



6

Transform-and-Conquer

That's the secret to life . . . replace one worry with another.

—Charles M. Schulz (1922–2000), American cartoonist, the creator of *Peanuts*

This chapter deals with a group of design methods that are based on the idea of transformation. We call this general technique *transform-and-conquer* because these methods work as two-stage procedures. First, in the transformation stage, the problem's instance is modified to be, for one reason or another, more amenable to solution. Then, in the second or conquering stage, it is solved.

There are three major variations of this idea that differ by what we transform a given instance to (Figure 6.1):

- transformation to a simpler or more convenient instance of the same problem—we call it *instance simplification*
- transformation to a different representation of the same instance—we call it *representation change*
- transformation to an instance of a different problem for which an algorithm is already available—we call it *problem reduction*

In the first three sections of this chapter, we will encounter examples of the instance simplification variety. Section 6.1 deals with the simple but fruitful idea of presorting. Many questions about lists are easier to answer

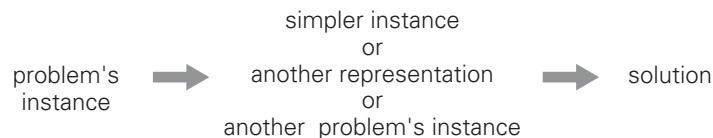


FIGURE 6.1 Transform-and-conquer strategy

if the lists are sorted. Of course, the benefits of a sorted list should more than compensate for the time spent on sorting; otherwise, we would be better off dealing with an unsorted list directly. Section 6.2 introduces one of the most important algorithms in applied mathematics: Gaussian elimination. This algorithm solves a system of linear equations by first transforming it to another system with a special property that makes finding a solution quite easy. In Section 6.3, the ideas of instance simplification and representation change are applied to search trees. The results are AVL trees and multiway balanced search trees; of the latter we consider the simplest case, 2–3 trees.

Section 6.4 presents heaps and heapsort. Even if you are already familiar with this important data structure and its application to sorting, you can still benefit from looking at them in this new light of transform-and-conquer design. In Section 6.5, we discuss Horner’s rule, a remarkable algorithm for evaluating polynomials. If there were an Algorithm Hall of Fame, Horner’s rule would be a serious candidate for induction based on the algorithm’s elegance and efficiency. We also consider there two algorithms for the exponentiation problem, both based on the representation change idea.

The chapter concludes with a review of several applications of the third variety of transform-and-conquer: problem reduction. This variety should be considered the most radical of the three: one problem is reduced to another, i.e., transformed into an entirely different problem. This is a very powerful idea, and it is extensively used in the complexity theory (Chapter 10). Its application to designing practical algorithms is not trivial, however. First, we need to identify a new problem into which the given problem should be transformed. Then we must make sure that the transformation algorithm followed by the algorithm for solving the new problem is time efficient compared to other algorithmic alternatives. Among several examples, we discuss an important special case of *mathematical modeling*, or expressing a problem in terms of purely mathematical objects such as variables, functions, and equations.

6.1 Presorting

Presorting is an old idea in computer science. In fact, interest in sorting algorithms is due, to a significant degree, to the fact that many questions about a list are easier to answer if the list is sorted. Obviously, the time efficiency of algorithms that involve sorting may depend on the efficiency of the sorting algorithm being used. For the sake of simplicity, we assume throughout this section that lists are implemented as arrays, because some sorting algorithms are easier to implement for the array representation.

So far, we have discussed three elementary sorting algorithms—selection sort, bubble sort, and insertion sort—that are quadratic in the worst and average cases and two advanced algorithms—mergesort, which is always in $\Theta(n \log n)$, and quicksort, whose efficiency is also $\Theta(n \log n)$ in the average case but is quadratic in the worst case. Are there faster sorting algorithms? As we have already stated in Section 1.3 (see also Section 10.2), no general comparison-based sorting algorithm can have a better efficiency than $n \log n$ in the worst case and the same result holds for average-case efficiency.¹

Following are three examples that illustrate the idea of presorting. More examples can be found in the exercises to this section.

EXAMPLE 1 *Checking element uniqueness in an array*

If this element uniqueness problem looks familiar to you, it should; we considered a brute-force algorithm for the problem in Section 2.3 (see Example 2). The brute-force algorithm compared pairs of the array’s elements until either two equal elements were found or no more pairs were left. Its worst-case efficiency was in $\Theta(n^2)$.

Alternatively, we can sort the array first and then check only its consecutive elements: if the array has equal elements, a pair of them must be next to each other and vice versa.

ALGORITHM *PresortElementUniqueness(A[0..n – 1])*

```
//Solves the element uniqueness problem by sorting the array first
//Input: An array A[0..n – 1] or orderable elements
//Output: Returns “true” if A has no equal elements, “false” otherwise
Sort the array A
for  $i \leftarrow 0$  to  $n - 2$  do
    if  $A[i] = A[i + 1]$  return false
return true
```

The running time of this algorithm is the sum of the time spent on sorting and the time spent on checking consecutive elements. Since the former requires at least $n \log n$ comparisons and the latter needs no more than $n - 1$ comparisons, it is the sorting part that will determine the overall efficiency of the algorithm. So, if we use a quadratic sorting algorithm here, the entire algorithm will not be more efficient than the brute-force one. But if we use a good sorting algorithm, such as mergesort, with worst-case efficiency in $\Theta(n \log n)$, the worst-case efficiency of the entire presorting-based algorithm will be also in $\Theta(n \log n)$:

$$T(n) = T_{\text{sort}}(n) + T_{\text{scan}}(n) \in \Theta(n \log n) + \Theta(n) = \Theta(n \log n). \quad \blacksquare$$

1. Sorting algorithms called *radix sorts* are linear but in terms of the total number of input bits. These algorithms work by comparing individual bits or pieces of keys rather than keys in their entirety. Although the running time of these algorithms is proportional to the number of input bits, they are still essentially $n \log n$ algorithms because the number of bits per key must be at least $\log_2 n$ in order to accommodate n distinct keys of input.

EXAMPLE 2 *Computing a mode*

A **mode** is a value that occurs most often in a given list of numbers. For example, for 5, 1, 5, 7, 6, 5, 7, the mode is 5. (If several different values occur most often, any of them can be considered a mode.) The brute-force approach to computing a mode would scan the list and compute the frequencies of all its distinct values, then find the value with the largest frequency. In order to implement this idea, we can store the values already encountered, along with their frequencies, in a separate list. On each iteration, the i th element of the original list is compared with the values already encountered by traversing this auxiliary list. If a matching value is found, its frequency is incremented; otherwise, the current element is added to the list of distinct values seen so far with the frequency of 1.

It is not difficult to see that the worst-case input for this algorithm is a list with no equal elements. For such a list, its i th element is compared with $i - 1$ elements of the auxiliary list of distinct values seen so far before being added to the list with a frequency of 1. As a result, the worst-case number of comparisons made by this algorithm in creating the frequency list is

$$C(n) = \sum_{i=1}^n (i - 1) = 0 + 1 + \cdots + (n - 1) = \frac{(n - 1)n}{2} \in \Theta(n^2).$$

The additional $n - 1$ comparisons needed to find the largest frequency in the auxiliary list do not change the quadratic worst-case efficiency class of the algorithm.

As an alternative, let us first sort the input. Then all equal values will be adjacent to each other. To compute the mode, all we need to do is to find the longest run of adjacent equal values in the sorted array.

ALGORITHM *PresortMode*($A[0..n - 1]$)

```
//Computes the mode of an array by sorting it first
//Input: An array  $A[0..n - 1]$  of orderable elements
//Output: The array's mode
Sort the array  $A$ 
 $i \leftarrow 0$  //current run begins at position  $i$ 
 $modefrequency \leftarrow 0$  //highest frequency seen so far
while  $i \leq n - 1$  do
     $runlength \leftarrow 1$ ;  $runvalue \leftarrow A[i]$ 
    while  $i + runlength \leq n - 1$  and  $A[i + runlength] = runvalue$ 
         $runlength \leftarrow runlength + 1$ 
    if  $runlength > modefrequency$ 
         $modefrequency \leftarrow runlength$ ;  $modevalue \leftarrow runvalue$ 
     $i \leftarrow i + runlength$ 
return  $modevalue$ 
```

The analysis here is similar to the analysis of Example 1: the running time of the algorithm will be dominated by the time spent on sorting since the remainder of the algorithm takes linear time (why?). Consequently, with an $n \log n$ sort, this method's worst-case efficiency will be in a better asymptotic class than the worst case efficiency of the brute-force algorithm. ■

EXAMPLE 3 *Searching problem*

Consider the problem of searching for a given value v in a given array of n sortable items. The brute-force solution here is sequential search (Section 3.1) that needs n comparisons in the worst case. If the array is sorted first, we can then apply binary search that requires only $\lfloor \log_2 n \rfloor + 1$ comparisons in the worst case. Assuming the most efficient $n \log n$ sort, the total running time of such a searching algorithm in the worst case will be

$$T(n) = T_{\text{sort}}(n) + T_{\text{search}}(n) = \Theta(n \log n) + \Theta(\log n) = \Theta(n \log n),$$

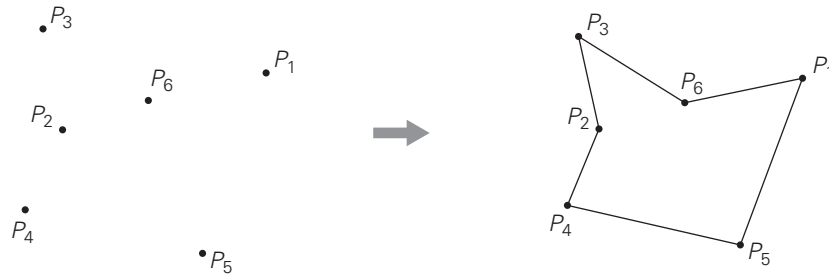
which is inferior to sequential search. The same will also be true for the average-case efficiency. Of course, if we are to search in the same list more than once, the time spent on sorting might well be justified. (Problem 6 in the exercises asks to estimate the smallest number of searches needed to justify presorting.) ■


Before we finish our discussion of presorting, we should mention that many, if not most, geometric algorithms dealing with sets of points use presorting in one way or another. Points can be sorted by one of their coordinates or by their distance from a particular line or by some angle, and so on. For example, presorting was used in the divide-and-conquer algorithms for the closest-pair problem and for the convex-hull problem, which were discussed in Section 4.6.

Exercises 6.1

1. Recall that the *median* of a list of n numbers is defined as its $\lceil n/2 \rceil$ smallest element. (The median is larger than one half the elements and is smaller than the other half.) Design a presorting-based algorithm for finding the median and determine its efficiency class.
2. Consider the problem of finding the distance between the two closest numbers in an array of n numbers. (The distance between two numbers x and y is computed as $|x - y|$.)
 - a. Design a presorting-based algorithm for solving this problem and determine its efficiency class.

- b.** Compare the efficiency of this algorithm with that of the brute-force algorithm (see Problem 9 in Exercises 1.2).
- 3.** Let $A = \{a_1, \dots, a_n\}$ and $B = \{b_1, \dots, b_m\}$ be two sets of numbers. Consider the problem of finding their intersection, i.e., the set C of all the numbers that are in both A and B .
- a.** Design a brute-force algorithm for solving this problem and determine its efficiency class.
- b.** Design a presorting-based algorithm for solving this problem and determine its efficiency class.
- 4.** Consider the problem of finding the smallest and largest elements in an array of n numbers.
- a.** Design a presorting-based algorithm for solving this problem and determine its efficiency class.
- b.** Compare the efficiency of the three algorithms: (i) the brute-force algorithm, (ii) this presorting-based algorithm, and (iii) the divide-and-conquer algorithm (see Problem 2 in Exercises 4.1).
- 5.** Show that the average-case efficiency of one-time searching by the algorithm that consists of the most efficient comparison-based sorting algorithm followed by binary search is inferior to the average-case efficiency of sequential search.
- 6.** Estimate how many searches will be needed to justify time spent on presorting an array of 10^3 elements if sorting is done by mergesort and searching is done by binary search. (You may assume that all searches are for elements known to be in the array.) What about an array of 10^6 elements?
- 7.** To sort or not to sort? Design a reasonably efficient algorithm for solving each of the following problems and determine its efficiency class.
- a.** You are given n telephone bills and m checks sent to pay the bills ($n \geq m$). Assuming that telephone numbers are written on the checks, find out who failed to pay. (For simplicity, you may also assume that only one check is written for a particular bill and that it covers the bill in full.)
- b.** You have a file of n student records indicating each student's number, name, home address, and date of birth. Find out the number of students from each of the 50 U.S. states.
- 8.** Given a set of $n \geq 3$ points in the x - y coordinate plane, connect them in a simple polygon, i.e., a closed path through all the points so that its line segments (the polygon's edges) do not intersect (except for neighboring edges at their common vertex). For example,



- a. Does the problem always have a solution? Does it always have a unique solution?
 - b. Design a reasonably efficient algorithm for solving this problem and indicate its efficiency class.
9. You have an array of n numbers and an integer s . Find out whether the array contains two elements whose sum is s . (For example, for the array 5, 9, 1, 3 and $s = 6$, the answer is yes, but for the same array and $s = 7$, the answer is no.) Design an algorithm for this problem with a better than quadratic time efficiency.
10.  a. Design an efficient algorithm for finding all sets of anagrams in a large file such as a dictionary of English words [Ben00]. For example, *eat*, *ate*, and *tea* belong to one such a set.
- b. Write a program implementing the algorithm.

6.2 Gaussian Elimination

You are certainly familiar with systems of two linear equations in two unknowns:

$$\begin{aligned} a_{11}x + a_{12}y &= b_1 \\ a_{21}x + a_{22}y &= b_2 \end{aligned}$$

Recall that unless the coefficients of one equation are proportional to the coefficients of the other, the system has a unique solution. The standard method for finding this solution is to use either equation to express one of the variables as a function of the other and then substitute the result into the other equation, yielding a linear equation whose solution is then used for finding the value of the second variable.

In many applications, we need to solve a system of n equations in n unknowns—

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

—where n is a large number. Theoretically we can solve such a system by generalizing the substitution method for solving systems of two linear equations (what general design technique would such a method be based upon?); however, the resulting algorithm would be extremely cumbersome.

Fortunately, there is a much more elegant algorithm for solving systems of linear equations called **Gaussian elimination**.² The idea of Gaussian elimination is to transform a system of n linear equations with n unknowns to an equivalent system (i.e., a system with the same solution as the original one) with an upper-triangular coefficient matrix, a matrix with all zeros below its main diagonal:

$$\begin{array}{l} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{array} \implies \begin{array}{l} a'_{11}x_1 + a'_{12}x_2 + \cdots + a'_{1n}x_n = b'_1 \\ a'_{22}x_2 + \cdots + a'_{2n}x_n = b'_2 \\ \vdots \\ a'_{nn}x_n = b'_n \end{array}$$

In matrix notations, we can write it as

$$Ax = b \implies A'x = b',$$

where

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}, \quad A' = \begin{bmatrix} a'_{11} & a'_{12} & \cdots & a'_{1n} \\ 0 & a'_{22} & \cdots & a'_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a'_{nn} \end{bmatrix}, \quad b' = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{bmatrix}.$$

(We added primes to the matrix elements and right-hand sides of the new system to stress the point that their values differ from their counterparts in the original system.)

Why is the system with the upper-triangular coefficient matrix better than a system with an arbitrary coefficient matrix? Because we can easily solve the system with an upper-triangular coefficient matrix by backward substitutions as follows. First, we can immediately find the value of x_n from the last equation; then we can substitute this value into the next to last equation to get x_{n-1} and so on until we substitute the known values of the last $n - 1$ variables into the first equation, from which we find the value of x_1 .

2. The method is named after Carl Friedrich Gauss (1777–1855), who—like other giants in the history of mathematics such as Isaac Newton and Leonhard Euler—made numerous fundamental contributions to both theoretical and computational mathematics.

So how can we get from a system with an arbitrary coefficient matrix A to an equivalent system with an upper-triangular coefficient matrix A' ? We can do it through a series of the so-called elementary operations:

- Exchanging two equations of the system
- Replacing an equation with its nonzero multiple
- Replacing an equation with a sum or difference of this equation and some multiple of another equation

Since no elementary operation can change a solution to a system, any system that is obtained through a series of such operations will have the same solution as the original one. Let us see how we can get to a system with an upper-triangular matrix. First, we use a_{11} as a pivot to make all x_1 coefficients zeros in the equations below the first one. Specifically, we replace the second equation with the difference between it and the first equation multiplied by a_{21}/a_{11} to get an equation with a zero coefficient for x_1 . Doing the same for the third, fourth, and finally n th equation—with the multiples $a_{31}/a_{11}, a_{41}/a_{11}, \dots, a_{n1}/a_{11}$ of the first equation, respectively—makes all the coefficients of x_1 below the first equation zero. Then we get rid of all the coefficients of x_2 by subtracting an appropriate multiple of the second equation from each of the equations below the second one. Repeating this elimination for each of the first $n - 1$ variables ultimately yields a system with an upper-triangular coefficient matrix.

Before we look at an example of Gaussian elimination, let us note that we can operate with just a system's coefficient matrix augmented, as its $(n + 1)$ st column, with the equations' right-hand side values. In other words, we need to write explicitly neither the variable names nor the plus and equality signs.

EXAMPLE *Solve the system by Gaussian elimination*

$$2x_1 - x_2 + x_3 = 1$$

$$4x_1 + x_2 - x_3 = 5$$

$$x_1 + x_2 + x_3 = 0$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{array}{l} \text{row 2} - \frac{4}{2} \text{ row 1} \\ \text{row 3} - \frac{1}{2} \text{ row 1} \end{array}$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & \frac{3}{2} & \frac{1}{2} & -\frac{1}{2} \end{bmatrix} \begin{array}{l} \text{row 3} - \frac{1}{2} \text{ row 2} \end{array}$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{bmatrix}$$

Now we can obtain the solution by backward substitutions:

$$x_3 = (-2)/2 = -1, x_2 = (3 - (-3)x_3)/3 = 0, \text{ and } x_1 = (1 - x_3 - (-1)x_2)/2 = 1. \quad \blacksquare$$

Here is a pseudocode for the elimination stage of Gaussian elimination.

ALGORITHM *GaussElimination*($A[1..n, 1..n]$, $b[1..n]$)

//Applies Gaussian elimination to matrix A of a system's coefficients,

//augmented with vector b of the system's right-hand side values.

//Input: Matrix $A[1..n, 1..n]$ and column-vector $b[1..n]$

//Output: An equivalent upper-triangular matrix in place of A with the

//corresponding right-hand side values in the $(n + 1)$ st column.

for $i \leftarrow 1$ **to** n **do** $A[i, n + 1] \leftarrow b[i]$ //augments the matrix

for $i \leftarrow 1$ **to** $n - 1$ **do**

for $j \leftarrow i + 1$ **to** n **do**

for $k \leftarrow i$ **to** $n + 1$ **do**

$A[j, k] \leftarrow A[j, k] - A[i, k] * A[j, i] / A[i, i]$

There are two important observations to make about this pseudocode. First, it is not always correct: if $A[i, i] = 0$, we cannot divide by it and hence cannot use the i th row as a pivot for the i th iteration of the algorithm. In such a case, we should take advantage of the first elementary operation and exchange the i th row with some row below it that has a nonzero coefficient in the i th column. (If the system has a unique solution, which is the normal case for systems under consideration, such a row must exist.)

Since we have to be prepared for the possibility of row exchanges anyway, we can take care of another potential difficulty: the possibility that $A[i, i]$ is so small and consequently the scaling factor $A[j, i]/A[i, i]$ so large that the new value of $A[j, k]$ might become distorted by a round-off error caused by a subtraction of two numbers of greatly different magnitudes.³ To avoid this problem, we can always look for a row with the largest absolute value of the coefficient in the i th column, exchange it with the i th row, and then use it as the i th iteration's pivot. This modification, called *partial pivoting*, guarantees that the magnitude of the scaling factor will never exceed 1.

The second observation is the fact that the innermost loop is written with a glaring inefficiency. Can you find it before checking the following pseudocode, which both incorporates partial pivoting and eliminates this inefficiency?

ALGORITHM *BetterGaussElimination*($A[1..n, 1..n]$, $b[1..n]$)

//Implements Gaussian elimination with partial pivoting

//Input: Matrix $A[1..n, 1..n]$ and column-vector $b[1..n]$

3. We discuss this phenomenon, called the *subtractive cancellation*, in more detail in Section 10.4.

```

//Output: An equivalent upper-triangular matrix in place of A and the
//corresponding right-hand side values in place of the (n + 1)st column.
for i ← 1 to n do A[i, n + 1] ← b[i] //appends b to A as the last column
for i ← 1 to n - 1 do
    pivotrow ← i
    for j ← i + 1 to n do
        if |A[j, i]| > |A[pivotrow, i]| pivotrow ← j
    for k ← i to n + 1 do
        swap(A[i, k], A[pivotrow, k])
    for j ← i + 1 to n do
        temp ← A[j, i] / A[i, i]
        for k ← i to n + 1 do
            A[j, k] ← A[j, k] - A[i, k] * temp

```

Let us find the time efficiency of this algorithm. Its innermost loop consists of a single line

$$A[j, k] \leftarrow A[j, k] - A[i, k] * temp$$

which contains one multiplication and one subtraction. On most computers, multiplication is unquestionably more expensive than addition/subtraction, and hence it is the multiplication count that is usually quoted as the algorithm's basic operation.⁴ The standard summation formulas and rules reviewed in Section 2.3 (see also Appendix A) are very helpful in the following derivation:

$$\begin{aligned}
 C(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \sum_{k=i}^{n+1} 1 = \sum_{i=1}^{n-1} \sum_{j=i+1}^n (n+1-i+1) = \sum_{i=1}^{n-1} \sum_{j=i+1}^n (n+2-i) \\
 &= \sum_{i=1}^{n-1} (n+2-i)(n-(i+1)+1) = \sum_{i=1}^{n-1} (n+2-i)(n-i) \\
 &= (n+1)(n-1) + n(n-2) + \cdots + 3 \cdot 1 \\
 &= \sum_{j=1}^{n-1} (j+2)j = \sum_{j=1}^{n-1} j^2 + \sum_{j=1}^{n-1} 2j = \frac{(n-1)n(2n-1)}{6} + 2 \frac{(n-1)n}{2} \\
 &= \frac{n(n-1)(2n+5)}{6} \approx \frac{1}{3}n^3 \in \Theta(n^3).
 \end{aligned}$$

4. As we mentioned in Section 2.1, on some computers, multiplication is not necessarily more expensive than addition/subtraction. For this algorithm, this point is moot since we can simply count the number of times the innermost loop is executed, which is, of course, exactly the same number as the number of multiplications and the number of subtractions there.

Since the second (backward substitution) stage of the Gaussian elimination is in $\Theta(n^2)$, as we will ask you to show in the exercises, the running time is dominated by the cubic elimination stage, making the entire algorithm cubic as well.

Theoretically, Gaussian elimination always either yields an exact solution to a system of linear equations when the system has a unique solution or discovers that no such solution exists. In the latter case, the system will have either no solutions or infinitely many of them. In practice, solving systems of significant size on a computer by this method is not nearly so straightforward as the method would lead us to believe. The principal difficulty lies in preventing an accumulation of round-off errors (see Section 10.4). Consult textbooks on numerical analysis that analyze this and other implementation issues in great detail.

***LU* Decomposition and Other Applications**

Gaussian elimination has an interesting and very useful by-product called ***LU decomposition*** of the coefficient matrix. In fact, modern commercial implementations of Gaussian elimination are based on such a decomposition rather than on the basic algorithm outlined earlier.

EXAMPLE Let us return to the example at the beginning of this section, where we applied Gaussian elimination to the matrix

$$A = \begin{bmatrix} 2 & -1 & 1 \\ 4 & 1 & -1 \\ 1 & 1 & 1 \end{bmatrix}.$$

Consider the lower-triangular matrix L made up of 1's on its main diagonal and the row multiples used in the Gaussian elimination process

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{bmatrix}$$

and the upper-triangular matrix U that was the result of this elimination

$$U = \begin{bmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{bmatrix}.$$

It turns out that the product LU of these matrices is equal to matrix A . (For this particular pair of L and U , you can verify this fact by direct multiplication, but as a general proposition, it needs, of course, a proof, which we will omit.)

Therefore, solving the system $Ax = b$ is equivalent to solving the system $LUx = b$. The latter system can be solved as follows. Denote $y = Ux$, then $Ly = b$. Solve the system $Ly = b$ first, which is easy to do because L is a lower-triangular matrix; then solve the system $Ux = y$, with the upper-triangular matrix

U , to find x . Thus, for the system at the beginning of this section, we first solve $Ly = b$:

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 5 \\ 0 \end{bmatrix}.$$

Its solution is

$$y_1 = 1, \quad y_2 = 5 - 2y_1 = 3, \quad y_3 = 0 - \frac{1}{2}y_1 - \frac{1}{2}y_2 = -2.$$

Solving $Ux = y$ means solving

$$\begin{bmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ -2 \end{bmatrix},$$

and the solution is

$$x_3 = (-2)/2 = -1, \quad x_2 = (3 - (-3)x_3)/3 = 0, \quad x_1 = (1 - x_3 - (-1)x_2)/2 = 1. \quad \blacksquare$$

Note that once we have the LU decomposition of matrix A , we can solve systems $Ax = b$ with as many right-hand side vectors b as we want to, one at a time. This is a distinct advantage over the classic Gaussian elimination discussed earlier. Also note that the LU decomposition does not actually require extra memory because we can store the nonzero part of U in the upper-triangular part of A (including the main diagonal) and store the nontrivial part of L below the main diagonal of A .

Computing a Matrix Inverse

Gaussian elimination is a very useful algorithm that tackles one of the most important problems of applied mathematics: solving systems of linear equations. In fact, Gaussian elimination can also be applied to several other problems of linear algebra, such as computing a matrix **inverse**. The inverse of an n -by- n matrix A is an n -by- n matrix, denoted A^{-1} , such that

$$AA^{-1} = I,$$

where I is the n -by- n identity matrix (the matrix with all zero elements except the main diagonal elements, which are all ones). Not every square matrix has an inverse, but when it exists, the inverse is unique. If a matrix A does not have an inverse, it is called **singular**. One can prove that a matrix is singular if and only if one of its rows is a linear combination (a sum of some multiples) of the other rows. A convenient way to check whether a matrix is nonsingular is to apply Gaussian elimination: if it yields an upper-triangular matrix with no zeros on the main diagonal, the matrix is nonsingular; otherwise, it is singular. So being singular is a very special situation, and most square matrices do have their inverses.

Theoretically, inverse matrices are very important because they play the role of reciprocals in matrix algebra, overcoming the absence of the explicit division operation for matrices. For example, in a complete analogy with a linear equation in one unknown $ax = b$ whose solution can be written as $x = a^{-1}b$ (if a is not zero), we can express a solution to a system of n equations in n unknowns $Ax = b$ as $x = A^{-1}b$ (if A is nonsingular) where b is, of course, a vector, not a number.

According to the definition of the inverse matrix for a nonsingular n -by- n matrix A , to find it, we need to find n^2 numbers x_{ij} , $1 \leq i, j \leq n$, such that

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & & & \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & & & \\ 0 & 0 & \cdots & 1 \end{bmatrix}.$$

We can find the unknowns by solving n systems of linear equations that have the same coefficient matrix A , the vector of unknowns x^j is the j th column of the inverse, and the right-hand side vector e^j is the j th column of the identity matrix ($1 \leq j \leq n$):

$$Ax^j = e^j.$$

We can solve these systems by applying Gaussian elimination to matrix A augmented by the n -by- n identity matrix. Better yet, we can use Gaussian elimination to find the LU decomposition of A and then solve the systems $LUx^j = e^j$, $j = 1, \dots, n$, as explained earlier.

Computing a Determinant

Another problem that can be solved by Gaussian elimination is computing a determinant. The **determinant** of an n -by- n matrix A , denoted $\det A$ or $|A|$, is a number whose value can be defined recursively as follows. If $n = 1$, i.e., if A consists of a single element a_{11} , $\det A$ is equal to a_{11} ; for $n > 1$, $\det A$ is computed by the recursive formula

$$\det A = \sum_{j=1}^n s_j a_{1j} \det A_j,$$

where s_j is $+1$ if j is odd and -1 if j is even, a_{1j} is the element in row 1 and column j , and A_j is the $n - 1$ -by- $n - 1$ matrix obtained from matrix A by deleting its row 1 and column j .

In particular, for a 2-by-2 matrix, the definition implies a formula that is easy to remember:

$$\det \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = a_{11} \det [a_{22}] - a_{12} \det [a_{21}] = a_{11}a_{22} - a_{12}a_{21}.$$

In other words, the determinant of a 2-by-2 matrix is simply equal to the difference between the products of its diagonal elements.

For a 3-by-3 matrix, we get

$$\begin{aligned} \det \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \\ = a_{11} \det \begin{bmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{bmatrix} - a_{12} \det \begin{bmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{bmatrix} + a_{13} \det \begin{bmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \\ = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{21}a_{32}a_{13} - a_{31}a_{22}a_{13} - a_{21}a_{12}a_{33} - a_{32}a_{23}a_{11}. \end{aligned}$$

Incidentally, this formula is very handy in a variety of applications. In particular, we used it twice already in Section 4.6 as a part of the quickhull algorithm.

But what if we need to compute a determinant of a large matrix? (Although this is a task that is rarely needed in practice, it is worth discussing nevertheless.) Using the recursive definition can be of little help because it implies computing the sum of $n!$ terms. Here, Gaussian elimination comes to the rescue again. The central point is the fact that the determinant of an upper-triangular matrix is equal to the product of elements on its main diagonal, and it is easy to see how elementary operations employed by the algorithm influence the determinant's value. (Basically, it either remains unchanged or changes a sign or is multiplied by the constant used by the elimination algorithm.) As a result, we can compute the determinant of an n -by- n matrix in cubic time.

Determinants play an important role in the theory of systems of linear equations. Specifically, a system of n linear equations in n unknowns $Ax = b$ has a unique solution if and only if the determinant of its coefficient matrix, $\det A$, is not equal to zero. Moreover, this solution can be found by the formulas called **Cramer's rule**:

$$x_1 = \frac{\det A_1}{\det A}, \dots, x_j = \frac{\det A_j}{\det A}, \dots, x_n = \frac{\det A_n}{\det A},$$

where $\det A_j$ is the determinant of the matrix obtained by replacing the j th column of A by the column b . (We will ask you to investigate whether using Cramer's rule is a good algorithm for solving systems of linear equations in the exercises.)

Exercises 6.2

1. Solve the following system by Gaussian elimination.

$$\begin{aligned} x_1 + x_2 + x_3 &= 2 \\ 2x_1 + x_2 + x_3 &= 3 \\ x_1 - x_2 + 3x_3 &= 8 \end{aligned}$$

2. **a.** Solve the system of the previous question by the LU decomposition method.
- b.** From the standpoint of general algorithm design techniques, how would you classify the LU decomposition method?
3. Solve the system of Problem 1 by computing the inverse of its coefficient matrix and then multiplying it by the right-hand side vector.
4. Would it be correct to get the efficiency class of the elimination stage of Gaussian elimination as follows?

$$\begin{aligned}
 C(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \sum_{k=i}^{n+1} 1 = \sum_{i=1}^{n-1} (n+2-i)(n-i) \\
 &= \sum_{i=1}^{n-1} [(n+2)n - i(2n+2) + i^2] \\
 &= \sum_{i=1}^{n-1} (n+2)n - \sum_{i=1}^{n-1} (2n+2)i + \sum_{i=1}^{n-1} i^2.
 \end{aligned}$$

Since $s_1(n) = \sum_{i=1}^{n-1} (n+2)n \in \Theta(n^3)$, $s_2(n) = \sum_{i=1}^{n-1} (2n+2)i \in \Theta(n^3)$, and $s_3(n) = \sum_{i=1}^{n-1} i^2 \in \Theta(n^3)$, $s_1(n) - s_2(n) + s_3(n) \in \Theta(n^3)$.

5. Write a pseudocode for the back-substitution stage of Gaussian elimination and show that its running time is in $\Theta(n^2)$.
6. Assuming that division of two numbers takes three times longer than their multiplication, estimate how much faster *BetterGaussElimination* is than *GaussElimination*. (Of course, you should also assume that a compiler is not going to eliminate the inefficiency in *GaussElimination*.)
7. **a.** Give an example of a system of two linear equations in two unknowns that has a unique solution and solve it by Gaussian elimination.
- b.** Give an example of a system of two linear equations in two unknowns that has no solution and apply Gaussian elimination to it.
- c.** Give an example of a system of two linear equations in two unknowns that has infinitely many solutions and apply Gaussian elimination to it.
8. The *Gauss-Jordan elimination* method differs from Gaussian elimination in that the elements above the main diagonal of the coefficient matrix are made zero at the same time and by the same use of a pivot row as the elements below the main diagonal.
 - a.** Apply the Gauss-Jordan method to the system of Problem 1 of these exercises.
 - b.** What general design strategy is this algorithm based on?

- c. In general, how many multiplications are made by this method while solving a system of n equations in n unknowns? How does this compare with the number of multiplications made by the Gaussian elimination method in both its elimination and back-substitution stages?
9. A system $Ax = b$ of n linear equations in n unknowns has a unique solution if and only if $\det A \neq 0$. Is it a good idea to check this condition before applying Gaussian elimination to a system?
10. a. Apply Cramer's rule to solve the system of Problem 1 of these exercises.
 b. Estimate how many times longer it will take to solve a system of n linear equations in n unknowns by Cramer's rule than by Gaussian elimination. (Assume that all the determinants in Cramer's rule formulas are computed independently by Gaussian elimination.)

6.3 Balanced Search Trees

In Sections 1.4 and 4.4, we discussed the binary search tree—one of the principal data structures for implementing dictionaries. It is a binary tree whose nodes contain elements of a set of orderable items, one element per node, so that all elements in the left subtree are smaller than the element in the subtree's root and all the elements in the right subtree are greater than it. Note that this transformation from a set to a binary search tree is an example of the representation change technique. What do we gain by such transformation compared to the straightforward implementation of a dictionary by, say, an array? We gain in the time efficiency of searching, insertion, and deletion, which are all in $\Theta(\log n)$, but only in the average case. In the worst case, these operations are in $\Theta(n)$ because the tree can degenerate into a severely unbalanced one with height equal to $n - 1$.

Computer scientists have expended a lot of effort in trying to find a structure that preserves the good properties of the classical binary search tree—principally, the logarithmic efficiency of the dictionary operations and having the set's elements sorted—but avoids its worst-case degeneracy. They have come up with two approaches.

- The first approach is of the instance simplification variety: an unbalanced binary search tree is transformed to a balanced one. Specific implementations of this idea differ by their definition of balance. An **AVL tree** requires the difference between the heights of the left and right subtrees of every node never exceed 1. A **red-black tree** tolerates the height of one subtree being twice as large as the other subtree of the same node. If an insertion or deletion of a new node creates a tree with a violated balance requirement, the tree is restructured by one of a family of special transformations called **rotations** that restore the balance required. (In this section, we discuss only AVL trees.

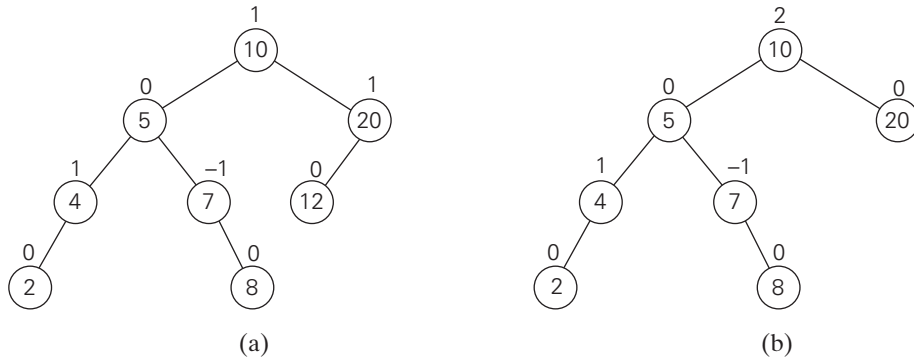


FIGURE 6.2 (a) AVL tree. (b) Binary search tree that is not an AVL tree. The numbers above the nodes indicate the nodes' balance factors.

Information about other types of binary search trees that utilize the idea of rebalancing via rotations, including red-black trees and so-called *splay trees*, can be found in the references.)

- The second approach is of the representation change variety: allow more than one element in a node of a search tree. Specific cases of such trees are **2-3 trees**, **2-3-4 trees**, and more general and important **B-trees**. They differ in the number of elements admissible in a single node of a search tree, but all are perfectly balanced. (We discuss the simplest of such trees, the 2-3 tree, in this section, leaving the discussion of *B-trees* for Chapter 7.)

AVL Trees

AVL trees were invented in 1962 by two Russian scientists, G. M. Adelson-Velsky and E. M. Landis [AVL62], after whom this data structure is named.

DEFINITION An *AVL tree* is a binary search tree in which the *balance factor* of every node, which is defined as the difference between the heights of the node's left and right subtrees, is either 0 or +1 or -1. (The height of the empty tree is defined as -1.)

For example, the binary search tree in Figure 6.2a is an AVL tree but the one in Figure 6.2b is not.

If an insertion of a new node makes an AVL tree unbalanced, we transform the tree by a rotation. A *rotation* in an AVL tree is a local transformation of its subtree rooted at a node whose balance has become either +2 or -2; if there are several such nodes, we rotate the tree rooted at the unbalanced node that is the closest to the newly inserted leaf. There are only four types of rotations; in fact, two of them are mirror images of the other two. In their simplest form, the four rotations are shown in Figure 6.3.

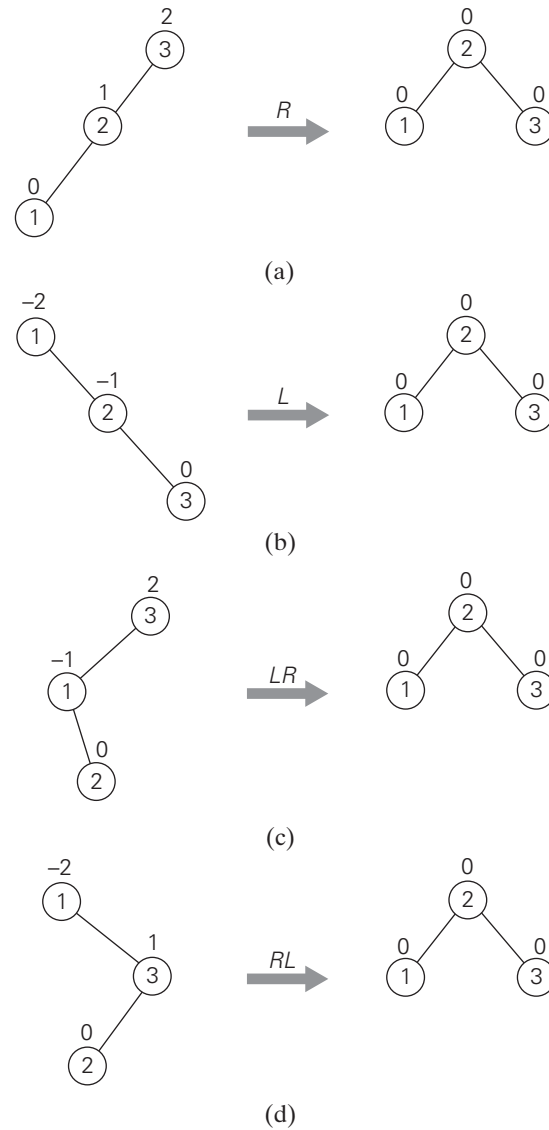


FIGURE 6.3 Four rotation types for AVL trees with three nodes. (a) Single *R*-rotation. (b) Single *L*-rotation. (c) Double *LR*-rotation. (d) Double *RL*-rotation.

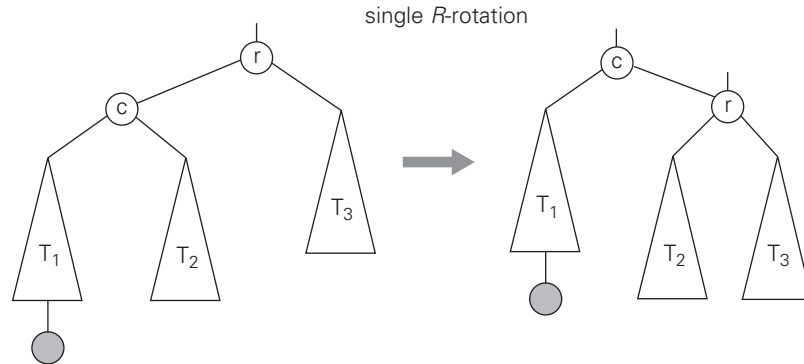


FIGURE 6.4 General form of the *R*-rotation in the AVL tree. A shaded node is the last one inserted.

The first rotation type is called the **single right rotation** or ***R*-rotation**. (Imagine rotating the edge connecting the root and its left child in the binary tree of Figure 6.3a to the right.) Figure 6.4 presents the single *R*-rotation in its most general form. Note that this rotation is performed after a new key is inserted into the left subtree of the left child of a tree whose root had the balance of +1 before the insertion.

The symmetric **single left rotation** or ***L*-rotation** is the mirror image of the single *R*-rotation. It is performed after a new key is inserted into the right subtree of the right child of a tree whose root had the balance of -1 before the insertion. (We will ask you to draw a diagram of the general case of the single *L*-rotation in the exercises.)

The second rotation type is called the **double left-right rotation** (***LR*-rotation**). It is, in fact, a combination of two rotations: we perform the *L*-rotation of the left subtree of root *r* followed by the *R*-rotation of the new tree rooted at *r* (Figure 6.5). It is performed after a new key is inserted into the right subtree of the left child of a tree whose root had the balance of +1 before the insertion.

The **double right-left rotation** (***RL*-rotation**) is the mirror image of the double *LR*-rotation and is left for the exercises.

Note that the rotations are not trivial transformations, though, fortunately, they can be done in a constant time. Not only should they guarantee that a resulting tree is balanced, but they should also preserve the basic requirements of a binary search tree. For example, in the initial tree of Figure 6.4, all the keys of subtree T_1 are smaller than *c*, which is smaller than all the keys of subtree T_2 , which are smaller than *r*, which is smaller than all the keys of subtree T_3 . And the same relationships among the key values hold, as they must, for the balanced tree after the rotation.

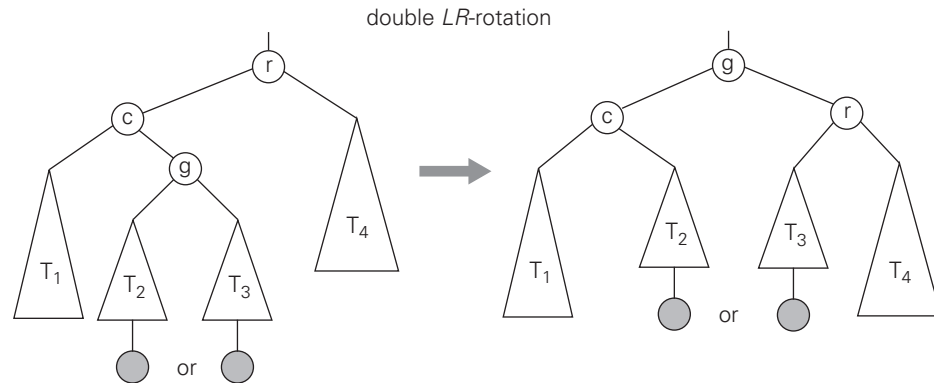


FIGURE 6.5 General form of the double *LR*-rotation in the AVL tree. A shaded node is the last one inserted. It can be either in the left subtree or in the right subtree of the root's grandchild.

An example of constructing an AVL tree for a given list of numbers is shown in Figure 6.6. As you trace the algorithm's operations, keep in mind that if there are several nodes with the ± 2 balance, the rotation is done for the tree rooted at the unbalanced node that is the closest to the newly inserted leaf.

How efficient are AVL trees? As with any search tree, the critical characteristic is the tree's height. It turns out that it is bounded both above and below by logarithmic functions. Specifically, the height h of any AVL tree with n nodes satisfies the inequalities

$$\lfloor \log_2 n \rfloor \leq h < 1.4405 \log_2(n + 2) - 1.3277.$$

(These weird-looking constants are roundoffs of some irrational numbers related to Fibonacci numbers and the golden ratio—see Section 2.5.)

The inequalities immediately imply that the operations of searching and insertion are $\Theta(\log n)$ in the worst case. Getting an exact formula for the average height of an AVL tree constructed for random lists of keys has proved to be difficult, but it is known from extensive experiments that it is about $1.01 \log_2 n + 0.1$ except when n is small ([KnuIII], p. 468). Thus, searching in an AVL tree requires, on average, almost the same number of comparisons as searching in a sorted array by binary search. The operation of key deletion in an AVL tree is considerably more difficult than insertion, but fortunately it turns out to be in the same efficiency class as insertion, i.e., logarithmic.

These impressive efficiency characteristics come at a price, however. The drawbacks of AVL trees are frequent rotations, the need to maintain balances for the tree's nodes, and overall complexity, especially of the deletion operation.

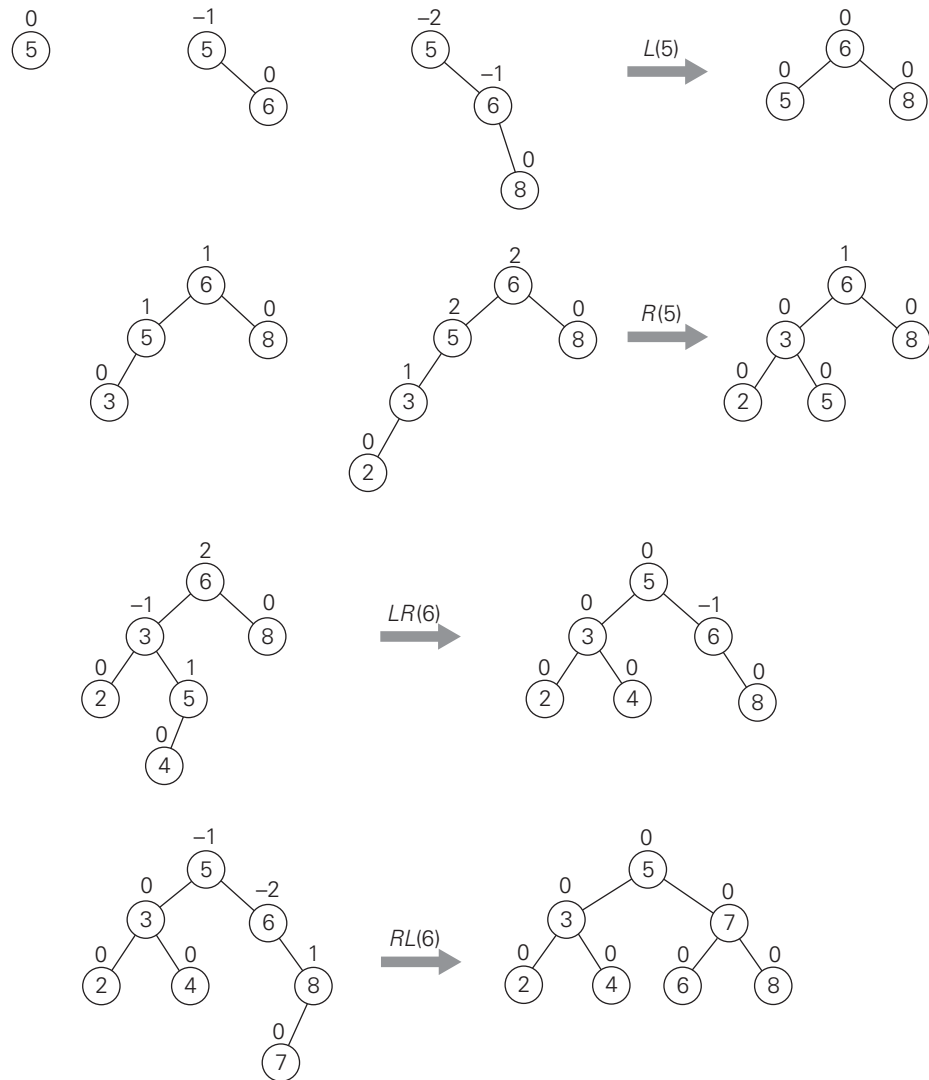


FIGURE 6.6 Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7 by successive insertions. The parenthesized number of a rotation's abbreviation indicates the root of the tree being reorganized.

These drawbacks have prevented AVL trees from becoming the standard structure for implementing dictionaries. At the same time, their underlying idea—that of rebalancing a binary search tree via rotations—has proved to be very fruitful and has led to discoveries of other interesting variations of the classical binary search tree.

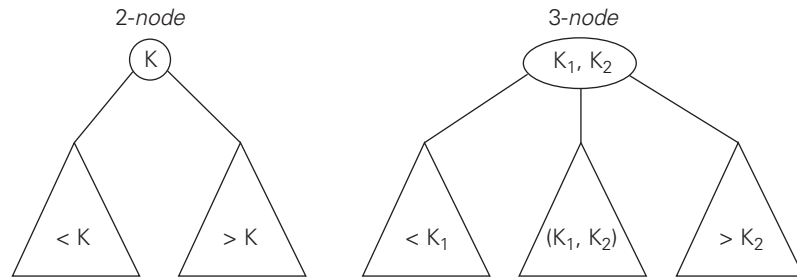


FIGURE 6.7 Two kinds of nodes of a 2-3 tree

2-3 Trees

As we mentioned at the beginning of this section, the second idea of balancing a search tree is to allow more than one key in the same node. The simplest implementation of this idea is 2-3 trees, introduced by the U.S. computer scientist John Hopcroft in 1970 (see [AHU74]). A **2-3 tree** is a tree that can have nodes of two kinds: 2-nodes and 3-nodes. A **2-node** contains a single key K and has two children: the left child serves as the root of a subtree whose keys are less than K and the right child serves as the root of a subtree whose keys are greater than K . (In other words, a 2-node is the same kind of node we have in the classical binary search tree.) A **3-node** contains two ordered keys K_1 and K_2 ($K_1 < K_2$) and has three children. The leftmost child serves as the root of a subtree with keys less than K_1 , the middle child serves as the root of a subtree with keys between K_1 and K_2 , and the rightmost child serves as the root of a subtree with keys greater than K_2 (Figure 6.7).

The last requirement of the 2-3 tree is that all its leaves must be on the same level, i.e., a 2-3 tree is always **height-balanced**: the length of a path from the root of the tree to a leaf must be the same for every leaf. It is this property that we “buy” by allowing more than a single key in a search tree’s node.

Searching for a given key K in a 2-3 tree is quite straightforward. We start with the root. If the root is a 2-node, we act as if it were a binary search tree: we either stop if K is equal to the root’s key or continue the search in the left or right subtree if K is, respectively, smaller or larger than the root’s key. If the root is a 3-node, we know after no more than two key comparisons whether the search can be stopped (if K is equal to one of the root’s keys) or in which of the root’s three subtrees it needs to be continued.

Inserting a new key in a 2-3 tree is done as follows. First of all, we always insert a new key K at a leaf except for the empty tree. The appropriate leaf is found by performing a search for K . If the leaf in question is a 2-node, we insert K there as either the first or the second key, depending on whether K is smaller or larger than the node’s old key. If the leaf is a 3-node, we split the leaf in two: the smallest

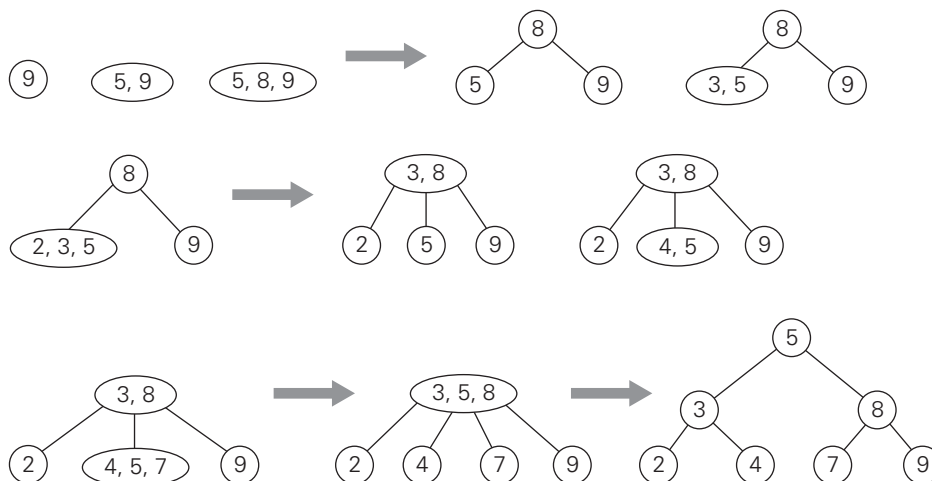


FIGURE 6.8 Construction of a 2-3 tree for the list 9, 5, 8, 3, 2, 4, 7.

of the three keys (two old ones and the new key) is put in the first leaf, the largest key is put in the second leaf, while the middle key is promoted to the old leaf's parent. (If the leaf happens to be the tree's root, a new root is created to accept the middle key.) Note that promotion of a middle key to its parent can cause the parent's overflow (if it was a 3-node) and hence can lead to several node splits along the chain of the leaf's ancestors.

An example of a 2-3 tree construction is given in Figure 6.8.

As for any search tree, the efficiency of the dictionary operations depends on the tree's height. So let us first find an upper bound for it. A 2-3 tree of height h with the smallest number of keys is a full tree of 2-nodes (such as the final tree in Figure 6.8 for $h = 2$). Therefore, for any 2-3 tree of height h with n nodes, we get the inequality

$$n \geq 1 + 2 + \dots + 2^h = 2^{h+1} - 1,$$

and hence

$$h \leq \log_2(n + 1) - 1.$$

On the other hand, a 2-3 tree of height h with the largest number of keys is a full tree of 3-nodes, each with two keys and three children. Therefore, for any 2-3 tree with n nodes,

$$n \leq 2 \cdot 1 + 2 \cdot 3 + \dots + 2 \cdot 3^h = 2(1 + 3 + \dots + 3^h) = 3^{h+1} - 1,$$

and hence

$$h \geq \log_3(n + 1) - 1.$$

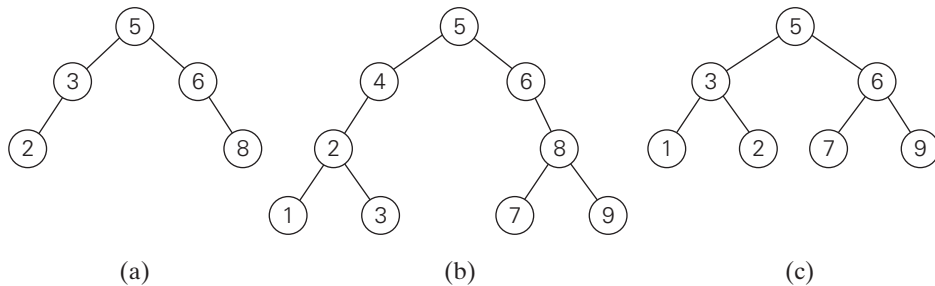
These lower and upper bounds on height h ,

$$\log_3(n + 1) - 1 \leq h \leq \log_2(n + 1) - 1,$$

imply that the time efficiencies of searching, insertion, and deletion are all in $\Theta(\log n)$ in both the worst and average case. We will consider a very important generalization of 2-3 trees, called B -trees, in Section 7.4.

Exercises 6.3

1. Which of the following binary trees are AVL trees?



2. **a.** For $n = 1, 2, 3, 4$, and 5 , draw all the binary trees with n nodes that satisfy the balance requirement of AVL trees.
b. Draw a binary tree of height 4 that can be an AVL tree and has the smallest number of nodes among all such trees.
3. Draw diagrams of the single L -rotation and of the double RL -rotation in their general form.
4. For each of the following lists, construct an AVL tree by inserting their elements successively, starting with the empty tree:
a. 1, 2, 3, 4, 5, 6
b. 6, 5, 4, 3, 2, 1
c. 3, 6, 5, 1, 2, 4
5. **a.** For an AVL tree containing real numbers, design an algorithm for computing the range (i.e., the difference between the largest and smallest numbers) in the tree and determine its worst-case efficiency.
b. True or false: The smallest and the largest keys in an AVL tree can always be found on either the last level or the next-to-last level.
6. Write a program for constructing an AVL tree for a given list of n distinct integers.

7. **a.** Construct a 2-3 tree for the list C, O, M, P, U, T, I, N, G. (Use the alphabetical order of the letters and insert them successively starting with the empty tree.)
- b.** Assuming that the probabilities of searching for each of the keys (i.e., the letters) are the same, find the largest number and the average number of key comparisons for successful searches in this tree.
8. Let T_B and T_{2-3} be, respectively, a classical binary search tree and a 2-3 tree constructed for the same list of keys inserted in the corresponding trees in the same order. True or false: Searching for the same key in T_{2-3} always takes fewer or the same number of key comparisons as searching in T_B .
9. For a 2-3 tree containing real numbers, design an algorithm for computing the range (i.e., the difference between the largest and smallest numbers) in the tree and determine its worst-case efficiency.
10. Write a program for constructing a 2-3 tree for a given list of n integers.

6.4 Heaps and Heapsort

The data structure called the “heap” is definitely not a disordered pile of items as the word’s definition in a standard dictionary might suggest. Rather it is a clever, partially ordered data structure that is especially suitable for implementing priority queues. Recall that a *priority queue* is a set of items with an orderable characteristic called an item’s *priority*, with the following operations:

- Finding an item with the highest (i.e., largest) priority
- Deleting an item with the highest priority
- Adding a new item to the set

It is primarily an efficient implementation of these operations that makes the heap both interesting and useful. The heap is also the data structure that serves as a cornerstone of a theoretically important sorting algorithm called heapsort. We discuss this algorithm after we define the heap and investigate its basic properties.

Notion of the Heap

DEFINITION A *heap* can be defined as a binary tree with keys assigned to its nodes (one key per node) provided the following two conditions are met:

1. The tree’s shape requirement—The binary tree is *essentially complete* (or simply *complete*), that is, all its levels are full except possibly the last level, where only some rightmost leaves may be missing.

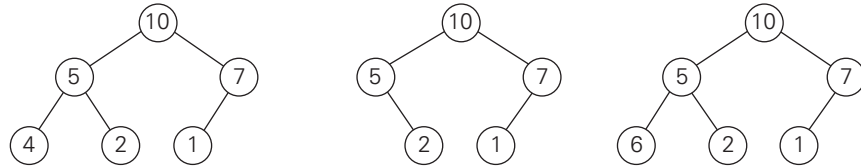


FIGURE 6.9 Illustration of the definition of “heap”: only the leftmost tree is a heap.

2. The parental dominance requirement—The key at each node is greater than or equal to the keys at its children. (This condition is considered automatically satisfied for all leaves.)⁵

For example, consider the trees of Figure 6.9. The first tree there is a heap, the second one is not a heap because the tree’s shape requirement is violated, and the third one is not a heap because the parental dominance requirement fails for the node with key 5.

Note that key values in a heap are ordered top down; that is, a sequence of values on any path from the root to a leaf is decreasing (nonincreasing, if equal keys are allowed). However, there is no left-to-right order in key values; that is, there is no relationship among key values for nodes either on the same level of the tree or, more generally, in the left and right subtrees of the same node.

Here is a list of important properties of heaps, which are not difficult to prove (check these properties for the heap of Figure 6.10, as an example).

1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\lfloor \log_2 n \rfloor$.
2. The root of a heap always contains its largest element.
3. A node of a heap considered with all its descendants is also a heap.
4. A heap can be implemented as an array by recording its elements in the top-down, left-to-right fashion. It is convenient to store the heap’s elements in positions 1 through n of such an array, leaving $H[0]$ either unused or putting there a sentinel whose value is greater than every element in the heap. In such a representation,
 - a. the parental node keys will be in the first $\lfloor n/2 \rfloor$ positions of the array, while the leaf keys will occupy the last $\lceil n/2 \rceil$ positions.
 - b. the children of a key in the array’s parental position i ($1 \leq i \leq \lfloor n/2 \rfloor$) will be in positions $2i$ and $2i + 1$, and, correspondingly, the parent of a key in position i ($2 \leq i \leq n$) will be in position $\lfloor i/2 \rfloor$.

5. Some authors require the key at each node to be *less* than or equal to the keys at its children. We call this variation a *min-heap*.

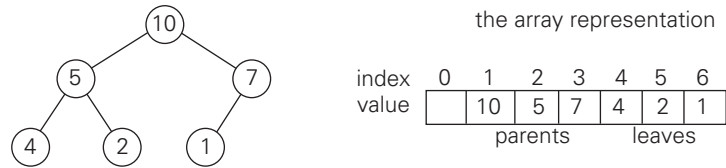


FIGURE 6.10 Heap and its array representation

As an example, you can check these properties for the heap of Figure 6.10.

Thus, we could also define a heap as an array $H[1..n]$ in which every element in position i in the first half of the array is greater than or equal to the elements in positions $2i$ and $2i + 1$, i.e.,

$$H[i] \geq \max\{H[2i], H[2i + 1]\} \quad \text{for } i = 1, \dots, \lfloor n/2 \rfloor.$$

(Of course, if $2i + 1 > n$, just $H[i] \geq H[2i]$ needs to be satisfied.) While the ideas behind the majority of algorithms dealing with heaps are easier to understand if we think of heaps as binary trees, their actual implementations are usually much simpler and more efficient with arrays.

How can we construct a heap for a given list of keys? There are two principal alternatives for doing that. The first is the so-called **bottom-up heap construction** algorithm (illustrated in Figure 6.11). It initializes the essentially complete binary tree with n nodes by placing keys in the order given and then “heapifies” the tree as follows. Starting with the last parental node and ending with the root, the algorithm checks whether the parental dominance holds for the key at this node. If it does not, the algorithm exchanges the node’s key K with the larger key of its children and checks whether the parental dominance holds for K in its new position. This process continues until the parental dominance requirement for K is satisfied. (Eventually it has to because it holds automatically for any key in a leaf.) After completing the “heapification” of the subtree rooted at the current parental node, the algorithm proceeds to do the same for the node’s immediate predecessor. The algorithm stops after this is done for the tree’s root.

Before we introduce a pseudocode of the bottom-up heap construction algorithm, we should make one more remark. Since the value of a node’s key does not change during the process of sifting it down the tree, there is no need to involve it in intermediate swaps. You can think of this improvement as swapping the empty node with larger keys in its children until a final position is reached where it accepts the “erased” value again.

ALGORITHM *HeapBottomUp*($H[1..n]$)

```
//Constructs a heap from the elements of a given array
// by the bottom-up algorithm
//Input: An array  $H[1..n]$  of orderable items
```

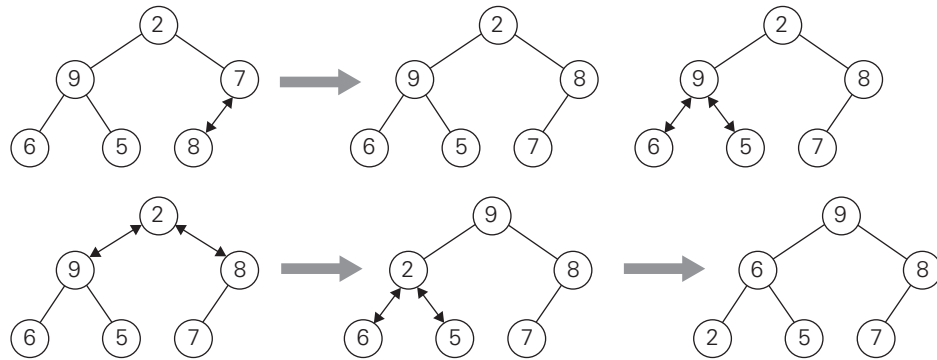


FIGURE 6.11 Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8

```

//Output: A heap  $H[1..n]$ 
for  $i \leftarrow \lfloor n/2 \rfloor$  downto 1 do
   $k \leftarrow i$ ;  $v \leftarrow H[k]$ 
   $heap \leftarrow \text{false}$ 
  while not  $heap$  and  $2 * k \leq n$  do
     $j \leftarrow 2 * k$ 
    if  $j < n$  //there are two children
      if  $H[j] < H[j + 1]$   $j \leftarrow j + 1$ 
    if  $v \geq H[j]$ 
       $heap \leftarrow \text{true}$ 
    else  $H[k] \leftarrow H[j]$ ;  $k \leftarrow j$ 
   $H[k] \leftarrow v$ 

```

How efficient is this algorithm in the worst case? Assume, for simplicity, that $n = 2^k - 1$ so that a heap's tree is full, i.e., the largest number of nodes occur on each level. Let h be the height of the tree; according to the first property of heaps in the list at the beginning of the section, $h = \lfloor \log_2 n \rfloor$ (or just $\lceil \log_2(n + 1) \rceil - 1 = k - 1$ for the specific values of n we are considering). Each key on level i of the tree will travel to the leaf level h in the worst case of the heap construction algorithm. Since moving to the next level down requires two comparisons—one to find the larger child and the other to determine whether the exchange is required—the total number of key comparisons involving a key on level i will be $2(h - i)$. So the total number of key comparisons in the worst case will be

$$C_{\text{worst}}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h - i) = \sum_{i=0}^{h-1} 2(h - i)2^i = 2(n - \log_2(n + 1)),$$

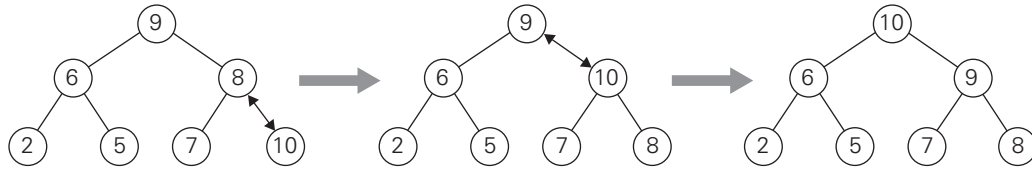


FIGURE 6.12 Inserting a key (10) into the heap constructed in Figure 6.11. The new key is sifted up via a swap with its parent until it is not larger than its parent (or is in the root).

where the validity of the last equality can be proved either by using the closed-form formula for the sum $\sum_{i=1}^h i2^i$ (see Appendix A) or by mathematical induction in h . Thus, with this bottom-up algorithm, a heap of size n can be constructed with fewer than $2n$ comparisons.

The alternative (and less efficient) algorithm constructs a heap by successive insertions of a new key into a previously constructed heap; some people call it the *top-down heap construction* algorithm. So how can we insert a new key K into a heap? First, attach a new node with key K in it after the last leaf of the existing heap. Then sift K up to its appropriate place in the new heap as follows. Compare K with its parent's key: if the latter is greater than or equal to K , stop (the structure is a heap); otherwise, swap these two keys and compare K with its new parent. This swapping continues until K is not greater than its last parent or it reaches the root (illustrated in Figure 6.12). In this algorithm, too, we can sift up an empty node until it reaches its proper position, where it will get K 's value.

Obviously, this insertion operation cannot require more key comparisons than the heap's height. Since the height of a heap with n nodes is about $\log_2 n$, the time efficiency of insertion is in $O(\log n)$.

How can we delete an item from a heap? We consider here only the most important case of deleting the root's key, leaving the question about deleting an arbitrary key in a heap for the exercises. (Authors of textbooks like to do such things to their readers, do they not?) So deleting the root's key from a heap can be done with the following algorithm (illustrated in Figure 6.13).

Step 1 Exchange the root's key with the last key K of the heap.

Step 2 Decrease the heap's size by 1.

Step 3 "Heapify" the smaller tree by sifting K down the tree exactly in the same way we did it in the bottom-up heap construction algorithm. That is, verify the parental dominance for K : if it holds, we are done; if not, swap K with the larger of its children and repeat this operation until the parental dominance condition holds for K in its new position.

The efficiency of deletion is determined by the number of key comparisons needed to "heapify" the tree after the swap has been made and the size of the tree is decreased by 1. Since it cannot require more key comparisons than twice the heap's height, the time efficiency of deletion is in $O(\log n)$ as well.

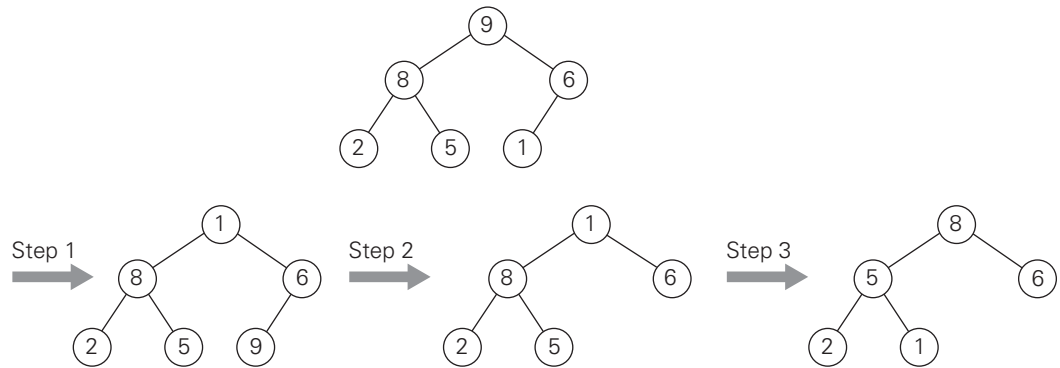


FIGURE 6.13 Deleting the root's key from a heap. The key to be deleted is swapped with the last key, after which the smaller tree is "heapified" by exchanging the new key at its root with the larger key at its children until the parental dominance requirement is satisfied.

Heapsort

Now we can describe *heapsort*—an interesting sorting algorithm discovered by J. W. J. Williams [Wil64]. This is a two-stage algorithm that works as follows.

Stage 1 (heap construction): Construct a heap for a given array.

Stage 2 (maximum deletions): Apply the root-deletion operation $n - 1$ times to the remaining heap.

As a result, the array elements are eliminated in decreasing order. But since under the array implementation of heaps, an element being deleted is placed last, the resulting array will be exactly the original array sorted in ascending order. Heapsort is traced on a specific input in Figure 6.14. (The same input as the one of Figure 6.11 is intentionally used so that you can compare the tree and array implementations of the bottom-up heap construction algorithm.)

Since we already know that the heap construction stage of the algorithm is in $O(n)$, we need to investigate just the time efficiency of the second stage. For the number of key comparisons, $C(n)$, needed for eliminating the root keys from the heaps of diminishing sizes from n to 2, we get the following inequality:

$$\begin{aligned}
 C(n) &\leq 2\lceil \log_2(n-1) \rceil + 2\lceil \log_2(n-2) \rceil + \cdots + 2\lceil \log_2 1 \rceil \leq 2 \sum_{i=1}^{n-1} \log_2 i \\
 &\leq 2 \sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1) \log_2(n-1) \leq 2n \log_2 n.
 \end{aligned}$$

This means that $C(n) \in O(n \log n)$ for the second stage of heapsort. For both stages, we get $O(n) + O(n \log n) = O(n \log n)$. A more detailed analysis shows that, in

Stage 1 (heap construction)	Stage 2 (maximum deletions)
2 9 7 6 5 8	9 6 8 2 5 7
2 9 8 6 5 7	7 6 8 2 5 19
2 9 8 6 5 7	8 6 7 2 5
9 2 8 6 5 7	5 6 7 2 18
9 6 8 2 5 7	7 6 5 2
	2 6 5 17
	6 2 5
	5 2 16
	5 2
	2 15
	2

FIGURE 6.14 Sorting the array 2, 9, 7, 6, 5, 8 by heapsort


fact, the time efficiency of heapsort is in $\Theta(n \log n)$ in both the worst and average cases. Thus, heapsort's time efficiency falls in the same class as that of mergesort; and, unlike the latter, it is in place, i.e., it does not require any extra storage. Timing experiments on random files show that heapsort runs more slowly than quicksort but is competitive with mergesort.

Exercises 6.4

- Construct a heap for the list 1, 8, 6, 5, 3, 7, 4 by the bottom-up algorithm.
 - Construct a heap for the list 1, 8, 6, 5, 3, 7, 4 by successive key insertions (top-down algorithm).
 - Is it always true that the bottom-up and top-down algorithms yield the same heap for the same input?
- Outline an algorithm for checking whether an array $H[1..n]$ is a heap and determine its time efficiency.
- Find the smallest and the largest number of keys that a heap of height h can contain.
 - Prove that the height of a heap with n nodes is equal to $\lfloor \log_2 n \rfloor$.

4. Prove the following equality used in Section 6.4

$$\sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1)), \text{ where } n = 2^{h+1} - 2.$$

5. **a.** Design an efficient algorithm for finding and deleting an element of the smallest value in a heap and determine its time efficiency.
b. Design an efficient algorithm for finding and deleting an element of a given value v in a given heap H and determine its time efficiency.
6. Sort the following lists by heapsort by using the array representation of heaps:
a. 1, 2, 3, 4, 5 (in increasing order)
b. 5, 4, 3, 2, 1 (in increasing order)
c. S, O, R, T, I, N, G (in alphabetical order)
7. Is heapsort a stable sorting algorithm?
8. What variety of the transform-and-conquer technique does heapsort represent?
9. Implement three advanced sorting algorithms—mergesort, quicksort, and heapsort—in the language of your choice and investigate their performance on arrays of sizes $n = 10^2, 10^3, 10^4, 10^5$ and 10^6 . For each of these sizes, consider
a. randomly generated files of integers in the range $[1..n]$.
b. increasing files of integers $1, 2, \dots, n$.
c. decreasing files of integers $n, n-1, \dots, 1$.
-  10. Imagine a handful of uncooked spaghetti, individual rods whose lengths represent numbers that need to be sorted.
a. Outline a “spaghetti sort”—a sorting algorithm that takes advantage of this unorthodox representation.
b. What does this example of computer science folklore (see [Dew93]) have to do with the topic of this chapter in general and heapsort in particular?

6.5 Horner's Rule and Binary Exponentiation

In this section we discuss the problem of computing the value of a polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 \quad (6.1)$$

at a given point x and its important special case of computing x^n . Polynomials constitute the most important class of functions because they possess a wealth of good properties on the one hand and can be used for approximating other types of functions on the other. The problem of manipulating polynomials efficiently

has been important for several centuries; new discoveries were still being made in the last 50 years. By far the most important of them was the *fast Fourier transform (FFT)*. The practical importance of this remarkable algorithm, which is based on representing a polynomial by its values at specially chosen points, was such that some people consider it one of the most important algorithmic discoveries of all times. Because of its relative complexity, we do not discuss the FFT algorithm in this book. An interested reader will find a wealth of literature on the subject including reasonably accessible treatments in such textbooks as [Sed88] and [CLRS01].

Horner's Rule

Horner's rule is an old but very elegant and efficient algorithm for evaluating a polynomial. It is named after the British mathematician W. G. Horner, who published it in the early 19th century. But according to Knuth ([KnuII], p. 486), the method was used by Isaac Newton 150 years before Horner. You will appreciate this method much more if you first design an algorithm for the polynomial evaluation problem by yourself and investigate its efficiency (see Problems 1 and 2 in the exercises to this section).

Horner's rule is a good example of the representation change technique since it is based on representing $p(x)$ by a formula different from (6.1). This new formula is obtained from (6.1) by successively taking x as a common factor in the remaining polynomials of diminishing degrees:

$$p(x) = (\dots (a_n x + a_{n-1})x + \dots)x + a_0. \quad (6.2)$$

For example, for the polynomial $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$, we get

$$\begin{aligned} p(x) &= 2x^4 - x^3 + 3x^2 + x - 5 \\ &= x(2x^3 - x^2 + 3x + 1) - 5 \\ &= x(x(2x^2 - x + 3) + 1) - 5 \\ &= x(x(x(2x - 1) + 3) + 1) - 5. \end{aligned} \quad (6.3)$$

It is in formula (6.2) that we will substitute a value of x at which the polynomial needs to be evaluated. It is hard to believe that this is a way to an efficient algorithm, but the unpleasant appearance of formula (6.2) is just that, an appearance. As we shall see, there is no need to go explicitly through transformation leading to it: all we need is an original list of the polynomial's coefficients.

The pen-and-pencil calculation can be conveniently organized with a two-row table. The first row contains the polynomial's coefficients (including all the coefficients equal to zero, if any) listed from the highest a_n to the lowest a_0 . The second row is used to store intermediate results except for its first entry which is a_n . After this initialization, the next table's entry is computed as the x 's value times the last entry in the second row plus the next coefficient from the first row. The final entry computed in this fashion is the value being sought.

EXAMPLE 1 Evaluate $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$ at $x = 3$.

coefficients	2	-1	3	1	-5
$x = 3$	2	$3 \cdot 2 + (-1) = 5$	$3 \cdot 5 + 3 = 18$	$3 \cdot 18 + 1 = 55$	$3 \cdot 55 + (-5) = 160$

So $p(3) = 160$. (On comparing the table's entries with formula (6.3), you will see that $3 \cdot 2 + (-1) = 5$ is the value of $2x - 1$ at $x = 3$, $3 \cdot 5 + 3 = 18$ is the value of $x(2x - 1) + 3$ at $x = 3$, $3 \cdot 18 + 1 = 55$ is the value of $x(x(2x - 1) + 3) + 1$ at $x = 3$, and finally, $3 \cdot 55 + (-5) = 160$ is the value of $x(x(x(2x - 1) + 3) + 1) - 5 = p(x)$ at $x = 3$.) ■

The pseudocode for this algorithm is the shortest one imaginable for a non-trivial algorithm:

ALGORITHM *Horner*($P[0..n], x$)

```
//Evaluates a polynomial at a given point by Horner's rule
//Input: An array  $P[0..n]$  of coefficients of a polynomial of degree  $n$ 
//      (stored from the lowest to the highest) and a number  $x$ 
//Output: The value of the polynomial at  $x$ 
 $p \leftarrow a_n$ 
for  $i \leftarrow n - 1$  downto 0 do
     $p \leftarrow x * p + a_i$ 
return  $p$ 
```

The number of multiplications and the number of additions are given by the same sum:

$$M(n) = A(n) = \sum_{i=0}^{n-1} 1 = n.$$

To appreciate how efficient Horner's rule is, consider only the first term of a polynomial of degree n : $a_n x^n$. Just computing this single term by the brute-force algorithm would require n multiplications, whereas Horner's rule computes, in addition to this term, $n - 1$ other terms, and it still uses the same number of multiplications! It is not surprising that Horner's rule is an optimal algorithm for polynomial evaluation without preprocessing the polynomial's coefficients. But it took scientists 150 years after Horner's publication to come to the realization that such a question was worth investigating.

Horner's rule also has some useful by-products. The intermediate numbers generated by the algorithm in the process of evaluating $p(x)$ at some point x_0 turn out to be the coefficients of the quotient of the division of $p(x)$ by $x - x_0$, while the

final result, in addition to being $p(x_0)$, is equal to the remainder of this division. Thus, according to the example, the quotient and the remainder of the division of $2x^4 - x^3 + 3x^2 + x - 5$ by $x - 3$ are $2x^3 + 5x^2 + 18x + 55$ and 150 , respectively. This division algorithm, known as **synthetic division**, is more convenient than so-called “long division.” (However, unlike the long division, it is applicable only to divisions by $x - c$ where c is some constant.)

Binary Exponentiation

The amazing efficiency of Horner’s rule fades if the method is applied to computing a^n , which is the value of x^n at $x = a$. In fact, it degenerates to the brute-force multiplication of a by itself, with wasteful additions of zeros in between. Since computing a^n (actually, $a^n \bmod m$) is an essential operation in several important primality-testing and encryption methods, we consider now two algorithms for computing a^n that are based on the representation change idea. They both exploit the binary representation of exponent n , but one of them processes this binary string left to right whereas the second does it right to left.

Let

$$n = b_I \dots b_i \dots b_0$$

be the bit string representing a positive integer n in the binary number system. This means that the value of n can be computed as the value of the polynomial

$$p(x) = b_I x^I + \dots + b_i x^i + \dots + b_0 \quad (6.4)$$

at $x = 2$. For example, if $n = 13$, its binary representation is 1101 and

$$13 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0.$$

Let us now compute the value of this polynomial by applying Horner’s rule and see what the method’s operations imply for computing the power

$$a^n = a^{p(2)} = a^{b_I 2^I + \dots + b_i 2^i + \dots + b_0}.$$

Horner’s rule for the binary polynomial $p(2)$	Implications for $a^n = a^{p(2)}$
$p \leftarrow 1$ //the leading digit is always 1 for $n \geq 1$	$a^p \leftarrow a^1$
for $i \leftarrow I - 1$ downto 0 do	for $i \leftarrow I - 1$ downto 0 do
$p \leftarrow 2p + b_i$	$a^p \leftarrow a^{2p+b_i}$

But

$$a^{2p+b_i} = a^{2p} \cdot a^{b_i} = (a^p)^2 \cdot a^{b_i} = \begin{cases} (a^p)^2 & \text{if } b_i = 0 \\ (a^p)^2 \cdot a & \text{if } b_i = 1 \end{cases}.$$

Thus, after initializing the accumulator's value to a , we can scan the bit string representing the exponent to always square the last value of the accumulator and, if the current binary digit is 1, also to multiply it by a . These observations lead to the following **left-to-right binary exponentiation** method of computing a^n .

ALGORITHM *LeftRightBinaryExponentiation*($a, b(n)$)

```
//Computes  $a^n$  by the left-to-right binary exponentiation algorithm
//Input: A number  $a$  and a list  $b(n)$  of binary digits  $b_1, \dots, b_0$ 
//      in the binary expansion of a positive integer  $n$ 
//Output: The value of  $a^n$ 
product  $\leftarrow a$ 
for  $i \leftarrow I - 1$  downto 0 do
    product  $\leftarrow$  product * product
    if  $b_i = 1$  product  $\leftarrow$  product *  $a$ 
return product
```

EXAMPLE Compute a^{13} by the left-to-right binary exponentiation algorithm. Here, $n = 13 = 1101_2$. So we have

binary digits of n	1	1	0	1
product accumulator	a	$a^2 \cdot a = a^3$	$(a^3)^2 = a^6$	$(a^6)^2 \cdot a = a^{13}$

■

Since the algorithm makes one or two multiplications on each repetition of its only loop, the total number of multiplications $M(n)$ made by it in computing a^n is

$$(b - 1) \leq M(n) \leq 2(b - 1),$$

where b is the length of the bit string representing the exponent n . Taking into account that $b - 1 = \lfloor \log_2 n \rfloor$, we can conclude that the efficiency of the left-to-right binary exponentiation is logarithmic. Thus, this algorithm is in a better efficiency class than the brute-force exponentiation, which always requires $n - 1$ multiplications.

The **right-to-left binary exponentiation** uses the same binary polynomial $p(2)$ (see (6.4)) yielding the value of n . But rather than applying Horner's rule to it as the previous method did, this one exploits it differently:

$$a^n = a^{b_1 2^1 + \dots + b_i 2^i + \dots + b_0} = a^{b_1 2^1} \cdot \dots \cdot a^{b_i 2^i} \cdot \dots \cdot a^{b_0}.$$

Thus, a^n can be computed as the product of the terms

$$a^{b_i 2^i} = \begin{cases} a^{2^i} & \text{if } b_i = 1 \\ 1 & \text{if } b_i = 0 \end{cases},$$

that is, the product of consecutive terms a^{2^i} , skipping those for which the binary digit b_i is zero. In addition, we can compute a^{2^i} by simply squaring the same term we computed for the previous value of i since $a^{2^i} = (a^{2^{i-1}})^2$. So we compute all such powers of a from the smallest to the largest (from right to left), but we include in the product accumulator only those whose corresponding binary digit is 1. Here is a pseudocode of this algorithm.

ALGORITHM *RightLeftBinaryExponentiation*($a, b(n)$)

```
//Computes  $a^n$  by the right-to-left binary exponentiation algorithm
//Input: A number  $a$  and a list  $b(n)$  of binary digits  $b_1, \dots, b_0$ 
//      in the binary expansion of a nonnegative integer  $n$ 
//Output: The value of  $a^n$ 
term ←  $a$  //initializes  $a^{2^i}$ 
if  $b_0 = 1$  product ←  $a$ 
else product ← 1
for  $i \leftarrow 1$  to  $I$  do
    term ← term * term
    if  $b_i = 1$  product ← product * term
return product
```

EXAMPLE 3 Compute a^{13} by the right-to-left binary exponentiation method. Here, $n = 13 = 1101_2$. So we have the following table filled in from right to left.

1	1	0	1	binary digits of n
a^8	a^4	a^2	a	terms a^{2^i}
$a^5 \cdot a^8 = a^{13}$	$a \cdot a^4 = a^5$		a	product accumulator

Obviously, the algorithm's efficiency is also logarithmic for the same reason the left-to-right binary multiplication is. The usefulness of both binary exponentiation algorithms is reduced somewhat by their reliance on availability of the explicit binary expansion of exponent n . Problem 8 in the exercises asks you to design an algorithm that does not have this shortcoming.

Exercises 6.5

1. Consider the following brute-force algorithm for evaluating a polynomial.

ALGORITHM *BruteForcePolynomialEvaluation*($P[0..n], x$)

```
//The algorithm computes the value of polynomial P at a given point x
//by the "highest to lowest term" brute-force algorithm
//Input: An array P[0..n] of the coefficients of a polynomial of degree n,
//       stored from the lowest to the highest and a number x
//Output: The value of the polynomial at the point x
p ← 0.0
for i ← n downto 0 do
    power ← 1
    for j ← 1 to i do
        power ← power * x
    p ← p + P[i] * power
return p
```

Find the total number of multiplications and the number of additions made by this algorithm.

2. Write a pseudocode for the brute-force polynomial evaluation that stems from substituting a given value of the variable into the polynomial's formula and evaluating it from the lowest term to the highest one. Determine the number of multiplications and the number of additions made by this algorithm.
3. **a.** Estimate how much faster Horner's rule is compared to the "lowest to highest term" brute-force algorithm of Problem 2 if (i) the time of one multiplication is significantly larger than the time of one addition; (ii) the time of one multiplication is about the same as the time of one addition.
 - b.** Is Horner's rule more time efficient at the expense of being less space efficient than the brute-force algorithm?
4. **a.** Apply Horner's rule to evaluate the polynomial

$$p(x) = 3x^4 - x^3 + 2x + 5 \text{ at } x = -2.$$

- b.** Use the results of Horner's rule application to find the quotient and remainder of the division of $p(x)$ by $x + 2$.
5. Compare the number of multiplications and additions/subtractions needed by the "long division" of a polynomial $p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_0$ by $x - c$ where c is some constant with the number of these operations in "synthetic division."

6.
 - a. Apply the left-to-right binary exponentiation algorithm to compute a^{17} .
 - b. Is it possible to extend the left-to-right binary exponentiation algorithm to work for every nonnegative integer exponent?
7. Apply the right-to-left binary exponentiation algorithm to compute a^{17} .
8. Design a nonrecursive algorithm for computing a^n that mimics the right-to-left binary exponentiation but does not explicitly use the binary representation of n .
9. Is it a good idea to use a general-purpose polynomial evaluation algorithm such as Horner's rule to evaluate the polynomial $p(x) = x^n + x^{n-1} + \dots + x + 1$?
10. According to the corollary of the Fundamental Theorem of Algebra, every polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_0$$

can be represented in the form

$$p(x) = a_n (x - x_1)(x - x_2) \dots (x - x_n)$$

where x_1, x_2, \dots, x_n are the roots of the polynomial (generally, complex and not necessarily distinct). Discuss which of the two representations is more convenient for each of the following operations:

- a. Polynomial evaluation at a given point
- b. Addition of two polynomials
- c. Multiplication of two polynomials

6.6 Problem Reduction

Here is my version of a well-known joke about mathematicians. Professor X, a noted mathematician, noticed that when his wife wanted to boil water for their tea, she took their kettle from a kitchen cabinet, filled it with water, and put it on the stove. Once, when his wife was away (if you have to know, she was signing her best-seller in a local bookstore), the professor had to boil water by himself. He saw that the kettle was sitting on the kitchen counter. What did Professor X do? He put the kettle in the cupboard first and then proceeded to follow his wife's routine.

The way Professor X approached his task is an example of an important problem-solving strategy called **problem reduction**. If you need to solve a problem, reduce it to another problem that you know how to solve (Figure 6.15).

The joke about the professor notwithstanding, the idea of problem reduction plays a central role in theoretical computer science where it is used to classify

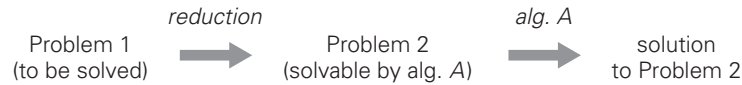


FIGURE 6.15 Problem reduction strategy

problems according to their complexity. We will touch on this classification in Chapter 10. But the strategy can be used for actual problem solving, too. The practical difficulty in applying it lies, of course, in finding a problem to which the problem at hand should be reduced. Moreover, if we want our efforts to be of practical value, we need our reduction-based algorithm to be more efficient than solving the original problem directly.

Note that we have already encountered this technique earlier in the book. In Section 6.5, for example, we mentioned the so-called synthetic division done by applying Horner's rule for polynomial evaluation. In Section 4.6, we used the following fact from analytical geometry: if $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, and $p_3 = (x_3, y_3)$ are three arbitrary points in the plane, then the determinant

$$\det \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{bmatrix} = x_1y_2 + x_3y_1 + x_2y_3 - x_3y_2 - x_2y_1 - x_1y_3$$

is positive if and only if the point p_3 is to the left of the directed line $\overrightarrow{p_1p_2}$ through points p_1 and p_2 . In other words, we reduced a geometric question about the relative locations of three points to a question about the sign of a determinant. In fact, the entire idea of analytical geometry is based on reducing geometric problems to algebraic ones. And the vast majority of geometric algorithms take advantage of this historic insight by René Descartes (1596–1650). In this section, we give a few more examples of algorithms based on the problem reduction strategy.

Computing the Least Common Multiple

Recall that the *least common multiple* of two positive integers m and n , denoted $lcm(m, n)$, is defined as the smallest integer that is divisible by both m and n . For example, $lcm(24, 60) = 120$, and $lcm(11, 5) = 55$. The least common multiple is one of the most important notions in elementary arithmetic and algebra; perhaps you remember the following middle-school method for computing it. Given the prime factorizations of m and n , $lcm(m, n)$ can be computed as the product of all the common prime factors of m and n times the product of m 's prime factors that are not in n times n 's prime factors that are not in m . For example,

$$\begin{aligned} 24 &= 2 \cdot 2 \cdot 2 \cdot 3 \\ 60 &= 2 \cdot 2 \cdot 3 \cdot 5 \\ lcm(24, 60) &= (2 \cdot 2 \cdot 3) \cdot 2 \cdot 5 = 120. \end{aligned}$$

As a computational procedure, this algorithm has the same drawbacks as the middle-school algorithm for computing the greatest common divisor discussed in Section 1.1: it is inefficient and requires a list of consecutive primes.

A much more efficient algorithm for computing the least common multiple can be designed by using problem reduction. After all, there is a very efficient algorithm (Euclid's algorithm) for finding the greatest common divisor, which is a product of all the common prime factors of m and n . Can we find a formula relating $\text{lcm}(m, n)$ and $\text{gcd}(m, n)$? It is not difficult to see that the product of $\text{lcm}(m, n)$ and $\text{gcd}(m, n)$ includes every factor of m and n exactly once and hence is simply equal to the product of m and n . This observation leads to the formula

$$\text{lcm}(m, n) = \frac{m \cdot n}{\text{gcd}(m, n)},$$

where $\text{gcd}(m, n)$ can be computed very efficiently by Euclid's algorithm.

Counting Paths in a Graph

As our next example, we will consider the problem of counting paths between two vertices in a graph. It is not difficult to prove by mathematical induction that the number of different paths of length $k > 0$ from the i th vertex to the j th vertex of a graph (undirected or directed) equals the (i, j) th element of A^k where A is the adjacency matrix of the graph. (Incidentally, the exponentiation algorithms we discussed before for computing powers of numbers are applicable to matrices as well.) Thus, the problem of counting a graph's paths can be solved with an algorithm for computing an appropriate power of its adjacency matrix.

As a specific example, consider the graph of Figure 6.16. Its adjacency matrix A and its square A^2 indicate the number of paths of length 1 and 2, respectively, between the corresponding vertices of the graph. In particular, there are three paths of length 2 that start and end at vertex a : $a - b - a$, $a - c - a$, and $a - d - a$ but only one path of length 2 from a to c : $a - d - c$.

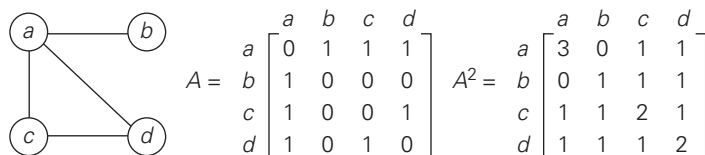


FIGURE 6.16 A graph, its adjacency matrix A , and its square A^2 . The elements of A and A^2 indicate the number of paths of lengths 1 and 2, respectively.

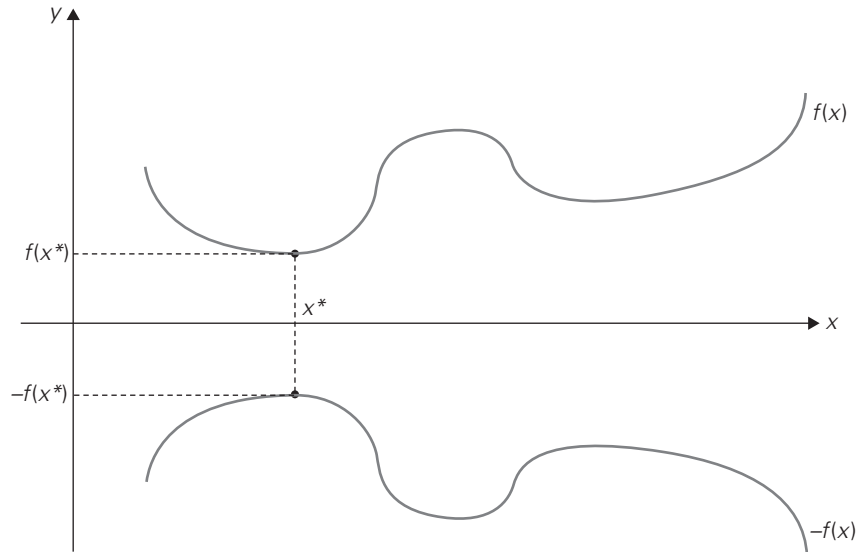


FIGURE 6.17 Relationship between minimization and maximization problems:
 $\min f(x) = -\max[-f(x)]$

Reduction of Optimization Problems

Our next example deals with solving optimization problems. If a problem seeks to find a maximum of some function, it is said to be a **maximization problem**; if it seeks to find a function's minimum, it is said to be a **minimization problem**. Suppose now that you have to find a minimum of some function $f(x)$ and you know an algorithm for maximizing the function. How can you take advantage of the latter? The answer lies in the simple formula

$$\min f(x) = -\max[-f(x)].$$

In other words, to minimize a function we can maximize its negative instead and, to get a correct minimal value of the function itself, change the sign of the answer. This property is illustrated for a function of one real variable in Figure 6.17.

Of course, the formula

$$\max f(x) = -\min[-f(x)]$$

is valid as well; it shows how a maximization problem can be reduced to an equivalent minimization problem.

This relationship between minimization and maximization problems is very general: it holds for functions defined on any domain D . In particular, we can

apply it to functions of several variables subject to additional constraints. A very important class of such problems is introduced in the next portion of this section.

Now that we are on the topic of function optimization, it is worth pointing out that the standard calculus procedure for finding extremum points of a function is, in fact, also based on problem reduction. Indeed, it suggests finding the function's derivative $f'(x)$ and then solving the equation $f'(x) = 0$ to find the function's critical points. In other words, the optimization problem is reduced to the problem of solving an equation as the principal part of finding extremum points. Note that we are not calling the calculus procedure an algorithm because it is not clearly defined. In fact, there is no general method for solving equations. A little secret of calculus textbooks is that problems are carefully selected so that critical points can always be found without difficulty. This makes the lives of both students and instructors easier but, in the process, may unintentionally create a wrong impression in students' minds.

Linear Programming

Many problems of optimal decision making can be reduced to an instance of the *linear programming* problem, which is a problem of optimizing a linear function of several variables subject to constraints in the form of linear equations and linear inequalities.

EXAMPLE 1 Consider a university endowment that needs to invest \$100 million. This sum must be split between three types of investments: stocks, bonds, and cash. The endowment managers expect an annual return of 10%, 7%, and 3% for their stock, bond, and cash investments, respectively. Since stocks are more risky than bonds, the endowment rules require an amount invested in stocks to be no more than one third of the moneys invested in bonds. In addition, at least 25% of the total amount invested in stocks and bonds must be invested in cash. How should the managers invest the money to maximize the return?

Let us create a mathematical model of this problem. Let x , y , and z be the amounts (in millions of dollars) invested in stocks, bonds, and cash, respectively. By using these variables, we can pose the following optimization problem:

$$\begin{aligned} &\text{maximize} && 0.10x + 0.07y + 0.03z \\ &\text{subject to} && x + y + z = 100 \\ &&& x \leq \frac{1}{3}y \\ &&& z \geq 0.25(x + y) \\ &&& x \geq 0, y \geq 0, z \geq 0. \end{aligned}$$

Although this particular problem is both small and simple, it does show how a problem of optimal decision making can be reduced to an instance of the general linear programming problem

$$\begin{aligned} &\text{maximize (or minimize)} && c_1x_1 + \cdots + c_nx_n \\ &\text{subject to} && a_{i1}x_1 + \cdots + a_{in}x_n \leq (\text{or } \geq \text{ or } =)b_i \quad \text{for } i = 1, \dots, m \\ &&& x_1 \geq 0, \dots, x_n \geq 0. \end{aligned}$$

(The last group of constraints—the so-called nonnegativity constraints—are, strictly speaking, unnecessary because they are special cases of more general constraints $a_{i1}x_1 + \cdots + a_{in}x_n \geq b_i$, but it is convenient to treat them separately.)

Linear programming has proved to be flexible enough to model a wide variety of important applications, such as airline crew scheduling, transportation and communication network planning, oil exploration and refining, and industrial production optimization. In fact, linear programming is considered by many as one of the most important achievements in the history of applied mathematics. The classic algorithm for this problem is called the *simplex method*; it was discovered by the U.S. mathematician George Dantzig in the 1940s [Dan63]. Although the worst-case efficiency of this algorithm is known to be exponential, it performs very well on typical inputs. The efforts of many computer scientists over the last 50 years have polished the algorithm and its computer implementations to a point where problems with tens if not hundreds of thousands of variables and constraints can be solved in a reasonable amount of time. Moreover, a few other algorithms for solving the general linear programming problem have been discovered relatively recently; the best-known of them is by Narendra Karmarkar [Kar84]. The theoretical advantage of these newer algorithms lies in their proven polynomial worst-case efficiency; Karmarkar's algorithm has also been found to be competitive with the simplex method in empirical tests.

It is important to stress, however, that the simplex method and Karmarkar's algorithm can successfully handle only linear programming problems that do not limit its variables to integer values. When variables of a linear programming problem are required to be integers, the linear programming problem is said to be an *integer linear programming* problem. Integer linear programming problems are known to be much more difficult. There is no known polynomial-time algorithm for solving an arbitrary instance of the general integer linear programming problem and, as we shall see in Chapter 10, such an algorithm quite possibly does not exist. Other approaches such as the branch-and-bound technique discussed in Section 11.2 are typically used for solving integer linear programming problems.

EXAMPLE 2 Let us see how the knapsack problem can be reduced to a linear programming problem. Recall from Section 3.4 that the knapsack problem can be posed as follows. Given a knapsack of capacity W and n items of weights w_1, \dots, w_n and values v_1, \dots, v_n , find the most valuable subset of the items that fits into the knapsack. We consider first the so-called *continuous* or *fractional* version of the

problem in which any fraction of any item given can be taken into the knapsack. Let x_j , $j = 1, \dots, n$, be a variable representing a fraction of item j taken into the knapsack. Obviously, x_j must satisfy the inequality $0 \leq x_j \leq 1$. Then the total weight of the items selected can be expressed by the sums $\sum_{j=1}^n w_j x_j$ and the total value of the items selected by the sum $\sum_{j=1}^n v_j x_j$. Thus, the continuous version of the knapsack problem can be posed as the following linear programming problem:

$$\begin{aligned} &\text{maximize} && \sum_{j=1}^n v_j x_j \\ &\text{subject to} && \sum_{j=1}^n w_j x_j \leq W \\ &&& 0 \leq x_j \leq 1 \text{ for } j = 1, \dots, n. \end{aligned}$$

There is no need to apply a general method for solving linear programming problems here: this particular problem can be solved by a simple special algorithm that will be introduced in Section 11.3. (But why wait? Try to discover it on your own now.) This reduction of the knapsack problem to an instance of the linear programming problem is still useful, though, to prove the correctness of the algorithm in question.

In the so called *discrete* or **0-1** version of the knapsack problem, we are allowed either to take an item in its entirety or not to take it at all. Hence, we have the following integer linear programming problem for this version:

$$\begin{aligned} &\text{maximize} && \sum_{j=1}^n v_j x_j \\ &\text{subject to} && \sum_{j=1}^n w_j x_j \leq W \\ &&& x_j \in \{0, 1\} \text{ for } j = 1, \dots, n. \end{aligned}$$

This seemingly minor modification makes a drastic difference for the complexity of this and similar problems constrained to take only discrete values in their potential ranges. Despite the fact that the 0-1 version might seem to be easier because it can ignore any subset of the continuous version that has a fractional value of an item, the 0-1 version is, in fact, much more complicated than its continuous counterpart. The reader interested in specific algorithms for solving this problem will find a wealth of literature on the subject including the monograph by Martello and Toth [MT90]. ■

Reduction to Graph Problems

As we mentioned in Section 1.3, many problems can be solved by a reduction to one of the standard graph problems. This is true in particular for a variety of puzzles and games. In these applications, vertices of a graph typically represent possible states of the problem in question while edges indicate permitted transitions among such states. One of the graph's vertices represents an initial state, while another represents a goal state of the problem. (There might be several vertices of the latter kind.) Such a graph is called a *state-space graph*. Thus, the transformation just described reduces the problem to the question about a path from the initial-state vertex to a goal-state vertex.

EXAMPLE As a specific example, let us revisit the classic river-crossing puzzle of Problem 1 in the exercises to Section 1.2. A peasant finds himself on a river bank with a wolf, a goat, and a head of cabbage. He needs to transport all three to the other side of the river in his boat. However, the boat has room only for the peasant himself and one other item (either the wolf, the goat, or the cabbage). In his absence, the wolf would eat the goat, and the goat would eat the cabbage. Find a way for the peasant to solve his problem or prove that it has no solution.

The state-space graph for this problem is given in Figure 6.18. Its vertices are labeled to indicate the states they represent: P, w, g, c stand for the peasant, the wolf, the goat, and the cabbage, respectively; the two bars || denote the river; for convenience, we also label the edges by indicating the boat's occupants for

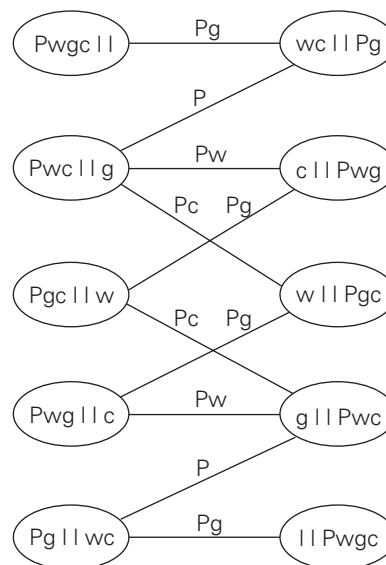


FIGURE 6.18 State-space graph for the peasant, wolf, goat, and cabbage puzzle

each crossing. In terms of this graph, we are interested in finding a path from the initial-state vertex labeled $Pwgc||$ to the final-state vertex labeled $||Pwgc$.

It is easy to see that there exist two distinct simple paths from the initial-state vertex to the final state vertex (what are they?). If we find them by applying breadth-first search, we get a formal proof that these paths have the smallest number of edges possible. Hence, this puzzle has two solutions, each of which requires seven river crossings. ■

Our success in solving this simple puzzle should not lead you to believe that generating and investigating state-space graphs is always a straightforward task. To get a better appreciation of state-space graphs, consult books on artificial intelligence (AI), the branch of computer science in which such problems are a principal subject. In this book, we will deal with an important special case of state-space graphs in Sections 11.1 and 11.2.

Exercises 6.6

1. a. Prove the equality

$$lcm(m, n) = \frac{m \cdot n}{gcd(m, n)}$$

that underlies the algorithm for computing $lcm(m, n)$.

- b. Euclid's algorithm is known to be in $O(\log n)$. If it is the algorithm that is used for computing $gcd(m, n)$, what is the efficiency of the algorithm for computing $lcm(m, n)$?
2. You are given a list of numbers for which you need to construct a min-heap. (A min-heap is a complete binary tree in which every key is less than or equal to the keys in its children.) How would you use an algorithm for constructing a max-heap (a heap as defined in Section 6.4) to construct a min-heap?
3. Prove that the number of different paths of length $k > 0$ from the i th vertex to the j th vertex of a graph (undirected or directed) equals the (i, j) th element of A^k where A is the adjacency matrix of the graph.
4. a. Design an algorithm with a time efficiency better than cubic for checking whether a graph with n vertices contains a cycle of length 3 [Man89].
 b. Consider the following algorithm for the same problem. Starting at an arbitrary vertex, traverse the graph by depth-first search and check whether its depth-first search forest has a vertex with a back edge leading to its grandparent. If it does, the graph contains a triangle; if it does not, the graph does not contain a triangle as its subgraph. Is this algorithm correct?
5. Given $n > 3$ points $P_1 = (x_1, y_1), \dots, P_n = (x_n, y_n)$ in the coordinate plane, design an algorithm to check whether all the points lie within a triangle with

its vertices at three of the points given. (You can either design an algorithm from scratch or reduce the problem to another one with a known algorithm.)

6. Consider the problem of finding, for a given positive integer n , the pair of integers whose sum is n and whose product is as large as possible. Design an efficient algorithm for this problem and indicate its efficiency class.
7. The assignment problem, introduced in Section 3.4, can be stated as follows. There are n people who need to be assigned to execute n jobs, one person per job. (That is, each person is assigned to exactly one job and each job is assigned to exactly one person.) The cost that would accrue if the i th person is assigned to the j th job is a known quantity $C[i, j]$ for each pair $i, j = 1, \dots, n$. The problem is to assign the people to the jobs to minimize the total cost of the assignment. Express the assignment problem as a 0-1 linear programming problem.
8. Solve the instance of the linear programming problem given in Section 6.6

$$\begin{aligned}
 &\text{maximize} && 0.10x + 0.07y + 0.03z \\
 &\text{subject to} && x + y + z = 100 \\
 &&& x \leq \frac{1}{3}y \\
 &&& z \geq 0.25(x + y) \\
 &&& x \geq 0, y \geq 0, z \geq 0.
 \end{aligned}$$

9. The graph-coloring problem is usually stated as the vertex-coloring problem: assign the smallest number of colors to vertices of a given graph so that no two adjacent vertices are the same color. Consider the **edge-coloring** problem: assign the smallest number of colors possible to edges of a given graph so that no two edges with the same end point are the same color. Explain how the edge-coloring problem can be reduced to a vertex-coloring problem.



10. **Jealous-Husbands Puzzle** There are n ($n \geq 2$) married couples who need to cross a river. They have a boat that can hold no more than two people at a time. To complicate matters, all the husbands are jealous and will not agree on any crossing procedure that would put a wife on the same bank of the river with another woman's husband without the wife's husband being there too, even if there are other people on the same bank. Can they cross the river under such constraints?
 - a. Solve the problem for $n = 2$.
 - b. Solve the problem for $n = 3$, which is the classical version of this problem.
 - c. Does the problem have a solution for every $n \geq 4$? If it does, indicate how many river crossings it will take; if it does not, explain why.

SUMMARY

- *Transform-and-conquer* is the fourth general algorithm design (and problem-solving) strategy discussed in the book. It is, in fact, a group of techniques based on the idea of transformation to a problem that is easier to solve.
- There are three principal varieties of the transform-and-conquer strategy: *instance simplification*, *representation change*, and *problem reduction*.
- *Instance simplification* is a technique of transforming an instance of a problem to an instance of the same problem with some special property that makes the problem easier to solve. List presorting, Gaussian elimination, and AVL trees are good examples of this technique.
- *Representation change* implies changing one representation of a problem's instance into another representation of the same instance. Examples discussed in this chapter include representation of a set by a 2-3 tree, heaps for representing priority queues, Horner's rule for polynomial evaluation, and two binary exponentiation algorithms.
- *Problem reduction* calls for transforming a problem given to another problem that can be solved by a known algorithm. Among examples of applying this idea to algorithmic problem solving (see Section 6.6), reductions to linear programming and reductions to graph problems are especially important.
- Some examples used to illustrate the transform-and-conquer techniques happen to be very important data structures and algorithms. They are: heaps and heapsort, AVL and 2-3 trees, Gaussian elimination, and Horner's rule.
- A *heap* is an essentially complete binary tree with keys (one per node) satisfying the parental dominance requirement. Though defined as binary trees, heaps are normally implemented as arrays. Heaps are most important for the efficient implementation of priority queues; they also underlie heapsort.
- *Heapsort* is a theoretically important sorting algorithm based on arranging elements of an array in a heap and then successively removing the largest element from a remaining heap. The algorithm's running time is in $\Theta(n \log n)$ both in the worst case and in the average case; in addition, it is in place.
- *AVL trees* are binary search trees that are always balanced to the extent possible for a binary tree. The balance is maintained by transformations of four types called *rotations*. All basic operations on AVL trees are in $\Theta(\log n)$; it eliminates the bad worst-case efficiency of classic binary search trees.

- *2-3 trees* achieve a perfect balance in a search tree by allowing a node to contain up to two ordered keys and have up to three children. This idea can be generalized to yield very important *B-trees*, discussed later in the book.
- *Gaussian elimination*—an algorithm for solving systems of linear equations—is a principal algorithm in linear algebra. It solves a system by transforming it to an equivalent system with an upper-triangular coefficient matrix, which is easy to solve by backward substitutions. Gaussian elimination requires about $\frac{1}{3}n^3$ multiplications.
- *Horner's rule* is an optimal algorithm for polynomial evaluation without coefficient preprocessing. It requires only n multiplications and n additions. It also has a few useful by-products such as the synthetic division algorithm.
- Two *binary exponentiation* algorithms for computing a^n were introduced in Section 6.5. Both of them exploit the binary representation of the exponent n , but they process it in the opposite directions: left to right and right to left.
- *Linear programming* concerns optimizing a linear function of several variables subject to constraints in the form of linear equations and linear inequalities. There are efficient algorithms capable of solving very large instances of this problem with many thousands of variables and constraints, provided the variables are not required to be integers. The latter, called *integer linear programming* problems, constitute a much more difficult class of problems.