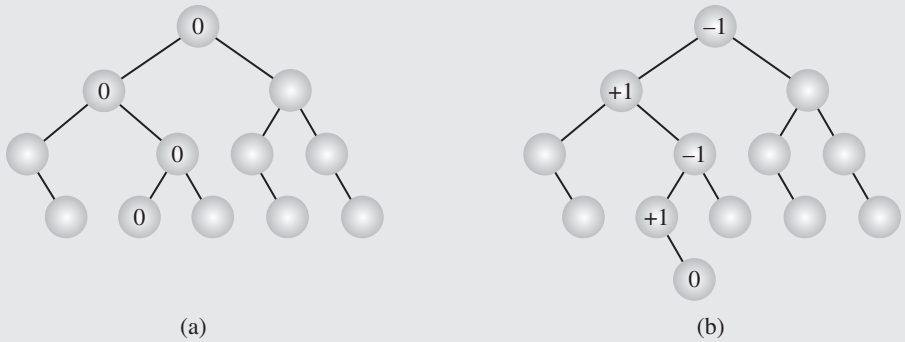


FIGURE 6.44 In an (a) AVL tree a (b) new node is inserted requiring no height adjustments.



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.

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, because 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 deleting a node, 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.

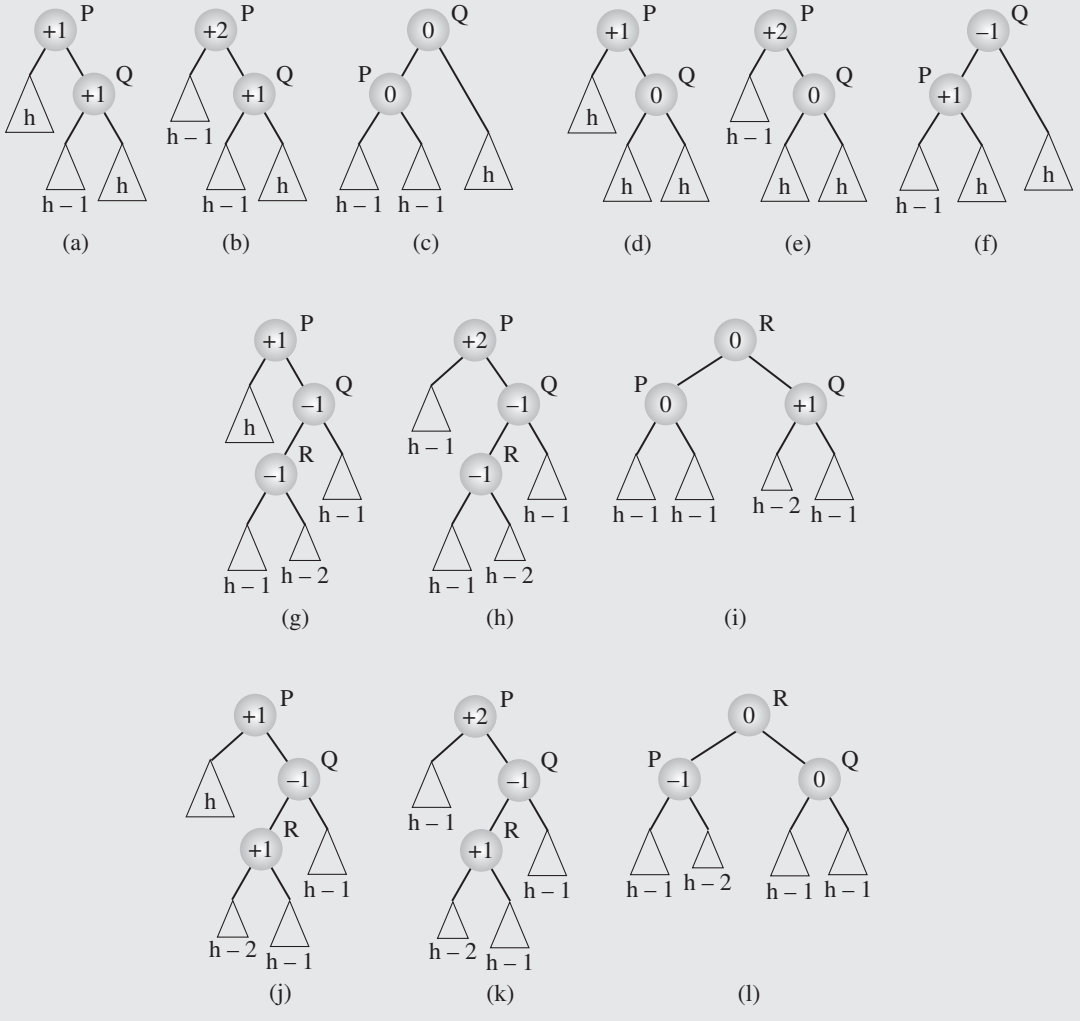
The previous analyses indicate that insertions and deletions require at most $1.44 \lg(n + 2)$ searches. Also, insertion can require one single or one double rotation, and deletion can require $1.44 \lg(n + 2)$ rotations in the worst case. But as also indicated, the average case requires $\lg(n) + .25$ 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.

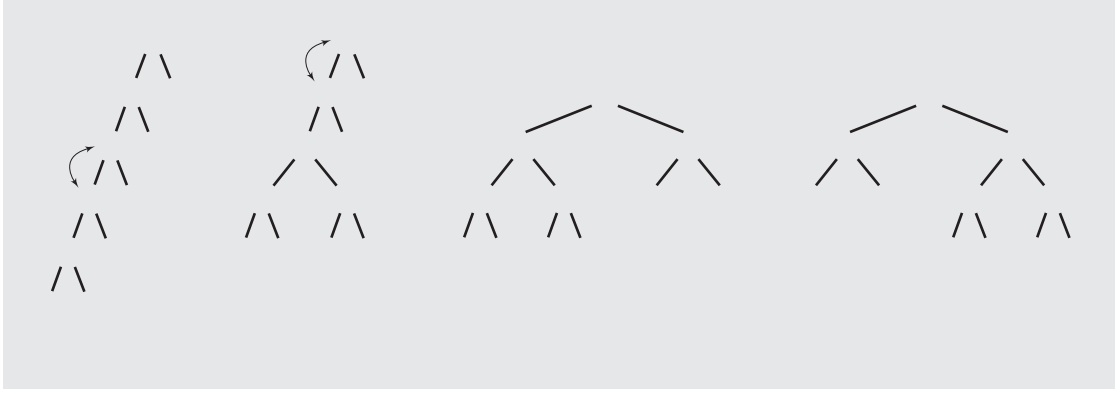
AVL trees can be extended by allowing the difference in height $\Delta > 1$ (Foster 1973). Not unexpectedly, the worst-case height increases with Δ and

$$h = \begin{cases} 1.81 \lg(n) - 0.71 & \text{if } \Delta = 2 \\ 2.15 \lg(n) - 1.13 & \text{if } \Delta = 3 \end{cases}$$

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.

FIGURE 6.45 Rebalancing an AVL tree after deleting a node.





6.9 Heaps

A particular kind of binary tree, called a *heap,* has the following two properties:

1. The value of each node is greater than or equal to 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 "greater" in the first property is replaced with "less," 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

$$\texttt{heap[i]} \geq \texttt{heap[2}\cdot\texttt{i + 1]}, \text{ for } 0 \leq i < \frac{n-1}{2}$$

FIGURE 6.51 Examples of (a) heaps and (b–c) nonheaps.

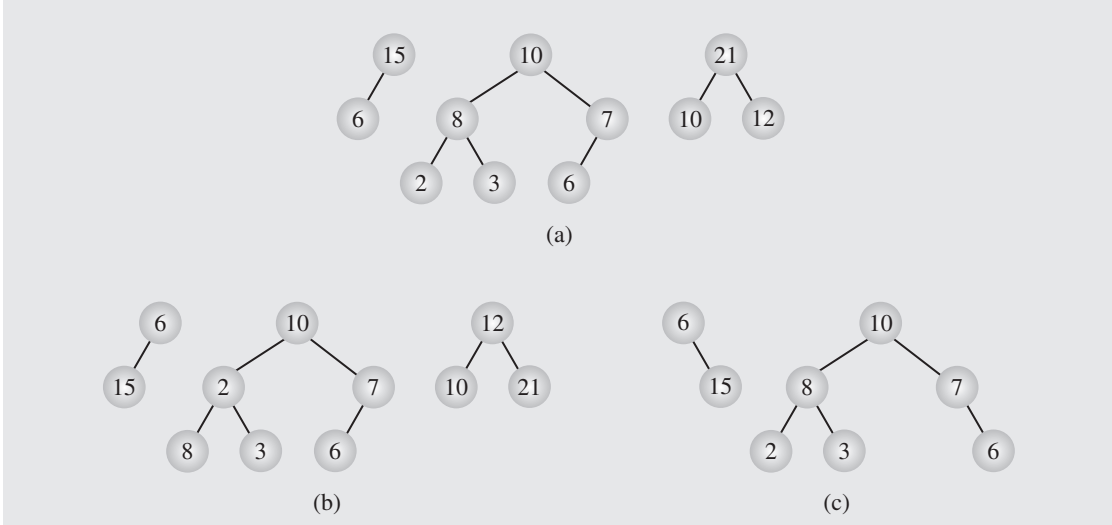
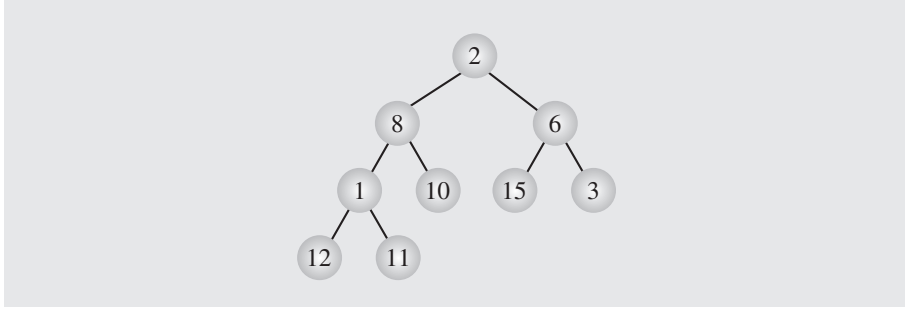


FIGURE 6.52 The array [2 8 6 1 10 15 3 12 11] seen as a tree.

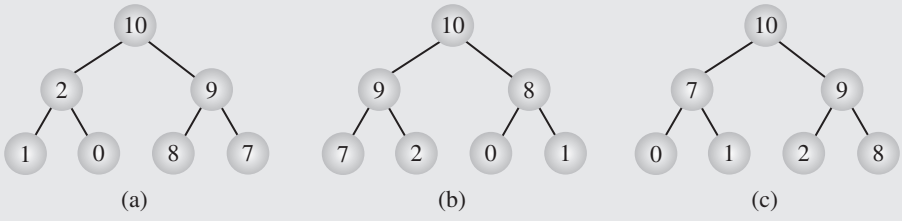


and

$$\texttt{heap[i]} \geq \texttt{heap[2 * i + 2]}, \text{ for } 0 \leq i < \frac{n-2}{2}$$

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.

FIGURE 6.53 Different heaps constructed with the same elements.



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 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;
```

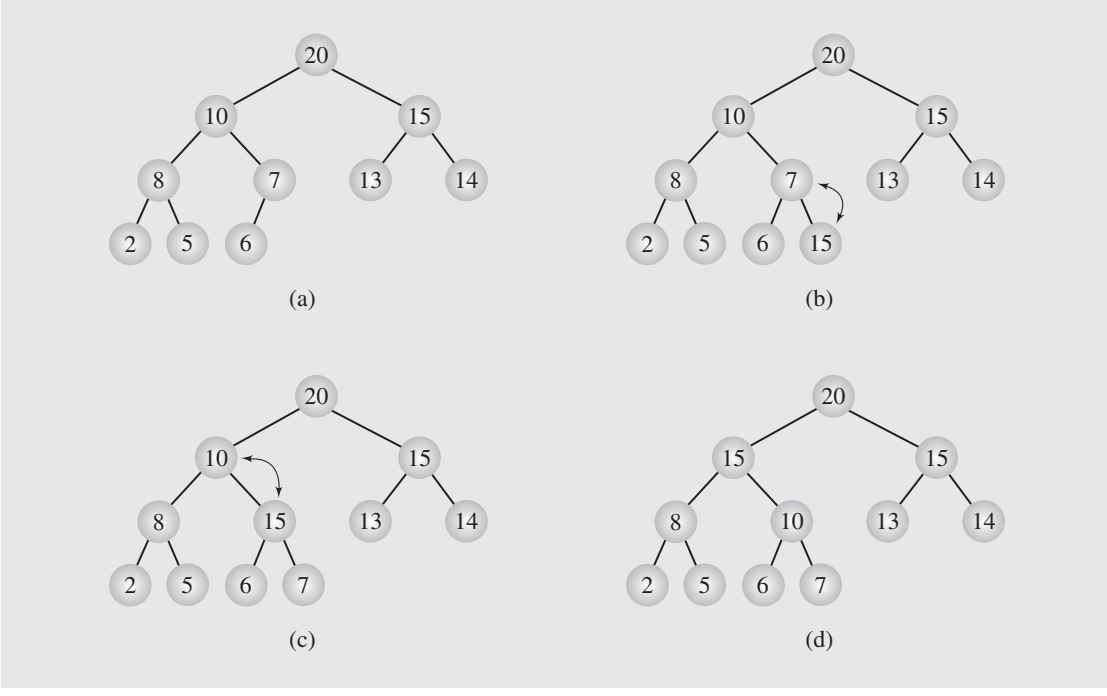
For example, the number 15 is added to the heap in Figure 6.54a as the next leaf (Figure 6.54b), which destroys the heap property of the tree. To restore this property, 15 has to be moved up the tree until it either ends up in the root or finds a parent that is not less than 15. In this example, the 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, because 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.

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;
```

FIGURE 6.54 Enqueuing an element to a heap.




```
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.

6.9.2 Organizing Arrays as Heaps

Heaps can be implemented as arrays, and in that sense, each heap is an array, but not all arrays are heaps. In some situations, however, most notably in heap sort (see 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

FIGURE 6.55 Dequeuing an element from a heap.

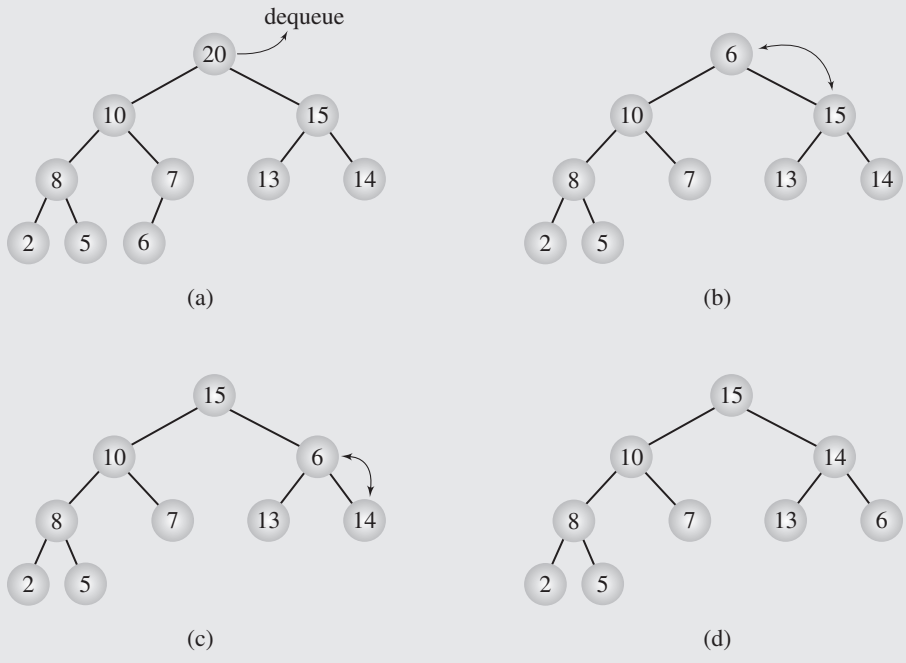


FIGURE 6.56 Implementation of an algorithm to move the root element down a tree.

```
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];
}
```

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 by the reader 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

$$\sum_{k=1}^{n} \lfloor \lg k \rfloor \leq \sum_{k=1}^{n} \lg k = \lg 1 + \cdots + \lg n = \lg(1 \cdot 2 \cdot \cdots \cdot n) = \lg(n!) = O(n \lg n)$$

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 down to 0
        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 size of the array. 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. Because 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 because 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, and so on, 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).

FIGURE 6.57 Organizing an array as a heap with a top-down method.

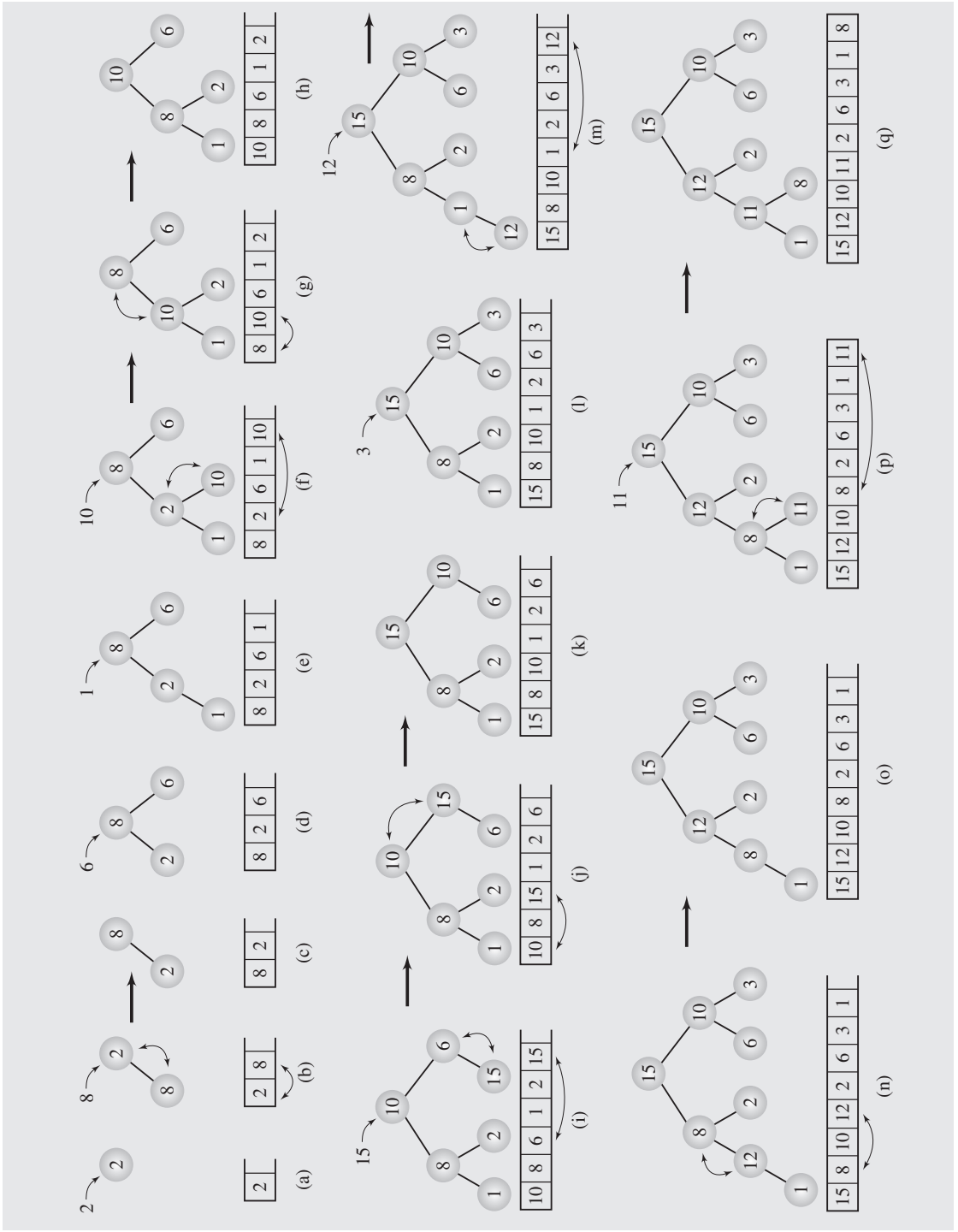
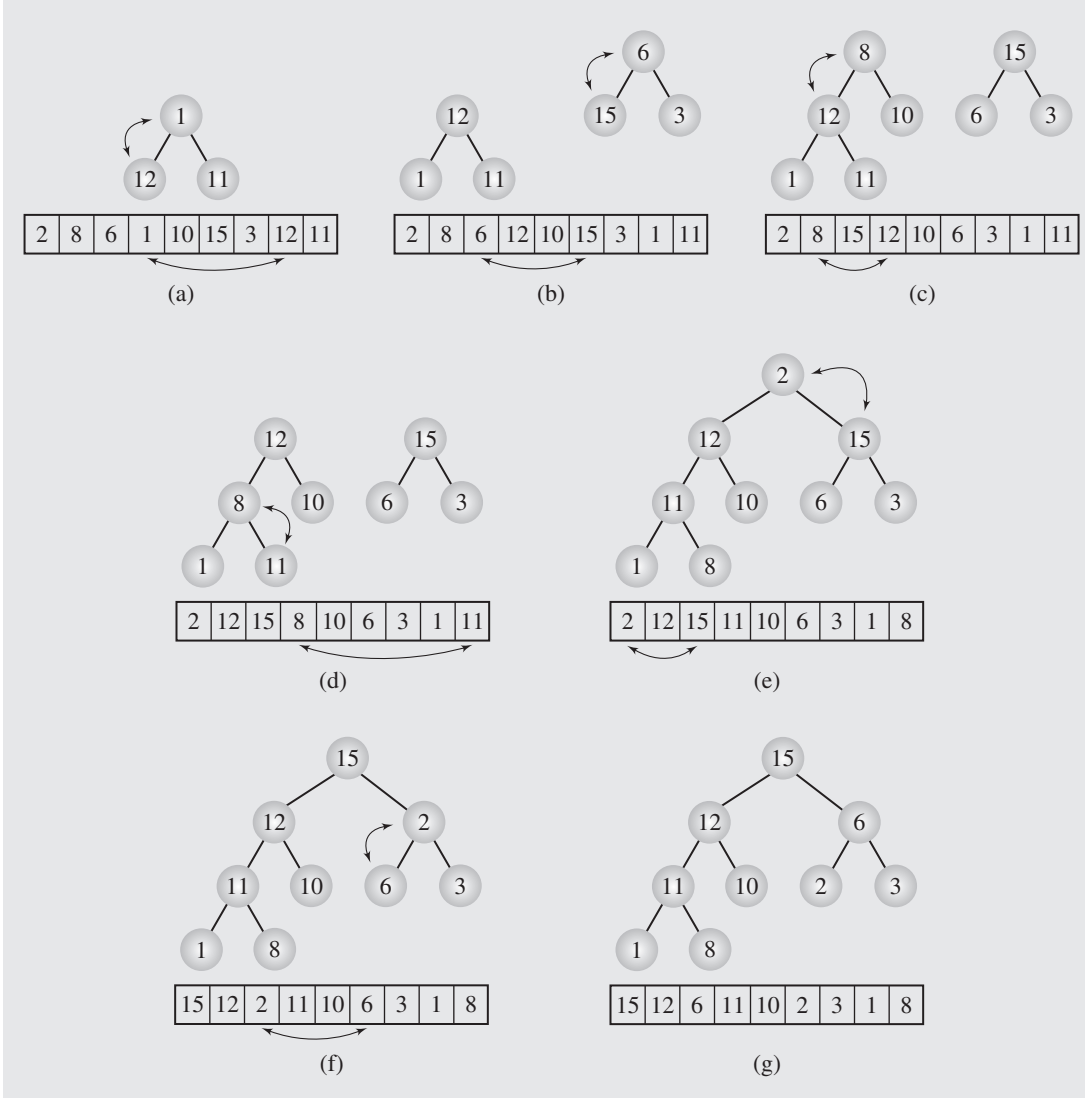


FIGURE 6.58 Transforming the array [2 8 6 1 10 15 3 12 11] into a heap with a bottom-up method.



When a new element is analyzed, both its subtrees are already heaps, as is the case with number 2, and both its subtrees 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 because 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 down the tree to end up in one of the leaves. Because 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 $O(n)$ because the series $\sum_{i=2}^{\infty} \frac{i}{2^i}$ converges to 1.5 and $\sum_{i=2}^{\infty} \frac{1}{2^i}$ converges to 0.5. For an array that 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 $O(n)$, because for each node in `moveDown()`, 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 (Doberkat 1984; Knuth 1998), 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.

Graphs

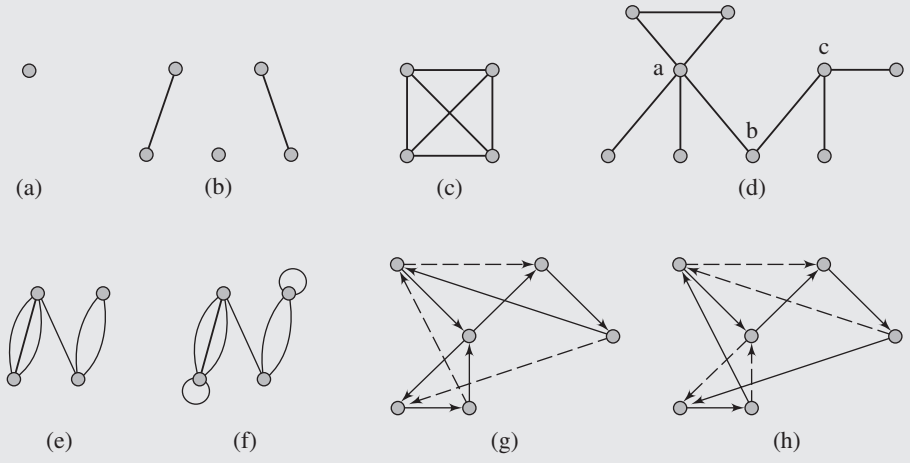
8

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, 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

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.



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*. Part of 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 various 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:

$$a_{ij} = \text{number of edges between } v_i \text{ and } v_j$$

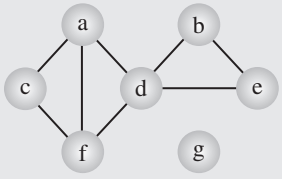
Another matrix representation of a graph is based on the incidence of vertices and edges and is called an *incidence matrix.* An incidence matrix of graph $G = (V,E)$ is a $|V| \times |E|$ matrix such that

$$a_{ij} = \begin{cases} 1 & \text{if edge } e_j \text{ is incident with vertex } v_i \\ 0 & \text{otherwise} \end{cases}$$

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.

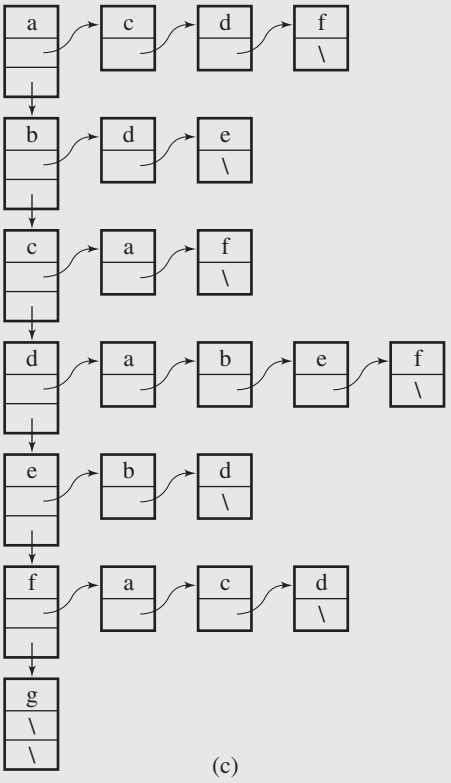
FIGURE 8.2 Graph representations. (a) A graph represented as (b–c) an adjacency list, (d) an adjacency matrix, and (e) an incidence matrix.



(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	0	1
g	0	0	0	0	0	0	0	0

(e)

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.

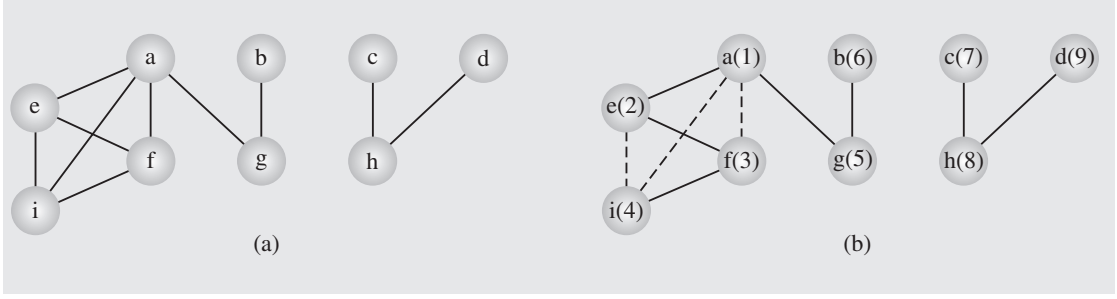
An algorithm for traversing a graph, known as the depth-first search algorithm, was developed by John Hopcroft and Robert Tarjan. In this algorithm, each vertex *v* is visited and then each unvisited vertex adjacent to *v* is visited. If a vertex *v* has no adjacent vertices or all of its adjacent vertices 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`.

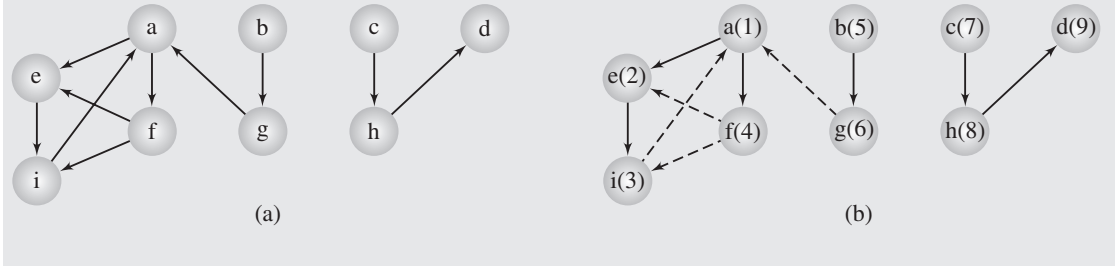
FIGURE 8.3 An example of application of the `depthFirstSearch()` algorithm to a graph.



Note that this algorithm guarantees generating a tree (or a forest, a set of trees) that 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 that 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 8.4 The `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 $O(|V|^2)$.

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) is 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 8.5 An example of application of the `breadthFirstSearch()` algorithm to a graph.

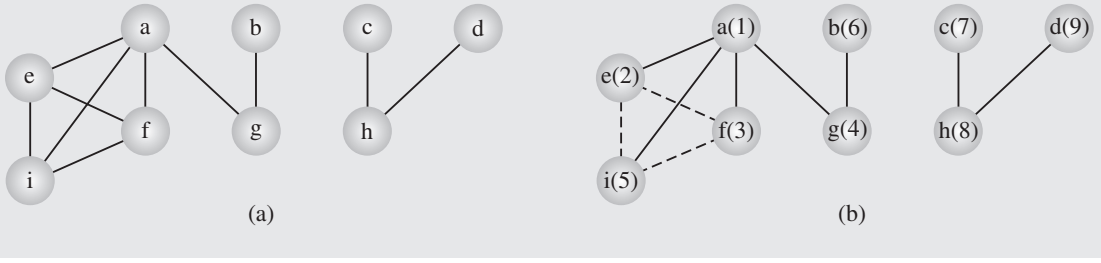
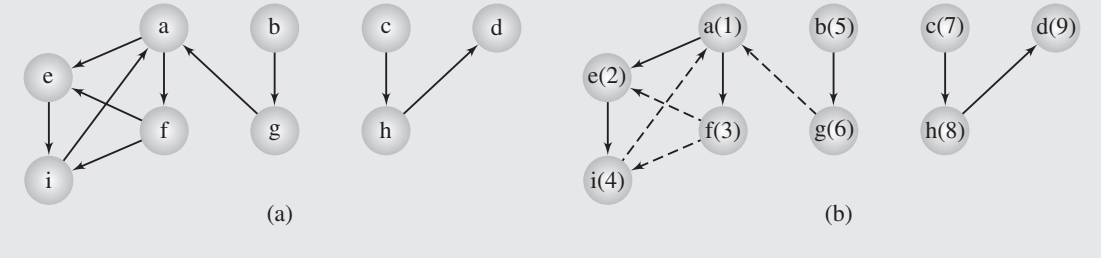


FIGURE 8.6 The `breadthFirstSearch()` algorithm applied to a digraph.



Sorting

9

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 can, 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 that 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 American Standard Code for Information Interchange (ASCII) code is commonly used, although other choices such as Extended Binary Coded Decimal Interchange Code (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.

Because 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 (it can be 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 are of different levels of complexity. A simple method can be only 20 percent 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 percent must not be neglected. Simple algorithms often perform better with a small amount of data than their more complex counterparts whose effectiveness may become obvious only 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, the third element, `data[2]`, is considered. 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

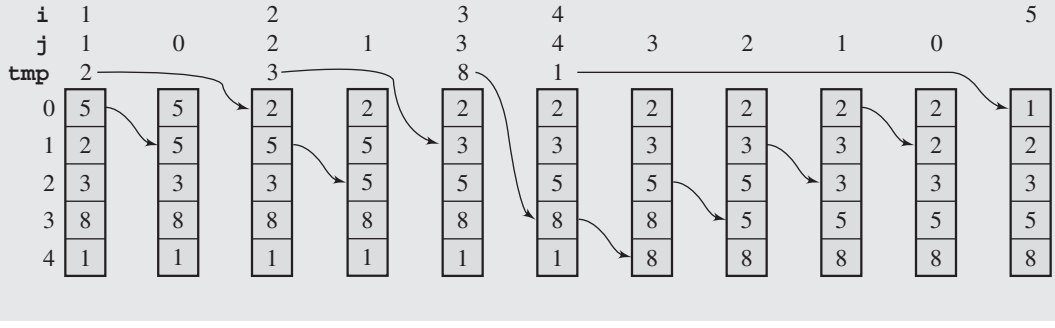
than `data[1]` and not less than `data[0]`, then only `data[1]` is moved to position 2 and its place is taken by `data[2]`. If, finally, `data[2]` is not less than both its predecessors, it stays in its current position. Each element `data[i]` is inserted into its proper location j such that $0 \leq j \leq i$, 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 to n-1
      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.

FIGURE 9.1 The array [5 2 3 8 1] sorted by insertion sort.



Because 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-1]`. 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

$$\frac{1 + 2 + \cdots + i}{i} = \frac{\frac{1}{2}i(i+1)}{i} = \frac{i+1}{2}$$

To obtain the average number of all comparisons, the computed figure has to be added for all is (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 times; that is

$$\frac{0 + 1 + \cdots + i}{i} = \frac{\frac{1}{2}i(i+1)}{i+1} = \frac{i}{2}$$

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,

$$\sum_{i=1}^{n-1}\left(\frac{i}{2} + 2\right) = \frac{1}{2}\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 2 = \frac{\frac{1}{2}n(n-1)}{2} + 2(n-1) = \frac{n^2 + 7n - 8}{4}$$

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

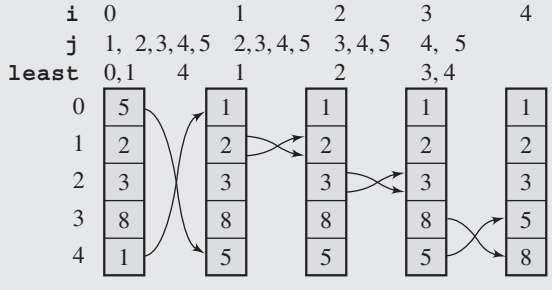
Selection sort is an attempt to localize the exchanges of array elements by finding a misplaced element first and putting it in its final place. The element with the lowest value is selected and exchanged with the element in the first position. Then, the smallest value among the remaining elements `data[1]`, . . . , `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 to n-2
      select the smallest element among data[i],...,data[n-1];
      swap it with data[i];
```

It is rather obvious that `n-2` should be the last value for `i`, because if all elements but the last have been already considered and placed in their proper positions, then the *n*th 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]);
    }
}
```

FIGURE 9.2 The array [5 2 3 8 1] sorted by selection sort.



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(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.

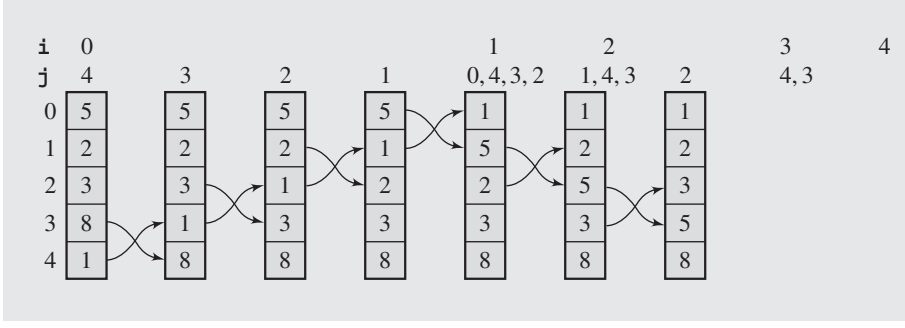
A pseudocode of the algorithm is as follows:

```
bubblesort(data[],n)
   for i = 0 to n−2
        for j = n-1 down to i+1
             swap elements in positions  j  and j-1  if they are out of order;
```

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]);
}
```

FIGURE 9.3 The array [5 2 3 8 1] sorted by bubble sort.



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. To find the number of moves in the average case, note that 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 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.

An apparent improvement of bubble sort is obtained by adding a flag to discontinue processing after a pass in which no swap was performed:

```
template<class T>
void bubblesort2(T data[], const int n) {
     bool again = true;
     for (int i = 0; i < n-1 && again; i++)
          for (int j = n-1, again = false; j > i; --j)
               if (data[j] < data[j-1]) {
                    swap(data[j],data[j-1]);
                    again = true;
               }
}
```

The improvement, however, is insignificant because in the worst case the improved bubble sort behaves just as the original one. The worst case for the number

of comparisons is when the largest element is at the very beginning of data before sorting starts because this element can be moved only by one position in each pass. There are $(n - 1)!$ such worst cases in an array in which all elements are different. The cases, when the second largest element is at the beginning or the largest element is in the second position, and there are also $(n - 1)!$ such cases, are just as bad (only one fewer pass would be needed than in the worst case). The cases when the third largest element is in the first position are not far behind, etc. Therefore, very seldom the flag `again` fulfills its duty and very often – because an additional variable has to be maintained by `bubblesort2()` – the improved version is even slower than `bubblesort()`. Therefore, by itself, `bubblesort2()` is not an interesting modification of bubble sort. But comb sort, which builds on `bubblesort2()`, certainly is.

10 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 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 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 that can transform a particular key K, be it a string, number, record, or the like, 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 1,000 cells is usually adequate.

But even if this table can accommodate all the variables in the program, how can we design a function h that 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 3,782 cells (for a variable K made out of 31 letters "z," $h(K) = 31 \cdot 122 = 3{,}782$). 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, 1,368, can be divided modulo *TSize* or, if the size of the table is 1,000, 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 that 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 = 1,566.

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. An integer in a C++ implementation for the IBM PC computer is 2 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 3,121, then $3{,}121^2 = 9{,}740{,}641$, and for the 1,000-cell table, $h(3{,}121) = 406$, which is the middle part of $3{,}121^2$. 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 1,024, then, in this example, the binary representation of $3{,}121^2$ is the bit string 100101000101000001100001, 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, 1,234; the last four, 6,789; the first two combined with the last two, 1,289; 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 that 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., 1133 for the publishing company Cengage Learning). 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.

10.1.6 Universal Hash Functions

When very little is known about keys, a *universal class of hash functions* can be used; a class of functions is universal when for any sample, a randomly chosen member of that class will be expected to distribute the sample evenly, whereby members of that class guarantee low probability of collisions (Carter and Wegman 1979).

Let H be a class of functions from a set of *keys* to a hash table of *TSize*. We say that H is universal if for any two different keys x and y from *keys*, the number of hash functions h in H for which $h(x) = h(y)$ equals $|H|/TSize$. That is, H is universal if no pair of distinct keys are mapped into the same index by a randomly chosen function h with the probability equal to $1/TSize$. In other words, there is one chance in *TSize* that two keys collide when a randomly picked hash function is applied. One class of such functions is defined as follows.

For a prime number $p \geq |keys|$, and randomly chosen numbers a and b,

$$H = \{h_{a,b}(K): h_{a,b}(K) = ((aK+b) \bmod p) \bmod TSize \text{ and } 0 \leq a, b < p\}$$

Another example is a class H for keys considered to be sequences of bytes, $K = K_0K_1 \ldots K_{r-1}$. For some prime $p \geq 2^8 = 256$ and a sequence $a = a_0, a_1, \ldots, a_{r-1}$,

$$H = \{h_a(K): h_a(K) = \left(\left(\sum_{i=0}^{r-1} a_iK_i\right) \bmod p\right) \bmod TSize \text{ and } 0 \leq a_0, a_1, \ldots, a_{r-1} < p\}$$

10.2 Collision Resolution

Note that straightforward hashing is not without its problems, because 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 that 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 has 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.

10.2.1 Open Addressing

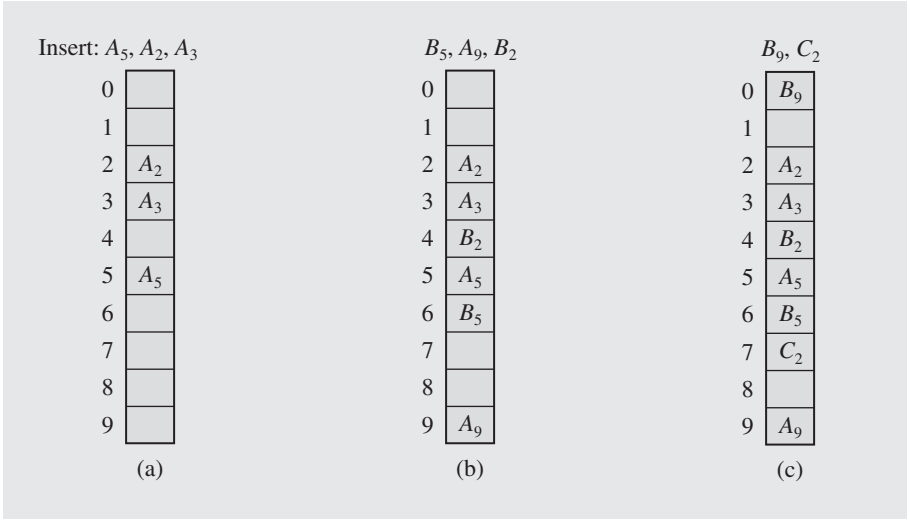
In the open addressing method, when a key collides with another key, the collision is resolved by finding an available table entry other than the position (address) to which the colliding key is originally hashed. If position $h(K)$ is occupied, then the positions in the probing sequence

$$norm(h(K) + p(1)), norm(h(K) + p(2)), \ldots, norm(h(K) + p(i)), \ldots$$

are tried until either an available cell is found or the same positions are tried repeatedly or the table is full. Function p is a *probing function,* i is a *probe,* and *norm* is a *normalization function,* most likely, division modulo the size of the table.

The simplest method is *linear probing,* for which $p(i) = i$, and for the ith probe, the position to be tried is $(h(K) + i)$ mod *TSize*. In linear probing, the position in which a key can be stored is found by sequentially searching all positions starting from the position calculated by the hash function until an empty cell is found. If the end of the table is reached and no empty cell has been found, the search is continued from the beginning of the table and stops—in the extreme case—in the cell preceding the one from which the search started. Linear probing, however, has a tendency to create clusters in the table. Figure 10.1 contains an example where a key K_i is hashed to the position i. In Figure 10.1a, three keys—A_5, A_2, and A_3—have been hashed to their home positions. Then B_5 arrives (Figure 10.1b), whose home position is occupied by A_5. Because the next position is available, B_5 is stored there. Next, A_9 is stored with no problem, but B_2 is stored in position 4, two positions from its home address. A large cluster has already been formed. Next, B_9 arrives. Position 9 is not available, and because it is the last cell of the table, the search starts from the beginning of the table, whose first slot can now host B_9. The next key, C_2, ends up in position 7, five positions from its home address.

FIGURE 10.1 Resolving collisions with the linear probing method. Subscripts indicate the home positions of the keys being hashed.



In this example, the empty cells following clusters have a much greater chance to be filled than other positions. This probability is equal to (*sizeof*(*cluster*) + 1)/*TSize*. Other empty cells have only 1/*TSize* chance of being filled. If a cluster is created, it has a tendency to grow, and the larger a cluster becomes, the larger the likelihood that it will become even larger. This fact undermines the performance of the hash table for storing and retrieving data. The problem at hand is how to avoid cluster buildup. An answer can be found in a more careful choice of the probing function *p*.

One such choice is a quadratic function so that the resulting formula is

$$p(i) = h(K) + (-1)^{i-1}((i + 1)/2)^2 \text{ for } i = 1, 2, \ldots, TSize - 1$$

This rather cumbersome formula can be expressed in a simpler form as a sequence of probes:

$$h(K) + i^2, h(K) - i^2 \text{ for } i = 1, 2, \ldots, (TSize - 1)/2$$

Including the first attempt to hash *K,* this results in the sequence:

$$h(K), h(K) + 1, h(K) - 1, h(K) + 4, h(K) - 4, \ldots, h(K) + (TSize - 1)^2/4,$$

$$h(K) - (TSize - 1)^2/4$$

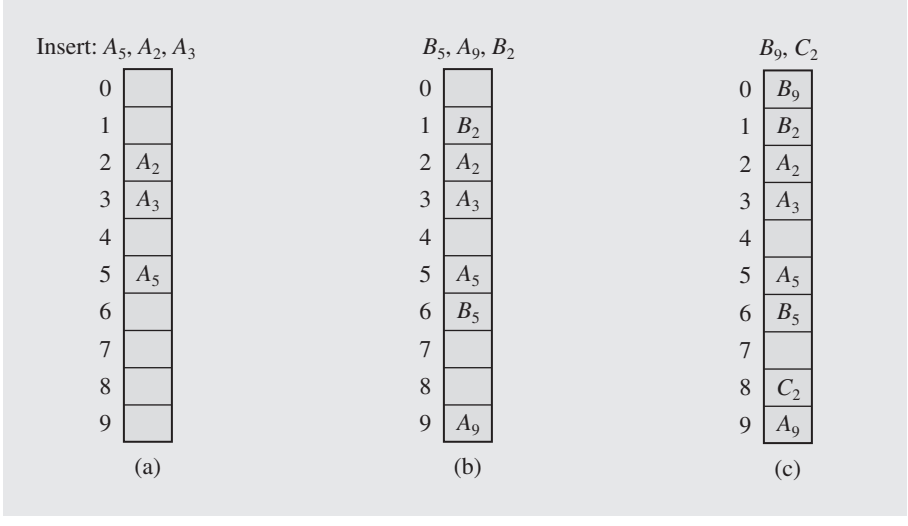
all divided modulo *TSize.* The size of the table should not be an even number, because only the even positions or only the odd positions are tried depending on the value of $h(K)$. Ideally, the table size should be a prime $4j + 3$ of an integer *j,* which

guarantees the inclusion of all positions in the probing sequence (Radke 1970). For example, if $j = 4$, then $TSize = 19$, and assuming that $h(K) = 9$ for some K, the resulting sequence of probes is[1]

9, 10, 8, 13, 5, 18, 0, 6, 12, 15, 3, 7, 11, 1, 17, 16, 2,14, 4

The table from Figure 10.1 would have the same keys in a different configuration, as in Figure 10.2. It still takes two probes to locate B_2 in some location, but for C_2, only four probes are required, not five.

FIGURE 10.2 Using quadratic probing for collision resolution.



Note that the formula determining the sequence of probes chosen for quadratic probing is not $h(K) + i^2$, for $i = 1, 2, \ldots, TSize - 1$, because the first half of the sequence

$$h(K) + 1, h(K) + 4, h(K) + 9, \ldots, h(K) + (TSize - 1)^2$$

covers only half of the table, and the second half of the sequence repeats the first half in the reverse order. For example, if $TSize = 19$, and $h(K) = 9$, then the sequence is

9, 10, 13, 18, 6, 15, 7, 1, 16, 14, 14, 16, 1, 7, 15, 6, 18, 13, 10

[1]Special care should be taken for negative numbers. When implementing these formulas, the operator % means division modulo a modulus. However, this operator is usually implemented as the *remainder* of division. For example, –6 % 23 is equal to –6, and not to 17, as expected. Therefore, when using the operator % for the implementation of division modulo, the modulus (the right operand of %) should be added to the result when the result is negative. Therefore, (–6 % 23) + 23 returns 17.

This is not an accident. The probes that render the same address are of the form

$$i = TSize/2 + 1 \text{ and } j = TSize/2 - 1$$

and they are probes for which

$$i^2 \bmod TSize = j^2 \bmod TSize$$

that is,

$$(i^2 - j^2) \bmod TSize$$

In this case,

$$\begin{aligned}(i^2 - j^2) &= (TSize/2 + 1)^2 - (TSize/2 - 1)^2 \\ &= (TSize^2/4 + TSize + 1 - TSize^2/4 + TSize - 1) \\ &= 2TSize\end{aligned}$$

and to be sure, $2TSize \bmod TSize = 0$.

Although using quadratic probing gives much better results than linear probing, the problem of cluster buildup is not avoided altogether, because for keys hashed to the same location, the same probe sequence is used. Such clusters are called *secondary clusters.* These secondary clusters, however, are less harmful than primary clusters.

Another possibility is to have p be a random number generator (Morris 1968), which eliminates the need to take special care about the table size. This approach prevents the formation of secondary clusters, but it causes a problem with repeating the same probing sequence for the same keys. If the random number generator is initialized at the first invocation, then different probing sequences are generated for the same key K. Consequently, K is hashed more than once to the table, and even then it might not be found when searched. Therefore, the random number generator should be initialized to the same seed for the same key before beginning the generation of the probing sequence. This can be achieved in C++ by using the `srand()` function with a parameter that depends on the key; for example, $p(i) =$ `srand(`*sizeof*$(K))\cdot i$ or `srand(`K`[0])` $+ i$. To avoid relying on `srand()`, a random number generator can be written that assures that each invocation generates a unique number between 0 and $TSize - 1$. The following algorithm was developed by Robert Morris for tables with $TSize = 2^n$ for some integer n:

```
generateNumber()
   static int r = 1;
   r = 5*r;
   r = mask out n + 2 low-order bits of r;
   return r/4;
```

The problem of secondary clustering is best addressed with *double hashing.* This method utilizes two hash functions, one for accessing the primary position of a key, *h,* and a second function, h_p, for resolving conflicts. The probing sequence becomes

$$h(K), h(K) + h_p(K), \ldots, h(K) + i \cdot h_p(K), \ldots$$

(all divided modulo *TSize*). The table size should be a prime number so that each position in the table can be included in the sequence. Experiments indicate that secondary clustering is generally eliminated because the sequence depends on the values of h_p, which, in turn, depend on the key. Therefore, if the key K_1 is hashed to the position *j,* the probing sequence is

$$j, j + h_p(K_1), j + 2 \cdot h_p(K_1), \ldots$$

(all divided modulo *TSize*). If another key K_2 is hashed to $j + h_p(K_1)$, then the next position tried is $j + h_p(K_1) + h_p(K_2)$, not $j + 2 \cdot h_p(K_1)$, which avoids secondary clustering if h_p is carefully chosen. Also, even if K_1 and K_2 are hashed primarily to the same position *j,* the probing sequences can be different for each. This, however, depends on the choice of the second hash function, h_p, which may render the same sequences for both keys. This is the case for function $h_p(K) =$ `strlen`(K), when both keys are of the same length.

Using two hash functions can be time-consuming, especially for sophisticated functions. Therefore, the second hash function can be defined in terms of the first, as in $h_p(K) = i \cdot h(K) + 1$. The probing sequence for K_1 is

$$j, 2j + 1, 5j + 2, \ldots$$

(modulo *TSize*). If K_2 is hashed to $2j + 1$, then the probing sequence for K_2 is

$$2j + 1, 4j + 3, 10j + 11, \ldots$$

which does not conflict with the former sequence. Thus, it does not lead to cluster buildup.

How efficient are all these methods? Obviously, it depends on the size of the table and on the number of elements already in the table. The inefficiency of these methods is especially evident for *unsuccessful searches,* searching for elements not in the table. The more elements in the table, the more likely it is that clusters will form (primary or secondary) and the more likely it is that these clusters are large.

Consider the case when linear probing is used for collision resolution. If *K* is not in the table, then starting from the position $h(K)$, all consecutively occupied cells are checked; the longer the cluster, the longer it takes to determine that *K,* in fact, is not in the table. In the extreme case, when the table is full, we have to check all the cells starting from $h(K)$ and ending with $(h(K) - 1) \bmod TSize$. Therefore, the search time increases with the number of elements in the table.

There are formulas that approximate the number of times for successful and unsuccessful searches for different hashing methods. These formulas were developed by Donald Knuth and are considered by Thomas Standish to be "among the prettiest in computer science." Figure 10.3 contains these formulas. Figure 10.4 contains a table showing the number of searches for different percentages of occupied cells. This table indicates that the formulas from Figure 10.3 provide only approximations of the number of searches. This is particularly evident for the higher percentages. For example, if 90 percent of the cells are occupied, then linear probing requires 50 trials to determine that the key being searched for is not in the table. However, for the full table of 10 cells, this number is 10, not 50.

FIGURE 10.3 Formulas approximating, for different hashing methods, the average numbers of trials for successful and unsuccessful searches (Knuth 1998).

	linear probing	quadratic probing[a]	double hashing
successful search	$\frac{1}{2}\left(1 + \frac{1}{1 - \text{LF}}\right)$	$1 - 1\text{n}(1 - \text{LF}) - \frac{\text{LF}}{2}$	$\frac{1}{\text{LF}} \ln \frac{1}{1 - \text{LF}}$
unsuccessful search	$\frac{1}{2}\left(1 + \frac{1}{(1 - \text{LF})^2}\right)$	$\frac{1}{1 - \text{LF}} - \text{LF} - 1\text{n}(1 - \text{LF})$	$\frac{1}{1 - \text{LF}}$

$$\text{Loading factor} \quad \text{LF} = \frac{\text{number of elements in the table}}{\text{table size}}$$

[a] The formulas given in this column approximate any open addressing method that causes secondary clusters to arise, and quadratic probing is only one of them.

FIGURE 10.4 The average numbers of successful searches and unsuccessful searches for different collision resolution methods.

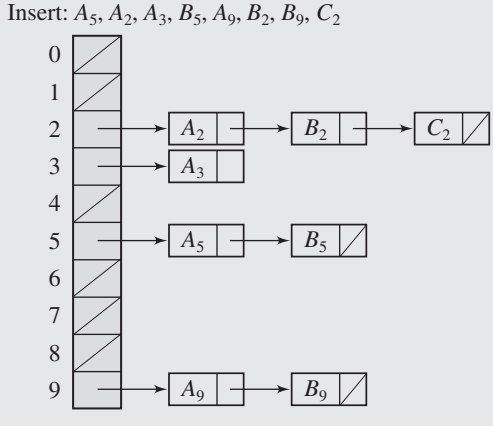
	Linear Probing		Quadratic Probing		Double Hashing	
LF	Successful	Unsuccessful	Successful	Unsuccessful	Successful	Unsuccessful
0.05	1.0	1.1	1.0	1.1	1.0	1.1
0.10	1.1	1.1	1.1	1.1	1.1	1.1
0.15	1.1	1.2	1.1	1.2	1.1	1.2
0.20	1.1	1.3	1.1	1.3	1.1	1.2
0.25	1.2	1.4	1.2	1.4	1.2	1.3
0.30	1.2	1.5	1.2	1.5	1.2	1.4
0.35	1.3	1.7	1.3	1.6	1.2	1.5
0.40	1.3	1.9	1.3	1.8	1.3	1.7
0.45	1.4	2.2	1.4	2.0	1.3	1.8
0.50	1.5	2.5	1.4	2.2	1.4	2.0
0.55	1.6	3.0	1.5	2.5	1.5	2.2
0.60	1.8	3.6	1.6	2.8	1.5	2.5
0.65	1.9	4.6	1.7	3.3	1.6	2.9
0.70	2.2	6.1	1.9	3.8	1.7	3.3
0.75	2.5	8.5	2.0	4.6	1.8	4.0
0.80	3.0	13.0	2.2	5.8	2.0	5.0
0.85	3.8	22.7	2.5	7.7	2.2	6.7
0.90	5.5	50.5	2.9	11.4	2.6	10.0
0.95	10.50	200.5	3.5	22.0	3.2	20.0

For the lower percentages, the approximations computed by these formulas are closer to the real values. The table in Figure 10.4 indicates that if the table is 65 percent full, then linear probing requires, on average, fewer than two trials to find an element in the table. Because this number is usually an acceptable limit for a hash function, linear probing requires 35 percent of the spaces in the table to be unoccupied to keep performance at an acceptable level. This may be considered too wasteful, especially for very large tables or files. This percentage is lower for a quadratic probing (25 percent) and for double hashing (20 percent), but it may still be considered large. Double hashing requires one cell out of five to be empty, which is a relatively high fraction. But all these problems can be solved by allowing more than one item to be stored in a given position or in an area associated with one position.

10.2.2 Chaining

Keys do not have to be stored in the table itself. In *chaining,* each position of the table is associated with a linked list or *chain* of structures whose `info` fields store keys or references to keys. This method is called *separate chaining,* and a table of references (pointers) is called a *scatter table.* In this method, the table can never overflow, because the linked lists are extended only upon the arrival of new keys, as illustrated in Figure 10.5. For short linked lists, this is a very fast method, but increasing the length of these lists can significantly degrade retrieval performance. Performance can be improved by maintaining an order on all these lists so that, for unsuccessful searches, an exhaustive search is not required in most cases or by using self-organizing linked lists (Pagli 1985).

FIGURE 10.5 In chaining, colliding keys are put on the same linked list.



This method requires additional space for maintaining pointers. The table stores only pointers, and each node requires one pointer field. Therefore, for n keys, $n + TSize$ pointers are needed, which for large n can be a very demanding requirement.

A version of chaining called *coalesced hashing* (or *coalesced chaining*) combines linear probing with chaining. In this method, the first available position is found for a key colliding with another key, and the index of this position is stored with the key already in the table. In this way, a sequential search down the table can be avoided by directly accessing the next element on the linked list. Each position *pos* of the table stores an object with two members: `info` for a key and `next` with the index of the next key that is hashed to *pos*. Available positions can be marked by, say, –2 in `next`; –1 can be used to indicate the end of a chain. This method requires $TSize \cdot sizeof(\texttt{next})$ more space for the table in addition to the space required for the keys. This is less than for chaining, but the table size limits the number of keys that can be hashed into the table.

An overflow area known as a *cellar* can be allocated to store keys for which there is no room in the table. This area should be located dynamically if implemented as a list of arrays.

Figure 10.6 illustrates an example where coalesced hashing puts a colliding key in the last position of the table. In Figure 10.6a, no collision occurs. In Figure 10.6b, B_5 is put in the last cell of the table, which is found occupied by A_9 when it arrives. Hence, A_9 is attached to the list accessible from position 9. In Figure 10.6c, two new colliding keys are added to the corresponding lists.

FIGURE 10.6 Coalesced hashing puts a colliding key in the last available position of the table.

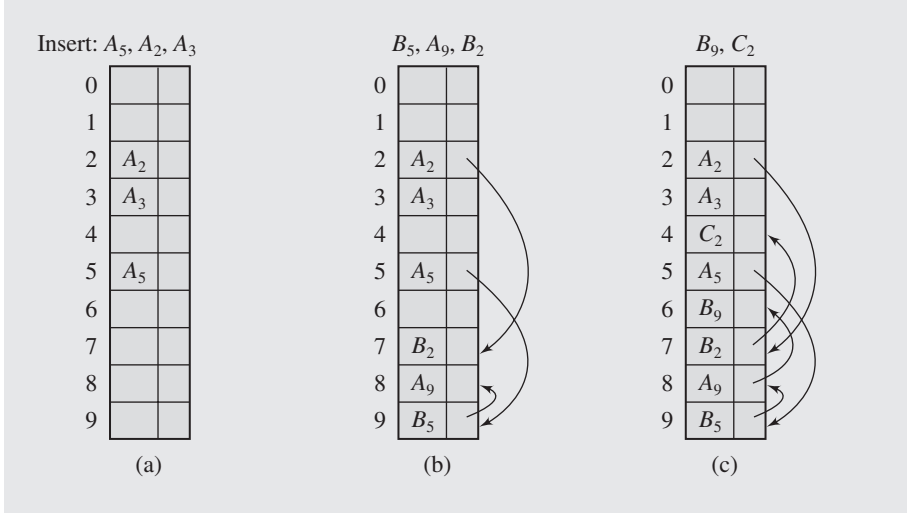


Figure 10.7 illustrates coalesced hashing that uses a cellar. Noncolliding keys are stored in their home positions, as in Figure 10.7a. Colliding keys are put in the last available slot of the cellar and added to the list starting from their home position, as in Figure 10.7b. In Figure 10.7c, the cellar is full, so an available cell is taken from the table when C_2 arrives.

FIGURE 10.7 Coalesced hashing that uses a cellar.