CHAPTER 2

Iteration, Induction, and Recursion

The power of computers comes from their ability to execute the same task, or different versions of the same task, repeatedly. In computing, the theme of *iteration* is met in a number of guises. Many concepts in data models, such as lists, are forms of repetition, as "A list either is empty or is one element followed by another, then another, and so on." Programs and algorithms use iteration to perform repetitive jobs without requiring a large number of similar steps to be specified individually, as "Do the next step 1000 times." Programming languages use looping constructs, like the while- and for-statements of C, to implement iterative algorithms.

Closely related to repetition is *recursion,* a technique in which a concept is defined, directly or indirectly, in terms of itself. For example, we could have defined a list by saying "A list either is empty or is an element followed by a list." Recursion is supported by many programming languages. In C, a function F can call itself, either directly from within the body of F itself, or indirectly by calling some other function, which calls another, and another, and so on, until finally some function in the sequence calls F. Another important idea, *induction,* is closely related to "recursion" and is used in many mathematical proofs.

Iteration, induction, and recursion are fundamental concepts that appear in many forms in data models, data structures, and algorithms. The following list gives some examples of uses of these concepts; each will be covered in some detail in this book.

1. *Iterative techniques.* The simplest way to perform a sequence of operations repeatedly is to use an iterative construct such as the for-statement of C.

2. *Recursive programming.* C and many other languages permit recursive functions, which call themselves either directly or indirectly. Often, beginning programmers are more secure writing iterative programs than recursive ones, but an important goal of this book is to accustom the reader to thinking and programming recursively, when appropriate. Recursive programs can be simpler to write, analyze, and understand.

Notation: The Summation and Product Symbols

An oversized Greek capital letter sigma is often used to denote a summation, as in $\sum_{i=1}^{n} i$. This particular expression represents the sum of the integers from 1 to n; that is, it stands for the sum $1 + 2 + 3 + \cdots + n$. More generally, we can sum any function $f(i)$ of the summation index i. (Of course, the index could be some symbol other than i.) The expression $\sum_{i=a}^{b} f(i)$ stands for

$$f(a) + f(a+1) + f(a+2) + \cdots + f(b)$$

For example, $\sum_{j=2}^{m} j^2$ stands for the sum $4 + 9 + 16 + \cdots + m^2$. Here, the function f is "squaring," and we used index j instead of i.

As a special case, if $b < a$, then there are no terms in the sum $\sum_{i=a}^{b} f(i)$, and the value of the expression, by convention, is taken to be 0. If $b = a$, then there is exactly one term, that for $i = a$. Thus, the value of the sum $\sum_{i=a}^{a} f(i)$ is just $f(a)$.

The analogous notation for products uses an oversized capital pi. The expression $\prod_{i=a}^{b} f(i)$ stands for the product $f(a) \times f(a+1) \times f(a+2) \times \cdots \times f(b)$; if $b < a$, the product is taken to be 1.

3. *Proofs by induction.* An important technique for showing that a statement is true is "proof by induction." We shall cover inductive proofs extensively, starting in Section 2.3. The following is the simplest form of an inductive proof. We begin with a statement $S(n)$ involving a variable n; we wish to prove that $S(n)$ is true. We prove $S(n)$ by first proving a *basis*, that is, the statement $S(n)$ for a particular value of n. For example, we could let $n = 0$ and prove the statement $S(0)$. Second, we must prove an *inductive step*, in which we prove that the statement S, for one value of its argument, follows from the same statement S for the previous values of its argument; that is, $S(n)$ implies $S(n+1)$ for all $n \geq 0$. For example, $S(n)$ might be the familiar summation formula

Basis

Inductive step

$$\sum_{i=1}^{n} i = n(n+1)/2 \tag{2.1}$$

which says that the sum of the integers from 1 to n equals $n(n+1)/2$. The basis could be $S(1)$ — that is, Equation (2.1) with 1 in place of n — which is just the equality $1 = 1 \times 2/2$. The inductive step is to show that $\sum_{i=1}^{n} i = n(n+1)/2$ implies that $\sum_{i=1}^{n+1} i = (n+1)(n+2)/2$; the former is $S(n)$, which is Equation (2.1) itself, while the latter is $S(n+1)$, which is Equation (2.1) with $n+1$ replacing n everywhere n appears. Section 2.3 will show us how to construct proofs such as this.

4. *Proofs of program correctness.* In computer science, we often wish to prove, formally or informally, that a statement $S(n)$ about a program is true. The statement $S(n)$ might, for example, describe what is true on the nth iteration of some loop or what is true for the nth recursive call to some function. Proofs of this sort are generally inductive proofs.

5. *Inductive definitions.* Many important concepts of computer science, especially those involving data models, are best defined by an induction in which we give

a basis rule defining the simplest example or examples of the concept, and an inductive rule or rules, where we build larger instances of the concept from smaller ones. For instance, we noted that a list can be defined by a basis rule (an empty list is a list) together with an inductive rule (an element followed by a list is also a list).

6. *Analysis of running time.* An important criterion for the "goodness" of an algorithm is how long it takes to run on inputs of various sizes (its "running time"). When the algorithm involves recursion, we use a formula called a *recurrence equation*, which is an inductive definition that predicts how long the algorithm takes to run on inputs of different sizes.

Each of these subjects, except the last, is introduced in this chapter; the running time of a program is the topic of Chapter 3.

2.1 What This Chapter Is About

In this chapter we meet the following major concepts.

- Iterative programming (Section 2.2)
- Inductive proofs (Sections 2.3 and 2.4)
- Inductive definitions (Section 2.6)
- Recursive programming (Sections 2.7 and 2.8)
- Proving the correctness of a program (Sections 2.5 and 2.9)

In addition, we spotlight, through examples of these concepts, several interesting and important ideas from computer science. Among these are

- Sorting algorithms, including selection sort (Section 2.2) and merge sort (Section 2.8)
- Parity checking and detection of errors in data (Section 2.3)
- Arithmetic expressions and their transformation using algebraic laws (Sections 2.4 and 2.6)
- Balanced parentheses (Section 2.6)

2.2 Iteration

Each beginning programmer learns to use iteration, employing some kind of looping construct such as the for- or while-statement of C. In this section, we present an example of an iterative algorithm, called "selection sort." In Section 2.5 we shall prove by induction that this algorithm does indeed sort, and we shall analyze its running time in Section 3.6. In Section 2.8, we shall show how recursion can help us devise a more efficient sorting algorithm using a technique called "divide and conquer."

Common Themes: Self-Definition and Basis-Induction

As you study this chapter, you should be alert to two themes that run through the various concepts. The first is self-definition, in which a concept is defined, or built, in terms of itself. For example, we mentioned that a list can be defined as being empty or as being an element followed by a list.

The second theme is basis-induction. Recursive functions usually have some sort of test for a "basis" case where no recursive calls are made and an "inductive" case where one or more recursive calls are made. Inductive proofs are well known to consist of a basis and an inductive step, as do inductive definitions. This basis-induction pairing is so important that these words are highlighted in the text to introduce each occurrence of a basis case or an inductive step.

There is no paradox or circularity involved in properly used self-definition, because the self-defined subparts are always "smaller" than the object being defined. Further, after a finite number of steps to smaller parts, we arrive at the basis case, at which the self-definition ends. For example, a list L is built from an element and a list that is one element shorter than L. When we reach a list with zero elements, we have the basis case of the definition of a list: "The empty list is a list."

As another example, if a recursive function works, the arguments of the call must, in some sense, be "smaller" than the arguments of the calling copy of the function. Moreover, after some number of recursive calls, we must get to arguments that are so "small" that the function does not make any more recursive calls.

Sorting

To sort a list of n elements we need to permute the elements of the list so that they appear in nondecreasing order.

✦ **Example 2.1.** Suppose we are given the list of integers $(3, 1, 4, 1, 5, 9, 2, 6, 5)$. We sort this list by permuting it into the sequence $(1, 1, 2, 3, 4, 5, 5, 6, 9)$. Note that sorting not only orders the values so that each is either less than or equal to the one that follows, but it also preserves the number of occurrences of each value. Thus, the sorted list has two 1's, two 5's, and one each of the numbers that appear once in the original list. ✦

We can sort a list of elements of any type as long as the elements have a "less-than" order defined on them, which we usually represent by the symbol $<$. For example, if the values are real numbers or integers, then the symbol $<$ stands for the usual less-than relation on reals or integers, and if the values are character strings, we would use the lexicographic order on strings. (See the box on "Lexicographic Order.") Sometimes when the elements are complex, such as structures, we might use only a part of each element, such as one particular field, for the comparison.

Sorted list

The comparison $a \leq b$ means, as always, that either $a < b$ or a and b are the same value. A list $(a_1, a_2, \ldots, a_n)$ is said to be *sorted* if $a_1 \leq a_2 \leq \cdots \leq a_n$; that is, if the values are in nondecreasing order. *Sorting* is the operation of taking an arbitrary list $(a_1, a_2, \ldots, a_n)$ and producing a list $(b_1, b_2, \ldots, b_n)$ such that

Lexicographic Order

The usual way in which two character strings are compared is according to their *lexicographic order*. Let $c_1c_2\cdots c_k$ and $d_1d_2\cdots d_m$ be two strings, where each of the c's and d's represents a single character. The lengths of the strings, k and m, need not be the same. We assume that there is a $<$ ordering on characters; for example, in C characters are small integers, so character constants and variables can be used as integers in arithmetic expressions. Thus we can use the conventional $<$ relation on integers to tell which of two characters is "less than" the other. This ordering includes the natural notion that lower-case letters appearing earlier in the alphabet are "less than" lower-case letters appearing later in the alphabet, and the same holds for upper-case letters.

We may then define the ordering on character strings called the *lexicographic, dictionary,* or *alphabetic* ordering, as follows. We say $c_1c_2\cdots c_k < d_1d_2\cdots d_m$ if either of the following holds:

Proper prefix

1. The first string is a *proper prefix* of the second, which means that $k < m$ and for $i = 1, 2, \ldots, k$ we have $c_i = d_i$. According to this rule, `bat` < `batter`. As a special case of this rule, we could have $k = 0$, in which case the first string has no characters in it. We shall use ϵ, the Greek letter epsilon, to denote the *empty string*, the string with zero characters. When $k = 0$, rule (1) says that $\epsilon < s$ for any nonempty string s.

Empty string

2. For some value of $i > 0$, the first $i - 1$ characters of the two strings agree, but the ith character of the first string is less than the ith character of the second string. That is, $c_j = d_j$ for $j = 1, 2, \ldots, i - 1$, and $c_i < d_i$. According to this rule, `ball` < `base`, because the two words first differ at position 3, and at that position `ball` has an `l`, which precedes the character `s` found in the third position of `base`.

1. List $(b_1, b_2, \ldots, b_n)$ is sorted.

Permutation

2. List $(b_1, b_2, \ldots, b_n)$ is a *permutation* of the original list. That is, each value appears in list $(a_1, a_2, \ldots, a_n)$ exactly as many times as that value appears in list $(b_1, b_2, \ldots, b_n)$.

A *sorting algorithm* takes as input an arbitrary list and produces as output a sorted list that is a permutation of the input.

✦ **Example 2.2.** Consider the list of words

```
base, ball, mound, bat, glove, batter
```

Given this input, and using lexicographic order, a sorting algorithm would produce this output: `ball, base, bat, batter, glove, mound`. ✦

Selection Sort: An Iterative Sorting Algorithm

Suppose we have an array `A` of n integers that we wish to sort into nondecreasing

Convention Regarding Names and Values

We can think of a variable as a box with a name and a value. When we refer to a variable, such as `abc`, we use the constant-width, or "computer" font for its name, as we did in this sentence. When we refer to the value of the variable `abc`, we shall use italics, as *abc*. To summarize, `abc` refers to the name of the box, and *abc* to its contents.

order. We may do so by iterating a step in which a smallest element[1] not yet part of the sorted portion of the array is found and exchanged with the element in the first position of the unsorted part of the array. In the first iteration, we find ("select") a smallest element among the values found in the full array `A[0..n-1]` and exchange it with `A[0]`.[2] In the second iteration, we find a smallest element in `A[1..n-1]` and exchange it with `A[1]`. We continue these iterations. At the start of the $i+1$st iteration, `A[0..i-1]` contains the i smallest elements in `A` sorted in nondecreasing order, and the remaining elements of the array are in no particular order. A picture of `A` just before the $i+1$st iteration is shown in Fig. 2.1.

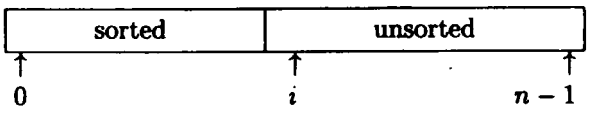


Fig. 2.1. Picture of array just before the $i+1$st iteration of selection sort.

In the $i+1$st iteration, we find a smallest element in `A[i..n-1]` and exchange it with `A[i]`. Thus, after the $i+1$st iteration, `A[0..i]` contains the $i+1$ smallest elements sorted in nondecreasing order. After the $(n-1)$st iteration, the entire array is sorted.

A C function for selection sort is shown in Fig. 2.2. This function, whose name is `SelectionSort`, takes an array `A` as the first argument. The second argument, n, is the length of array `A`.

Lines (2) through (5) select a smallest element in the unsorted part of the array, `A[i..n-1]`. We begin by setting the value of index `small` to i in line (2). The for-loop of lines (3) through (5) consider all higher indexes j in turn, and *small* is set to j if `A[j]` has a smaller value than any of the array elements in the range `A[i..j-1]`. As a result, we set the variable `small` to the index of the first occurrence of the smallest element in `A[i..n-1]`.

After choosing a value for the index `small`, we exchange the element in that position with the element in `A[i]`, in lines (6) to (8). If *small* $= i$, the exchange is performed, but has no effect on the array. Notice that in order to swap two elements, we need a temporary place to store one of them. Thus, we move the value

[1] We say "*a* smallest element" rather than "*the* smallest element" because there may be several occurrences of the smallest value. If so, we shall be happy with any of those occurrences.

[2] To describe a range of elements within an array, we adopt a convention from the language Pascal. If A is an array, then `A[i..j]` denotes those elements of A with indexes from i to j, inclusive.

```
    void SelectionSort(int A[], int n)
    {
        int i, j, small, temp;
(1)     for (i = 0; i < n-1; i++) {
            /* set small to the index of the first occur- */
            /* rence of the smallest element remaining */
(2)         small = i;
(3)         for (j = i+1; j < n; j++)
(4)             if (A[j] < A[small])
(5)                 small = j;
            /* when we reach here, small is the index of */
            /* the first smallest element in A[i..n-1]; */
            /* we now exchange A[small] with A[i] */
(6)         temp = A[small];
(7)         A[small] = A[i];
(8)         A[i] = temp;
        }
    }
```

Fig. 2.2. Iterative selection sort.

in `A[small]` to `temp` at line (6), move the value in `A[i]` to `A[small]` at line (7), and finally move the value originally in `A[small]` from `temp` to `A[i]` at line (8).

✦ **Example 2.3.** Let us study the behavior of `SelectionSort` on various inputs. First, let us look at what happens when we run `SelectionSort` on an array with no elements. When $n = 0$, the body of the for-loop of line (1) is not executed, so `SelectionSort` does "nothing" gracefully.

Now let us consider the case in which the array has only one element. Again, the body of the for-loop of line (1) is not executed. That response is satisfactory, because an array consisting of a single element is always sorted. The cases in which n is 0 or 1 are important boundary conditions, on which it is important to check the performance of any algorithm or program.

Finally, let us run `SelectionSort` on a small array with four elements, where `A[0]` through `A[3]` are

	0	1	2	3
A	40	30	20	10

We begin the outer loop with $i = 0$, and at line (2) we set `small` to 0. Lines (3) to (5) form an inner loop, in which `j` is set to 1, 2, and 3, in turn. With $j = 1$, the test of line (4) succeeds, since $A[1]$, which is 30, is less than $A[small]$, which is $A[0]$, or 40. Thus, we set `small` to 1 at line (5). At the second iteration of lines (3) to (5), with $j = 2$, the test of line (4) again succeeds, since $A[2] < A[1]$, and so we set `small` to 2 at line (5). At the last iteration of lines (3) to (5), with $j = 3$, the test of line (4) succeeds, since $A[3] < A[2]$, and we set `small` to 3 at line (5).

We now fall out of the inner loop to line (6). We set `temp` to 10, which is $A[small]$, then `A[3]` to $A[0]$, or 40, at line (7), and `A[0]` to 10 at line (8). Now, the

Sorting on Keys

When we sort, we apply a comparison operation to the values being sorted. Often the comparison is made only on specific parts of the values and the part used in the comparison is called the *key*.

For example, a course roster might be an array `A` of C structures of the form

```
struct STUDENT {
    int studentID;
    char *name;
    char grade;
} A[MAX];
```

We might want to sort by student ID, or name, or grade; each in turn would be the key. For example, if we wish to sort structures by student ID, we would use the comparison

```
A[j].studentID < A[small].studentID
```

at line (4) of `SelectionSort`. The type of array `A` and temporary `temp` used in the swap would be `struct STUDENT`, rather than `integer`. Note that entire structures are swapped, not just the key fields.

Since it is time-consuming to swap whole structures, a more efficient approach is to use a second array of pointers to `STUDENT` structures and sort only the pointers in the second array. The structures themselves remain stationary in the first array. We leave this version of selection sort as an exercise.

first iteration of the outer loop is complete, and array `A` appears as

	0	1	2	3
A	10	30	20	40

The second iteration of the outer loop, with $i = 1$, sets `small` to 1 at line (2). The inner loop sets `j` to 2 initially, and since $A[2] < A[1]$, line (5) sets `small` to 2. With $j = 3$, the test of line (4) fails, since $A[3] \geq A[2]$. Hence, *small* = 2 when we reach line (6). Lines (6) to (8) swap `A[1]` with `A[2]`, leaving the array

	0	1	2	3
A	10	20	30	40

Although the array now happens to be sorted, we still iterate the outer loop once more, with $i = 2$. We set `small` to 2 at line (2), and the inner loop is executed only with $j = 3$. Since the test of line (4) fails, *small* remains 2, and at lines (6) through (8), we "swap" `A[2]` with itself. The reader should check that the swapping has no effect when *small* = i. ✦

Figure 2.3 shows how the function `SelectionSort` can be used in a complete program to sort a sequence of n integers, provided that $n \leq 100$. Line (1) reads and stores n integers in an array `A`. If the number of inputs exceeds `MAX`, only the first `MAX` integers are put into `A`. A message warning the user that the number of inputs is too large would be useful here, but we omit it.

Line (3) calls SelectionSort to sort the array. Lines (4) and (5) print the integers in sorted order.

```
#include <stdio.h>

#define MAX 100
int A[MAX];
void SelectionSort(int A[], int n);

main()
{
    int i, n;
    /* read and store input in A */
(1) for (n = 0; n < MAX && scanf("%d", &A[n]) != EOF; n++)
(2)     ;
(3) SelectionSort(A,n); /* sort A */
(4) for (i = 0; i < n; i++)
(5)     printf("%d\n", A[i]); /* print A */
    }

void SelectionSort(int A[], int n)
{
    int i, j, small, temp;
    for (i = 0; i < n-1; i++) {
        small = i;
        for (j = i+1; j < n; j++)
                if (A[j] < A[small])
                    small = j;
        temp = A[small];
        A[small] = A[i];
        A[i] = temp;
    }
}
```

Fig. 2.3. A sorting program using selection sort.

EXERCISES

2.2.1: Simulate the function SelectionSort on an array containing the elements

a) 6, 8, 14, 17, 23
b) 17, 23, 14, 6, 8
c) 23, 17, 14, 8, 6

How many comparisons and swaps of elements are made in each case?

2.2.2**: What are the minimum and maximum number of (a) comparisons and (b) swaps that SelectionSort can make in sorting a sequence of n elements?

2.2.3: Write a C function that takes two linked lists of characters as arguments and returns `TRUE` if the first string precedes the second in lexicographic order. *Hint*: Implement the algorithm for comparing character strings that was described in this section. Use recursion by having the function call itself on the tails of the character strings when it finds that the first characters of both strings are the same. Alternatively, one can develop an iterative algorithm to do the same.

2.2.4*: Modify your program from Exercise 2.2.3 to ignore the case of letters in comparisons.

2.2.5: What does selection sort do if all elements are the same?

2.2.6: Modify Fig. 2.3 to perform selection sort when array elements are not integers, but rather structures of type `struct STUDENT`, as defined in the box "Sorting on Keys." Suppose that the key field is `studentID`.

2.2.7*: Further modify Fig. 2.3 so that it sorts elements of an arbitrary type T. You may assume, however, that there is a function *key* that takes an element of type T as argument and returns the key for that element, of some arbitrary type K. Also assume that there is a function *lt* that takes two elements of type K as arguments and returns `TRUE` if the first is "less than" the second, and `FALSE` otherwise.

2.2.8: Instead of using integer indexes into the array `A`, we could use pointers to integers to indicate positions in the array. Rewrite the selection sort algorithm of Fig. 2.3 using pointers.

2.2.9*: As mentioned in the box on "Sorting on Keys," if the elements to be sorted are large structures such as type `STUDENT`, it makes sense to leave them stationary in an array and sort pointers to these structures, found in a second array. Write this variation of selection sort.

2.2.10: Write an iterative program to print the distinct elements of an integer array.

2.2.11: Use the $\sum$ and $\prod$ notations described at the beginning of this chapter to express the following.

a) The sum of the odd integers from 1 to 377

b) The sum of the squares of the even integers from 2 to n (assume that n is even)

c) The product of the powers of 2 from 8 to 2^k

2.2.12: Show that when $small = i$, lines (6) through (8) of Fig. 2.2 (the swapping steps) do not have any effect on array `A`.

2.3 Inductive Proofs

Mathematical induction is a useful technique for proving that a statement $S(n)$ is true for all nonnegative integers n, or, more generally, for all integers at or above some lower limit. For example, in the introduction to this chapter we suggested that the statement $\sum_{i=1}^{n} i = n(n+1)/2$ can be proved true for all $n \geq 1$ by an induction on n.

Now, let $S(n)$ be some arbitrary statement about an integer n. In the simplest form of an inductive proof of the statement $S(n)$, we prove two facts:

Naming the Induction Parameter

It is often useful to explain an induction by giving the intuitive meaning of the variable n in the statement $S(n)$ that we are proving. If n has no special meaning, as in Example 2.4, we simply say "The proof is by induction on n." In other cases, n may have a physical meaning, as in Example 2.6, where n is the number of bits in the code words. There we can say, "The proof is by induction on the number of bits in the code words."

1. The *basis* case, which is frequently taken to be $S(0)$. However, the basis can be $S(k)$ for any integer k, with the understanding that then the statement $S(n)$ is proved only for $n \geq k$.

2. The *inductive step,* where we prove that for all $n \geq 0$ [or for all $n \geq k$, if the basis is $S(k)$], $S(n)$ implies $S(n+1)$. In this part of the proof, we assume that the statement $S(n)$ is true. $S(n)$ is called the *inductive hypothesis,* and assuming it to be true, we must then prove that $S(n+1)$ is true.

Inductive hypothesis

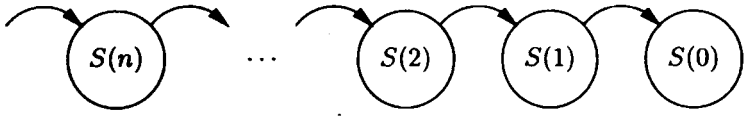


Fig. 2.4. In an inductive proof, each instance of the statement $S(n)$ is proved using the statement for the next lower value of n.

Figure 2.4 illustrates an induction starting at 0. For each integer n, there is a statement $S(n)$ to prove. The proof for $S(1)$ uses $S(0)$, the proof for $S(2)$ uses $S(1)$, and so on, as represented by the arrows. The way each statement depends on the previous one is uniform. That is, *by one proof of the inductive step, we prove each of the steps implied by the arrows in Fig. 2.4.*

✦ **Example 2.4.** As an example of mathematical induction, let us prove

STATEMENT $S(n)$: $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$ for any $n \geq 0$.

That is, the sum of the powers of 2, from the 0th power to the nth power, is 1 less than the $(n+1)$st power of 2.[3] For example, $1 + 2 + 4 + 8 = 16 - 1$. The proof proceeds as follows.

BASIS. To prove the basis, we substitute 0 for n in the equation $S(n)$. Then $S(n)$ becomes

[3] $S(n)$ can be proved without induction, using the formula for the sum of a geometric series. However, it will serve as a simple example of the technique of mathematical induction. Further, the proofs of the formulas for the sum of a geometric or arithmetic series that you have probably seen in high school are rather informal, and strictly speaking, mathematical induction should be used to prove those formulas.

$$\sum_{i=0}^{0} 2^i = 2^1 - 1 \tag{2.2}$$

There is only one term, for $i = 0$, in the summation on the left side of Equation (2.2), so that the left side of (2.2) sums to 2^0, or 1. The right side of Equation (2.2), which is $2^1 - 1$, or $2 - 1$, also has value 1. Thus we have proved the basis of $S(n)$; that is, we have shown that this equality is true for $n = 0$.

INDUCTION. Now we must prove the inductive step. We assume that $S(n)$ is true, and we prove the same equality with $n + 1$ substituted for n. The equation to be proved, $S(n + 1)$, is

$$\sum_{i=0}^{n+1} 2^i = 2^{n+2} - 1 \tag{2.3}$$

To prove Equation (2.3), we begin by considering the sum on the left side,

$$\sum_{i=0}^{n+1} 2^i$$

This sum is almost the same as the sum on the left side of $S(n)$, which is

$$\sum_{i=0}^{n} 2^i$$

except that (2.3) also has a term for $i = n + 1$, that is, the term 2^{n+1}

Since we are allowed to assume that the inductive hypothesis $S(n)$ is true in our proof of Equation (2.3), we should contrive to use $S(n)$ to advantage. We do so by breaking the sum in (2.3) into two parts, one of which is the sum in $S(n)$. That is, we separate out the last term, where $i = n + 1$, and write

$$\sum_{i=0}^{n+1} 2^i = \sum_{i=0}^{n} 2^i + 2^{n+1} \tag{2.4}$$

Now we can make use of $S(n)$ by substituting its right side, $2^{n+1} - 1$, for $\sum_{i=0}^{n} 2^i$ in Equation (2.4):

$$\sum_{i=0}^{n+1} 2^i = 2^{n+1} - 1 + 2^{n+1} \tag{2.5}$$

When we simplify the right side of Equation (2.5), it becomes $2 \times 2^{n+1} - 1$, or $2^{n+2} - 1$. Now we see that the summation on the left side of (2.5) is the same as the left side of (2.3), and the right side of (2.5) is equal to the right side of (2.3). We have thus proved the validity of Equation (2.3) by using the equality $S(n)$; that proof is the inductive step. The conclusion we draw is that $S(n)$ holds for every nonnegative value of n. ✦

Why Does Proof by Induction Work?

In an inductive proof, we first prove that $S(0)$ is true. Next we show that if $S(n)$ is true, then $S(n + 1)$ holds. But why can we then conclude that $S(n)$ is true for all $n \geq 0$? We shall offer two "proofs." A mathematician would point out that

Substituting for Variables

are often confused
expression involving

$$\sum_{i=0}^{m} 2^i = 2^{m+1} - 1$$

We then literally substitute the desired expression, $n + 1$ in this case, for each occurrence of m. That gives us

$$\sum_{i=0}^{n+1} 2^i = 2^{(n+1)+1} - 1$$

When we simplify $(n + 1) + 1$ to $n + 2$, we have (2.3).

Note that we should put parentheses around the expression substituted, to avoid accidentally changing the order of operations. For example, had we substituted $n + 1$ for m in the expression $2 \times m$, and not placed the parentheses around $n+1$, we would have gotten $2 \times n + 1$, rather than the correct expression $2 \times (n+1)$, which equals $2 \times n + 2$.

each of our "proofs" that induction works requires an inductive proof itself, and therefore is no proof at all. Technically, induction must be accepted as axiomatic. Nevertheless, many people find the following intuition useful.

In what follows, we assume that the basis value is $n = 0$. That is, we know that $S(0)$ is true and that for all n greater than 0, if $S(n)$ is true, then $S(n + 1)$ is true. Similar arguments work if the basis value is any other integer.

First "proof": *Iteration of the inductive step.* Suppose we want to show that $S(a)$ is true for a particular nonnegative integer a. If $a = 0$, we just invoke the truth of the basis, $S(0)$. If $a > 0$, then we argue as follows. We know that $S(0)$ is true, from the basis. The statement "$S(n)$ implies $S(n + 1)$," with 0 in place of n, says "$S(0)$ implies $S(1)$." Since we know that $S(0)$ is true, we now know that $S(1)$ is true. Similarly, if we substitute 1 for n, we get "$S(1)$ implies $S(2)$," and so we also know that $S(2)$ is true. Substituting 2 for n, we have "$S(2)$ implies $S(3)$," so that $S(3)$ is true, and so on. No matter what the value of a is, we eventually get to $S(a)$, and we are done.

Second "proof": *Least counterexample.* Suppose $S(n)$ were not true for at least one value of n. Let a be the least nonnegative integer for which $S(a)$ is false. If $a = 0$, then we contradict the basis, $S(0)$, and so a must be greater than 0. But if $a > 0$, and a is the least nonnegative integer for which $S(a)$ is false, then $S(a - 1)$ must be true. Now, the inductive step, with n replaced by $a-1$, tells us that $S(a-1)$ implies $S(a)$. Since $S(a-1)$ is true, $S(a)$ must be true, another contradiction. Since we assumed there were nonnegative values of n for which $S(n)$ is false and derived a contradiction, $S(n)$ must therefore be true for any $n \geq 0$.

Error-Detecting Codes

We shall now begin an extended example of "error-detecting codes," a concept that is interesting in its own right and also leads to an interesting inductive proof. When we transmit information over a data network, we code characters (letters, digits, punctuation, and so on) into strings of bits, that is, 0's and 1's. For the moment let us assume that characters are represented by seven bits. However, it is normal to transmit more than seven bits per character, and an eighth bit can be used to help detect some simple errors in transmission. That is, occasionally, one of the 0's or 1's gets changed because of noise during transmission, and is received as the opposite bit; a 0 entering the transmission line emerges as a 1, or vice versa. It is useful if the communication system can tell when one of the eight bits has been changed, so that it can signal for a retransmission.

To detect changes in a single bit, we must be sure that no two characters are represented by sequences of bits that differ in only one position. For then, if that position were changed, the result would be the code for the other character, and we could not detect that an error had occurred. For example, if the code for one character is the sequence of bits 01010101, and the code for another is 01000101, then a change in the fourth position from the left turns the former into the latter.

Parity bit

One way to be sure that no characters have codes that differ in only one position is to precede the conventional 7-bit code for the character by a *parity bit*. If the total number of 1's in a group of bits is odd, the group is said to have *odd parity*. If the number of 1's in the group is even, then the group has *even parity*. The coding scheme we select is to represent each character by an 8-bit code with even parity; we could as well have chosen to use only the codes with odd parity. We force the parity to be even by selecting the parity bit judiciously.

ASCII

✦ **Example 2.5.** The conventional ASCII (pronounced "ask-ee"; it stands for "American Standard Code for Information Interchange") 7-bit code for the character `A` is 1000001. That sequence of seven bits already has an even number of 1's, and so we prefix it by 0 to get 01000001. The conventional code for `C` is 1000011, which differs from the 7-bit code for `A` only in the sixth position. However, this code has odd parity, and so we prefix a 1 to it, yielding the 8-bit code 11000011 with even parity. Note that after prefixing the parity bits to the codes for `A` and `C`, we have 01000001 and 11000011, which differ in two positions, namely the first and seventh, as seen in Fig. 2.5. ✦

`A:` 0 1 0 0 0 0 0 1

`C:` 1 1 0 0 0 0 1 1

Fig. 2.5. We can choose the initial parity bit so the 8-bit code always has even parity.

We can always pick a parity bit to attach to a 7-bit code so that the number of 1's in the 8-bit code is even. We pick parity bit 0 if the 7-bit code for the character at hand has even parity, and we pick parity bit 1 if the 7-bit code has odd parity. In either case, the number of 1's in the 8-bit code is even.

No two sequences of bits that each have even parity can differ in only one position. For if two such bit sequences differ in exactly one position, then one has exactly one more 1 than the other. Thus, one sequence must have odd parity and the other even parity, contradicting our assumption that both have even parity. We conclude that addition of a parity bit to make the number of 1's even serves to create an error-detecting code for characters.

The parity-bit scheme is quite "efficient," in the sense that it allows us to transmit many different characters. Note that there are 2^n different sequences of n bits, since we may choose either of two values (0 or 1) for the first position, either of two values for the second position, and so on, a total of $2 \times 2 \times \cdots \times 2$ (n factors) possible strings. Thus, we might expect to be able to represent up to $2^8 = 256$ characters with eight bits.

However, with the parity scheme, we can choose only seven of the bits; the eighth is then forced upon us. We can thus represent up to 2^7, or 128 characters, and still detect single errors. That is not so bad; we can use 128/256, or half, of the possible 8-bit codes as legal codes for characters, and still detect an error in one bit.

Similarly, if we use sequences of n bits, choosing one of them to be the parity bit, we can represent 2^{n-1} characters by taking sequences of $n-1$ bits and prefixing the suitable parity bit, whose value is determined by the other $n-1$ bits. Since there are 2^n sequences of n bits, we can represent $2^{n-1}/2^n$, or half the possible number of characters, and still detect an error in any one of the bits of a sequence.

Is it possible to detect errors and use more than half the possible sequences of bits as legal codes? Our next example tells us we cannot. The inductive proof uses a statement that is not true for 0, and for which we must choose a larger basis, namely 1.

✦ **Example 2.6.** We shall prove the following by induction on n.

Error-detecting code

STATEMENT $S(n)$: If C is any set of bit strings of length n that is *error detecting* (i.e., if there are no two strings that differ in exactly one position), then C contains at most 2^{n-1} strings.

This statement is not true for $n = 0$. $S(0)$ says that any error-detecting set of strings of length 0 has at most 2^{-1} strings, that is, half a string. Technically, the set C consisting of only the empty string (string with no positions) is an error-detecting set of length 0, since there are no two strings in C that differ in only one position. Set C has more than half a string; it has one string to be exact. Thus, $S(0)$ is false. However, for all $n \geq 1$, $S(n)$ is true, as we shall see.

BASIS. The basis is $S(1)$; that is, any error-detecting set of strings of length one has at most $2^{1-1} = 2^0 = 1$ string. There are only two bit strings of length one, the string 0 and the string 1. However, we cannot have both of them in an error-detecting set, because they differ in exactly one position. Thus, every error-detecting set for $n = 1$ must have at most one string.

INDUCTION. Let $n \geq 1$, and assume that the inductive hypothesis — an error-detecting set of strings of length n has at most 2^{n-1} strings — is true. We must

show, using this assumption, that any error-detecting set C of strings with length $n+1$ has at most 2^n strings. Thus, divide C into two sets, C_0, the set of strings in C that begin with 0, and C_1, the set of strings in C that begin with 1. For instance, suppose $n = 2$ and C is the code with strings of length $n+1 = 3$ constructed using a parity bit. Then, as shown in Fig. 2.6, C consists of the strings 000, 101, 110, and 011; C_0 consists of the strings 000 and 011, and C_1 has the other two strings, 101 and 110.

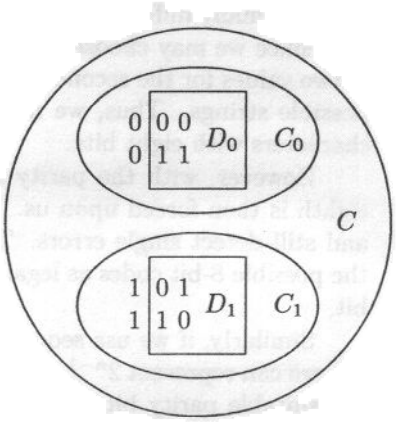


Fig. 2.6. The set C is split into C_0, the strings beginning with 0, and C_1, the strings beginning with 1. D_0 and D_1 are formed by deleting the leading 0's and 1's, respectively.

Consider the set D_0 consisting of those strings in C_0 with the leading 0 removed. In our example above, D_0 contains the strings 00 and 11. We claim that D_0 cannot have two strings differing in only one bit. The reason is that if there are two such strings — say $a_1a_2\cdots a_n$ and $b_1b_2\cdots b_n$ — then restoring their leading 0's gives us two strings in C_0, $0a_1a_2\cdots a_n$ and $0b_1b_2\cdots b_n$, and these strings would differ in only one position as well. But strings in C_0 are also in C, and we know that C does not have two strings that differ in only one position. Thus, neither does D_0, and so D_0 is an error detecting set.

Now we can apply the inductive hypothesis to conclude that D_0, being an error-detecting set with strings of length n, has at most 2^{n-1} strings. Thus, C_0 has at most 2^{n-1} strings.

We can reason similarly about the set C_1. Let D_1 be the set of strings in C_1, with their leading 1's deleted. D_1 is an error-detecting set with strings of length n, and by the inductive hypothesis, D_1 has at most 2^{n-1} strings. Thus, C_1 has at most 2^{n-1} strings. However, every string in C is in either C_0 or C_1. Therefore, C has at most $2^{n-1} + 2^{n-1}$, or 2^n strings.

We have proved that $S(n)$ implies $S(n+1)$, and so we may conclude that $S(n)$ is true for all $n \geq 1$. We exclude $n = 0$ from the claim, because the basis is $n = 1$, not $n = 0$. We now see that the error-detecting sets constructed by parity check are as large as possible, since they have exactly 2^{n-1} strings when strings of n bits are used. ✦

How to Invent Inductive Proofs

There is no "crank to turn" that is guaranteed to give you an inductive proof of any (true) statement $S(n)$. Finding inductive proofs, like finding proofs of any kind, or like writing programs that work, is a task with intellectual challenge, and we can only offer a few words of advice. If you examine the inductive steps in Examples 2.4 and 2.6, you will notice that in each case we had to rework the statement $S(n+1)$ that we were trying to prove so that it incorporated the inductive hypothesis, $S(n)$, plus something extra. In Example 2.4, we expressed the sum

$$1+2+4+\cdots+2^n+2^{n+1}$$

as the sum

$$1+2+4+\cdots+2^n$$

which the inductive hypothesis tells us something about, plus the extra term, 2^{n+1}.

In Example 2.6, we expressed the set C, with strings of length $n+1$, in terms of two sets of strings (which we called D_0 and D_1) of length n, so that we could apply the inductive hypothesis to these sets and conclude that both of these sets were of limited size.

Of course, working with the statement $S(n+1)$ so that we can apply the inductive hypothesis is just a special case of the more universal problem-solving adage "Use what is given." The hard part always comes when we must deal with the "extra" part of $S(n+1)$ and complete the proof of $S(n+1)$ from $S(n)$. However, the following is a universal rule:

- An inductive proof must at some point say "···and by the inductive hypothesis we know that···." If it doesn't, then it isn't a inductive proof.

EXERCISES

2.3.1: Show the following formulas by induction on n starting at $n=1$.

a) $\sum_{i=1}^{n} i = n(n+1)/2$.

b) $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$.

c) $\sum_{i=1}^{n} i^3 = n^2(n+1)^2/4$.

d) $\sum_{i=1}^{n} 1/i(i+1) = n/(n+1)$.

Triangular number

2.3.2: Numbers of the form $t_n = n(n+1)/2$ are called *triangular numbers*, because marbles arranged in an equilateral triangle, n on a side, will total $\sum_{i=1}^{n} i$ marbles, which we saw in Exercise 2.3.1(a) is t_n marbles. For example, bowling pins are arranged in a triangle 4 on a side and there are $t_4 = 4 \times 5/2 = 10$ pins. Show by induction on n that $\sum_{j=1}^{n} t_j = n(n+1)(n+2)/6$.

2.3.3: Identify the parity of each of the following bit sequences as even or odd:

a) 01101

b) 111000111

c) 010101

2.3.4: Suppose we use three digits — say 0, 1, and 2 — to code symbols. A set of strings C formed from 0's, 1's, and 2's is *error detecting* if no two strings in C differ in only one position. For example, $\{00, 11, 22\}$ is an error-detecting set with strings of length two, using the digits 0, 1, and 2. Show that for any $n \geq 1$, an error-detecting set of strings of length n using the digits 0, 1, and 2, cannot have more than 3^{n-1} strings.

2.3.5*: Show that for any $n \geq 1$, there is an error-detecting set of strings of length n, using the digits 0, 1, and 2, that has 3^{n-1} strings.

2.3.6*: Show that if we use k symbols, for any $k \geq 2$, then there is an error-detecting set of strings of length n, using k different symbols as "digits," with k^{n-1} strings, but no such set of strings with more than k^{n-1} strings.

2.3.7*: If $n \geq 1$, the number of strings using the digits 0, 1, and 2, with no two consecutive places holding the same digit, is $3 \times 2^{n-1}$. For example, there are 12 such strings of length three: 010, 012, 020, 021, 101, 102, 120, 121, 201, 202, 210, and 212. Prove this claim by induction on the length of the strings. Is the formula true for $n = 0$?

2.3.8*: Prove that the ripple-carry addition algorithm discussed in Section 1.3 produces the correct answer. *Hint*: Show by induction on i that after considering the first i places from the right end, the sum of the tails of length i for the two addends equals the number whose binary representation is the carry bit followed by the i bits of answer generated so far.

2.3.9*: The formula for the sum of n terms of a geometric series $a, ar, ar^2, \ldots, ar^{n-1}$ is

$$\sum_{i=0}^{n-1} ar^i = \frac{(ar^n - a)}{(r-1)}$$

Prove this formula by induction on n. Note that you must assume $r \neq 1$ for the formula to hold. Where do you use that assumption in your proof?

2.3.10: The formula for the sum of an arithmetic series with first term a and increment b, that is, $a, (a+b), (a+2b), \ldots, \big(a+(n-1)b\big)$, is

$$\sum_{i=0}^{n-1} a + bi = n\big(2a + (n-1)b\big)/2$$

a) Prove this formula by induction on n.

b) Show how Exercise 2.3.1(a) is an example of this formula.

2.3.11: Give two informal proofs that induction starting at 1 "works," although the statement $S(0)$ may be false.

2.3.12: Show by induction on the length of strings that the code consisting of the odd-parity strings detects errors.

Arithmetic and Geometric Sums

There are two formulas from high-school algebra that we shall use frequently. They each have interesting inductive proofs, which we ask the reader to provide in Exercises 2.3.9 and 2.3.10.

Arithmetic series

An *arithmetic series* is a sequence of n numbers of the form

$$a, (a+b), (a+2b), \ldots, \big(a+(n-1)b\big)$$

The first term is a, and each term is b larger than the one before. The sum of these n numbers is n times the average of the first and last terms; that is:

$$\sum_{i=0}^{n-1} a+bi = n\big(2a+(n-1)b\big)/2$$

For example, consider the sum of $3+5+7+9+11$. There are $n=5$ terms, the first is 3 and the last 11. Thus, the sum is $5 \times (3+11)/2 = 5 \times 7 = 35$. You can check that this sum is correct by adding the five integers.

Geometric series

A *geometric series* is a sequence of n numbers of the form

$$a, ar, ar^2, ar^3, \ldots, ar^{n-1}$$

That is, the first term is a, and each successive term is r times the previous term. The formula for the sum of n terms of a geometric series is

$$\sum_{i=0}^{n-1} ar^i = \frac{(ar^n - a)}{(r-1)}$$

Here, r can be greater or less than 1. If $r=1$, the above formula does not work, but all terms are a so the sum is obviously an.

As an example of a geometric series sum, consider $1+2+4+8+16$. Here, $n=5$, the first term a is 1, and the ratio r is 2. Thus, the sum is

$$(1 \times 2^5 - 1)/(2-1) = (32-1)/1 = 31$$

as you may check. For another example, consider $1+1/2+1/4+1/8+1/16$. Again $n=5$ and $a=1$, but $r=1/2$. The sum is

$$\big(1 \times (\tfrac{1}{2})^5 - 1\big)/(\tfrac{1}{2}-1) = (-31/32)/(-1/2) = 1\tfrac{15}{16}$$

Error-correcting code

2.3.13**: If no two strings in a code differ in fewer than three positions, then we can actually correct a single error, by finding the unique string in the code that differs from the received string in only one position. It turns out that there is a code of 7-bit strings that corrects single errors and contains 16 strings. Find such a code. *Hint*: Reasoning it out is probably best, but if you get stuck, write a program that searches for such a code.

2.3.14*: Does the even parity code detect any "double errors," that is, changes in two different bits? Can it correct any single errors?

Template for Simple Inductions

Let us summarize Section 2.3 by giving a template into which the simple inductions of that section fit. Section 2.4 will cover a more general template.

1. Specify the statement $S(n)$ to be proved. Say you are going to prove $S(n)$ by induction on n, for all $n \geq i_0$. Here, i_0 is the constant of the basis; usually i_0 is 0 or 1, but it could be any integer. Explain intuitively what n means, e.g., the length of codewords.
2. State the basis case, $S(i_0)$.
3. Prove the basis case. That is, explain why $S(i_0)$ is true.
4. Set up the inductive step by stating that you are assuming $S(n)$ for some $n \geq i_0$, the "inductive hypothesis." Express $S(n+1)$ by substituting $n+1$ for n in the statement $S(n)$.
5. Prove $S(n+1)$, assuming the inductive hypothesis $S(n)$.
6. Conclude that $S(n)$ is true for all $n \geq i_0$ (but not necessarily for smaller n).

2.4 Complete Induction

In the examples seen so far, we have proved that $S(n+1)$ is true using only $S(n)$ as an inductive hypothesis. However, since we prove our statement S for values of its parameter starting at the basis value and proceeding upward, we are entitled to use $S(i)$ for all values of i, from the basis value up to n. This form of induction is called *complete* (or sometimes *perfect* or *strong*) *induction*, while the simple form of induction of Section 2.3, where we used only $S(n)$ to prove $S(n+1)$ is sometimes called *weak* induction.

Strong and weak induction

Let us begin by considering how to perform a complete induction starting with basis $n = 0$. We prove that $S(n)$ is true for all $n \geq 0$ in two steps:

1. We first prove the basis, $S(0)$.
2. As an inductive hypothesis, we assume all of $S(0), S(1), \ldots, S(n)$ to be true. From these statements we prove that $S(n+1)$ holds.

As for weak induction described in the previous section, we can also pick some value a other than 0 as the basis. Then, for the basis we prove $S(a)$, and in the inductive step we are entitled to assume only $S(a), S(a+1), \ldots, S(n)$. Note that weak induction is a special case of complete induction in which we elect not to use any of the previous statements except $S(n)$ to prove $S(n+1)$.

Figure 2.7 suggests how complete induction works. Each instance of the statement $S(n)$ can (optionally) use any of the lower-indexed instances to its right in its proof.

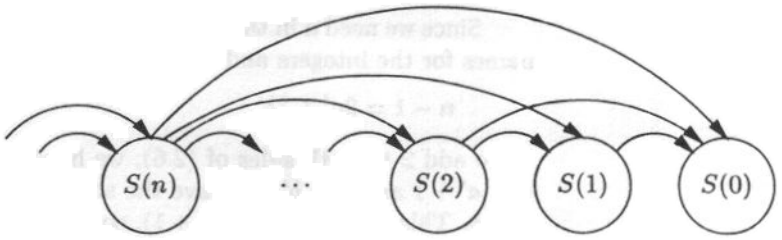


Fig. 2.7. Complete induction allows each instance to use one, some, or all of the previous instances in its proof.

Inductions With More Than One Basis Case

When performing a complete induction, there are times when it is useful to have more than one basis case. If we wish to prove a statement $S(n)$ for all $n \geq i_0$, then we could treat not only i_0 as a basis case, but also some number of consecutive integers above i_0, say $i_0, i_0 + 1, i_0 + 2, \ldots, j_0$. Then we must do the following two steps:

1. Prove each of the basis cases, the statements $S(i_0), S(i_0 + 1), \ldots, S(j_0)$.
2. As an inductive hypothesis, assume all of $S(i_0), S(i_0 + 1), \ldots, S(n)$ hold, for some $n \geq j_0$, and prove $S(n + 1)$.

✦ **Example 2.7.** Our first example of a complete induction is a simple one that uses multiple basis cases. As we shall see, it is only "complete" in a limited sense. To prove $S(n + 1)$ we do not use $S(n)$ but we use $S(n - 1)$ only. In more general complete inductions to follow, we use $S(n)$, $S(n - 1)$, and many other instances of the statement S.

Let us prove by induction on n the following statement for all $n \geq 0$.[4]

STATEMENT $S(n)$: There are integers a and b (positive, negative, or 0) such that $n = 2a + 3b$.

BASIS. We shall take both 0 and 1 as basis cases.

i) For $n = 0$ we may pick $a = 0$ and $b = 0$. Surely $0 = 2 \times 0 + 3 \times 0$.

ii) For $n = 1$, pick $a = -1$ and $b = 1$. Then $1 = 2 \times (-1) + 3 \times 1$.

INDUCTION. Now, we may assume $S(n)$ and prove $S(n + 1)$, for any $n \geq 1$. Note that we may assume n is at least the largest of the consecutive values for which we have proved the basis: $n \geq 1$ here. Statement $S(n + 1)$ says that $n + 1 = 2a + 3b$ for some integers a and b.

The inductive hypothesis says that all of $S(0), S(1), \ldots, S(n)$ are true. Note that we begin the sequence at 0 because that was the lowest of the consecutive basis cases. Since $n \geq 1$ can be assumed, we know that $n - 1 \geq 0$, and therefore, $S(n - 1)$ is true. This statement says that there are integers a and b such that $n - 1 = 2a + 3b$.

[4] Actually, this statement is true for all n, positive or negative, but the case of negative n requires a second induction which we leave as an exercise.

Since we need a in the statement $S(n+1)$, let us restate $S(n-1)$ to use different names for the integers and say there are integers a' and b' such that

$$n - 1 = 2a' + 3b' \tag{2.6}$$

If we add 2 to both sides of (2.6), we have $n + 1 = 2(a' + 1) + 3b'$. If we then let $a = a' + 1$ and $b = b'$, we have the statement $n + 1 = 2a + 3b$ for some integers a and b. This statement is $S(n + 1)$, so we have proved the induction. Notice that in this proof, we did not use $S(n)$, but we did use $S(n - 1)$. ✦

Justifying Complete Induction

Like the ordinary or "weak" induction discussed in Section 2.3, complete induction can be justified intuitively as a proof technique by a "least counterexample" argument. Let the basis cases be $S(i_0), S(i_0+1), \ldots, S(j_0)$, and suppose we have shown that for any $n \geq j_0$, $S(i_0), S(i_0+1), \ldots, S(n)$ together imply $S(n+1)$. Now, suppose $S(n)$ were not true for at least one value of $n \geq i_0$, and let b be the smallest integer equal to or greater than i_0 for which $S(b)$ is false. Then b cannot be between i_0 and j_0, or the basis is contradicted. Further, b cannot be greater than j_0. If it were, all of $S(i_0), S(i_0 + 1), \ldots, S(b - 1)$ would be true. But the inductive step would then tell us that $S(b)$ is true, yielding the contradiction.

Normal Forms for Arithmetic Expressions

We shall now explore an extended example concerning the transformation of arithmetic expressions to equivalent forms. It offers an illustration of a complete induction that takes full advantage of the fact that the statement S to be proved may be assumed for all arguments from n downward.

By way of motivation, a compiler for a programming language may take advantage of the algebraic properties of arithmetic operators to rearrange the order in which the operands of an arithmetic expression are evaluated. The goal is of this rearrangement is to find a way for the computer to evaluate the expression using less time than the obvious evaluation order takes.

In this section we consider arithmetic expressions containing a single associative and commutative operator, like +, and examine what rearrangements of operands are possible. We shall prove that if we have any expression involving only the operator +, then the value of the expression is equal to the value of any other expression with + applied to the same operands, ordered and/or grouped in any arbitrary way. For example,

$$\big(a_3 + (a_4 + a_1)\big) + (a_2 + a_5) = a_1 + \Big(a_2 + \big(a_3 + (a_4 + a_5)\big)\Big)$$

We shall prove this claim by performing two separate inductions, the first of which is a complete induction.

✦ **Example 2.8.** We shall prove by complete induction on n (the number of operands in an expression) the statement

Associativity and Commutativity

Associative law

Recall that the *associative law* for addition says that we can add three values either by adding the first two and then adding the third to the result, or by adding the first to the result of adding the second and third; the result will be the same. Formally,

$$(E_1 + E_2) + E_3 = E_1 + (E_2 + E_3)$$

where E_1, E_2, and E_3 are any arithmetic expressions. For instance,

$$(1 + 2) + 3 = 1 + (2 + 3)$$

Here, $E_1 = 1$, $E_2 = 2$, and $E_3 = 3$. Also,

$$((xy) + (3z - 2)) + (y + z) = xy + ((3z - 2) + (y + z))$$

Here, $E_1 = xy$, $E_2 = 3z - 2$, and $E_3 = y + z$.

Commutative law

Also recall that the *commutative law* for addition says that we can sum two expressions in either order. Formally,

$$E_1 + E_2 = E_2 + E_1$$

For example, $1 + 2 = 2 + 1$, and $xy + (3z - 2) = (3z - 2) + xy$.

STATEMENT $S(n)$: If E is an expression involving the operator $+$ and n operands, and a is one of those operands, then E can be transformed, by using the associative and commutative laws, into an expression of the form $a + F$, where F is an expression involving all the operands of E except a, grouped in some order using the operator $+$.

Statement $S(n)$ only holds for $n \geq 2$, since there must be at least one occurrence of the operator $+$ in E. Thus, we shall use $n = 2$ as our basis.

BASIS. Let $n = 2$. Then E can be only $a + b$ or $b + a$, for some operand b other than a. In the first case, we let F be the expression b, and we are done. In the second case, we note that by the commutative law, $b + a$ can be transformed into $a + b$, and so we may again let $F = b$.

INDUCTION. Let E have $n + 1$ operands, and assume that $S(i)$ is true for $i = 2, 3, \ldots, n$. We need to prove the inductive step for $n \geq 2$, so we may assume that E has at least three operands and therefore at least two occurrences of $+$. We can write E as $E_1 + E_2$ for some expressions E_1 and E_2. Since E has exactly $n + 1$ operands, and E_1 and E_2 must each have at least one of these operands, it follows that neither E_1 nor E_2 can have more than n operands. Thus, the inductive hypothesis applies to E_1 and E_2, as long as they have more than one operand each (because we started with $n = 2$ as the basis). There are four cases we must consider, depending whether a is in E_1 or E_2, and on whether it is or is not the only operand in E_1 or E_2.

a) E_1 is a by itself. An example of this case occurs when E is $a + (b + c)$; here E_1 is a and E_2 is $b + c$. In this case, E_2 serves as F; that is, E is already of the form $a + F$.

b) E_1 has more than one operand, and a is among them. For instance,

$$E = \big(c + (d + a)\big) + (b + e)$$

where $E_1 = c + (d + a)$ and $E_2 = b + e$. Here, since E_1 has no more than n operands but at least two operands, we can apply the inductive hypothesis to tell us that E_1 can be transformed, using the commutative and associative laws, into $a + E_3$. Thus, E can be transformed into $(a + E_3) + E_2$. We apply the associative law and see that E can further be transformed into $a + (E_3 + E_2)$. Thus, we may choose F to be $E_3 + E_2$, which proves the inductive step in this case. For our example E above, we may suppose that $E_2 = c + (d + a)$ is transformed by the inductive hypothesis into $a + (c + d)$. Then E can be regrouped into $a + \big((c + d) + (b + e)\big)$.

c) E_2 is a alone. For instance, $E = (b+c)+a$. In this case, we use the commutative law to transform E into $a + E_1$, which is of the desired form if we let F be E_1.

d) E_2 has more than one operand, including a. An example is $E = b + (a + c)$. Apply the commutative law to transform E into $E_2 + E_1$. Then proceed as in case (b). If $E = b + (a + c)$, we transform E first into $(a + c) + b$. By the inductive hypothesis, $a + c$ can be put in the desired form; in fact, it is already there. The associative law then transforms E into $a + (c + b)$.

In all four cases, we have transformed E to the desired form. Thus, the inductive step is proved, and we conclude that $S(n)$ for all $n \geq 2$. ✦

✦ **Example 2.9.** The inductive proof of Example 2.8 leads directly to an algorithm that puts an expression into the desired form. As an example, consider the expression

$$E = \big(x + (z + v)\big) + (w + y)$$

and suppose that v is the operand we wish to "pull out," that is, to play the role of a in the transformation of Example 2.8. Initially, we have an example of case (b), with $E_1 = x + (z + v)$, and $E_2 = w + y$.

Next, we must work on the expression E_1 and "pull out" v. E_1 is an example of case (d), and so we first apply the commutative law to transform it into $(z + v) + x$. As an instance of case (b), we must work on the expression $z + v$, which is an instance of case (c). We thus transform it by the commutative law into $v + z$.

Now E_1 has been transformed into $(v+z)+x$, and a further use of the associative law transforms it to $v+(z+x)$. That, in turn, transforms E into $\big(v+(z+x)\big)+(w+y)$. By the associative law, E can be transformed into $v + \big((z + x) + (w + y)\big)$. Thus, $E = v + F$, where F is the expression $(z + x) + (w + y)$. The entire sequence of transformations is summarized in Fig. 2.8. ✦

Now, we can use the statement proved in Example 2.8 to prove our original contention, that any two expressions involving the operator + and the same list of distinct operands can be transformed one to the other by the associative and commutative laws. This proof is by weak induction, as discussed in Section 2.3, rather than complete induction.

$$(x + (z + v)) + (w + y)$$
$$((z + v) + x) + (w + y)$$
$$((v + z) + x) + (w + y)$$
$$(v + (z + x)) + (w + y)$$
$$v + ((z + x) + (w + y))$$

Fig. 2.8. Using the commutative and associative laws, we can "pull out" any operand, such as v.

✦ **Example 2.10.** Let us prove the following statement by induction on n, the number of operands in an expression.

STATEMENT $T(n)$: If E and F are expressions involving the operator $+$ and the same set of n distinct operands, then it is possible to transform E into F by a sequence of applications of the associative and commutative laws.

BASIS. If $n = 1$, then the two expressions must both be a single operand a. Since they are the same expression, surely E is "transformable" into F.

INDUCTION. Suppose $T(n)$ is true, for some $n \geq 1$. We shall now prove $T(n+1)$. Let E and F be expressions involving the same set of $n+1$ operands, and let a be one of these operands. Since $n+1 \geq 2$, $S(n+1)$ — the statement from Example 2.8 — must hold. Thus, we can transform E into $a + E_1$ for some expression E_1 involving the other n operands of E. Similarly, we can transform F into $a + F_1$, for some expression F_1 involving the same n operands as E_1. What is more important, in this case, is that we can also perform the transformations in the opposite direction, transforming $a + F_1$ into F by use of the associative and commutative laws.

Now we invoke the inductive hypothesis $T(n)$ on the expressions E_1 and F_1. Each has the same n operands, and so the inductive hypothesis applies. That tells us we can transform E_1 into F_1, and therefore we can transform $a + E_1$ into $a + F_1$. We may thus perform the transformations

$$\begin{aligned} E &\to \cdots \to a + E_1 && \text{Using } S(n+1) \\ &\to \cdots \to a + F_1 && \text{Using } T(n) \\ &\to \cdots \to F && \text{Using } S(n+1) \text{ in reverse} \end{aligned}$$

to turn E into F. ✦

✦ **Example 2.11.** Let us transform $E = (x+y)+(w+z)$ into $F = \big((w+z)+y\big)+x$. We begin by selecting an operand, say w, to "pull out." If we check the cases in Example 2.8, we see that for E we perform the sequence of transformations

$$(x + y) + (w + z) \to (w + z) + (x + y) \to w + \big(z + (x + y)\big) \tag{2.7}$$

while for F we do

$$\big((w + z) + y\big) + x \to \big(w + (z + y)\big) + x \to w + \big((z + y) + x\big) \tag{2.8}$$

We now have the subproblem of transforming $z + (x + y)$ into $(z + y) + x$. We shall do so by "pulling out" x. The sequences of transformations are

$$z + (x + y) \to (x + y) + z \to x + (y + z) \tag{2.9}$$

and

$$(z + y) + x \to x + (z + y) \tag{2.10}$$

That, in turn, gives us a subproblem of transforming $y + z$ into $z + y$. We do so by an application of the commutative law. Strictly speaking, we use the technique of Example 2.8 to "pull out" y for each, leaving $y + z$ for each expression. Then the basis case for Example 2.10 tells us that the expression z can be "transformed" into itself.

We can now transform $z + (x + y)$ into $(z + y) + x$ by the steps of line (2.9), then applying the commutative law to subexpression $y + z$, and finally using the transformation of line (2.10), in reverse. We use these transformations as the middle part of the transformation from $(x + y) + (w + z)$ to $\big((w + z) + y\big) + x$. First we apply the transformations of line (2.7), and then the transformations just discussed to change $z + (x + y)$ into $(z + y) + x$, and finally the transformations of line (2.8) in reverse. The entire sequence of transformations is summarized in Fig. 2.9. ✦

$(x + y) + (w + z)$	Expression E
$(w + z) + (x + y)$	Middle of (2.7)
$w + \big(z + (x + y)\big)$	End of (2.7)
$w + \big((x + y) + z\big)$	Middle of (2.9)
$w + \big(x + (y + z)\big)$	End of (2.9)
$w + \big(x + (z + y)\big)$	Commutative law
$w + \big((z + y) + x\big)$	(2.10) in reverse
$\big(w + (z + y)\big) + x$	Middle of (2.8) in reverse
$\big((w + z) + y\big) + x$	Expression F, end of (2.8) in reverse

Fig. 2.9. Transforming one expression into another using the commutative and associative laws.

EXERCISES

2.4.1: "Pull out" from the expression $E = (u + v) + \Big(\big(w + (x + y)\big) + z\Big)$ each of the operands in turn. That is, start from E in each of the six parts, and use the techniques of Example 2.8 to transform E into an expression of the form $u + E_1$. Then transform E_1 into an expression of the form $v + E_2$, and so on.

2.4.2: Use the technique of Example 2.10 to transform

a) $w + \big(x + (y + z)\big)$ into $\big((w + x) + y\big) + z$
b) $(v + w) + \big((x + y) + z\big)$ into $\big((y + w) + (v + z)\big) + x$

2.4.3*: Let E be an expression with operators $+$, $-$, $*$, and $/$; each operator is binary only; that is, it takes two operands. Show, using a complete induction on the number of occurrences of operators in E, that if E has n operator occurrences, then E has $n + 1$ operands.

Binary operator

A Template for All Inductions

The following organization of inductive proofs covers complete inductions with multiple basis cases. As a special case it includes the weak inductions of Section 2.3, and it includes the common situation where there is only one basis case.

1. Specify the statement $S(n)$ to be proved. Say that you are going to prove $S(n)$ by induction on n, for $n \geq i_0$. Specify what i_0 is; often it is 0 or 1, but i_0 could be any integer. Explain intuitively what n represents.
2. State the basis case(s). These will be all the integers from i_0 up to some integer j_0. Often $j_0 = i_0$, but j_0 could be larger.
3. Prove each of the basis cases $S(i_0), S(i_0 + 1), \ldots, S(j_0)$.
4. Set up the inductive step by stating that you are assuming

   $$S(i_0), S(i_0 + 1), \ldots, S(n)$$

   (the "inductive hypothesis") and that you want to prove $S(n + 1)$. State that you are assuming $n \geq j_0$; that is, n is at least as great as the highest basis case. Express $S(n + 1)$ by substituting $n + 1$ for n in the statement $S(n)$.
5. Prove $S(n + 1)$ under the assumptions mentioned in (4). If the induction is a weak, rather than complete, induction, then only $S(n)$ will be used in the proof, but you are free to use any or all of the statements of the inductive hypothesis.
6. Conclude that $S(n)$ is true for all $n \geq i_0$ (but not necessarily for smaller n).

2.4.4: Give an example of a binary operator that is commutative but not associative.

2.4.5: Give an example of a binary operator that is associative but not commutative.

2.4.6*: Consider an expression E whose operators are all binary. The *length* of E is the number of symbols in E, counting an operator or a left or right parenthesis as one symbol, and also counting any operand such as 123 or `abc` as one symbol. Prove that E must have an odd length. *Hint*: Prove the claim by complete induction on the length of the expression E.

2.4.7: Show that every negative integer can be written in the form $2a + 3b$ for some (not necessarily positive) integers a and b.

2.4.8*: Show that every integer (positive or negative) can be written in the form $5a + 7b$ for some (not necessarily positive) integers a and b.

2.4.9*: Is every proof by weak induction (as in Section 2.3) also a proof by complete induction? Is every proof by complete induction also a proof by weak induction?

2.4.10*: We showed in this section how to justify complete induction by a least counterexample argument. Show how complete induction can also be justified by an iteration.

Truth in Advertising

There are many difficulties, both theoretical and practical, in proving programs correct. An obvious question is "What does it mean for a program to be 'correct'?" As we mentioned in Chapter 1, most programs in practice are written to satisfy some informal specification. The specification itself may be incomplete or inconsistent. Even if there were a precise formal specification, we can show that no algorithm exists to prove that an arbitrary program is equivalent to a given specification.

However, in spite of these difficulties, it is beneficial to state and prove assertions about programs. The loop invariants of a program are often the most useful short explanation one can give of how the program works. Further, the programmer should have a loop invariant in mind while writing a piece of code. That is, there must be a reason why a program works, and this reason often has to do with an inductive hypothesis that holds each time the program goes around a loop or each time it performs a recursive call. The programmer should be able to envision a proof, even though it may be impractical to write out such a proof line by line.

2.5 Proving Properties of Programs

In this section we shall delve into an area where inductive proofs are essential: proving that a program does what it is claimed to do. We shall see a technique for explaining what an iterative program does as it goes around a loop. If we understand what the loops do, we generally understand what we need to know about an iterative program. In Section 2.9, we shall consider what is needed to prove properties of recursive programs.

Loop Invariants

Inductive assertion

The key to proving a property of a loop in a program is selecting a *loop invariant,* or *inductive assertion,* which is a statement S that is true each time we enter a particular point in the loop. The statement S is then proved by induction on a parameter that in some way measures the number of times we have gone around the loop. For example, the parameter could be the number of times we have reached the test of a while-loop, it could be the value of the loop index in a for-loop, or it could be some expression involving the program variables that is known to increase by 1 for each time around the loop.

✦ **Example 2.12.** As an example, let us consider the inner loop of `SelectionSort` from Section 2.2. These lines, with the original numbering from Fig. 2.2, are

```
(2)         small = i;
(3)         for (j = i+1; j < n; j++)
(4)             if (A[j] < A[small])
(5)                 small = j;
```

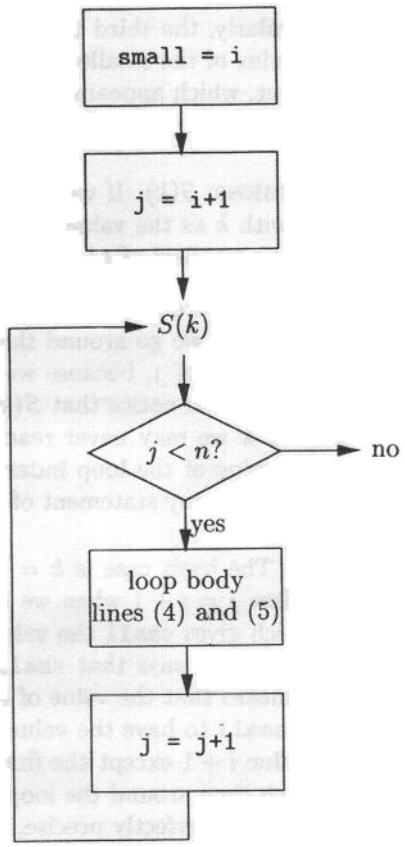


Fig. 2.10. Flowchart for the inner loop of `SelectionSort`.

Recall that the purpose of these lines is to make `small` equal to the index of an element of `A[i..n-1]` with the smallest value. To see why that claim is true, consider the flowchart for our loop shown in Fig. 2.10. This flowchart shows the five steps necessary to execute the program:

1. First, we need to initialize `small` to i, as we do in line (2).
2. At the beginning of the for-loop of line (3), we need to initialize `j` to $i + 1$.
3. Then, we need to test whether $j < n$.
4. If so, we execute the body of the loop, which consists of lines (4) and (5).
5. At the end of the body, we need to increment `j` and go back to the test.

In Fig. 2.10 we see a point just before the test that is labeled by a loop-invariant statement we have called $S(k)$; we shall discover momentarily what this statement must be. The first time we reach the test, `j` has the value $i + 1$ and `small` has the value i. The second time we reach the test, `j` has the value $i + 2$, because `j` has been incremented once. Because the body (lines 4 and 5) sets `small` to $i + 1$ if $A[i + 1]$ is less than $A[i]$, we see that `small` is the index of whichever of $A[i]$ and $A[i + 1]$ is smaller.[5]

[5] In case of a tie, *small* will be i. In general, we shall pretend that no ties occur and talk about "the smallest element" when we really mean "the first occurrence of the smallest element."

Similarly, the third time we reach the test, the value of `j` is $i + 3$ and `small` is the index of the smallest of `A[i..i+2]`. We shall thus try to prove the following statement, which appears to be the general rule.

STATEMENT $S(k)$: If we reach the test for $j < n$ in the for-statement of line (3) with k as the value of loop index `j`, then the value of `small` is the index of the smallest of `A[i..k-1]`.

Note that we are using the letter k to stand for one of the values that the variable `j` assumes, as we go around the loop. That is less cumbersome than trying to use j as the value of `j`, because we sometimes need to keep k fixed while the value of `j` changes. Also notice that $S(k)$ has the form "if we reach $\cdots$," because for some values of k we may never reach the loop test, as we broke out of the loop for a smaller value of the loop index `j`. If k is one of those values, then $S(k)$ is surely true, because any statement of the form "if A then B" is true when A is false.

BASIS. The basis case is $k = i + 1$, where i is the value of the variable `i` at line (3).[6] Now $j = i + 1$ when we begin the loop. That is, we have just executed line (2), which gives `small` the value i, and we have initialized `j` to $i + 1$ to begin the loop. $S(i + 1)$ says that `small` is the index of the smallest element in `A[i..i]`, which means that the value of `small` must be i. But we just observed that line (2) causes `small` to have the value i. Technically, we must also show that `j` can never have value $i + 1$ except the first time we reach the test. The reason, intuitively, is that each time around the loop, we increment `j`, so it will never again be as low as $i + 1$. (To be perfectly precise, we should give an inductive proof of the assumption that $j > i + 1$ except the first time through the test.) Thus, the basis, $S(i + 1)$, has been shown to be true.

INDUCTION. Now let us assume as our inductive hypothesis that $S(k)$ holds, for some $k \geq i + 1$, and prove $S(k + 1)$. First, if $k \geq n$, then we break out of the loop when `j` has the value k, or earlier, and so we are sure never to reach the loop test with the value of `j` equal to $k + 1$. In that case, $S(k + 1)$ is surely true.

Thus, let us assume that $k < n$, so that we actually make the test with `j` equal to $k+1$. $S(k)$ says that `small` indexes the smallest of `A[i..k-1]`, and $S(k+1)$ says that `small` indexes the smallest of `A[i..k]`. Consider what happens in the body of the loop (lines 4 and 5) when `j` has the value k; there are two cases, depending on whether the test of line (4) is true or not.

1. If $A[k]$ is not smaller than the smallest of `A[i..k-1]`, then the value of `small` does not change. In that case, however, `small` also indexes the smallest of `A[i..k]`, since $A[k]$ is not the smallest. Thus, the conclusion of $S(k + 1)$ is true in this case.

2. If $A[k]$ is smaller than the smallest of $A[i]$ through $A[k - 1]$, then `small` is set to k. Again, the conclusion of $S(k + 1)$ now holds, because k is the index of the smallest of `A[i..k]`.

---

[6] As far as the loop of lines (3) to (5) is concerned, `i` does not change. Thus, $i + 1$ is an appropriate constant to use as the basis value.

Thus, in either case, `small` is the index of the smallest of `A[i..k]`. We go around the for-loop by incrementing the variable `j`. Thus, just before the loop test, when `j` has the value $k+1$, the conclusion of $S(k+1)$ holds. We have now shown that $S(k)$ implies $S(k+1)$. We have completed the induction and conclude that $S(k)$ holds for all values $k \geq i+1$.

Next, we apply $S(k)$ to make our claim about the inner loop of lines (3) through (5). We exit the loop when the value of `j` reaches n. Since $S(n)$ says that `small` indexes the smallest of `A[i..n-1]`, we have an important conclusion about the working of the inner loop. We shall see how it is used in the next example. ✦

```
(1)     for (i = 0; i < n-1; i++) {
(2)         small = i;
(3)         for (j = i+1; j < n; j++)
(4)             if (A[j] < A[small])
(5)                 small = j;
(6)         temp = A[small];
(7)         A[small] = A[i];
(8)         A[i] = temp
        }
```

Fig. 2.11. The body of the `SelectionSort` function.

✦ **Example 2.13.** Now, let us consider the entire `SelectionSort` function, the heart of which we reproduce in Fig. 2.11. A flowchart for this code is shown in Fig. 2.12, where "body" refers to lines (2) through (8) of Fig. 2.11. Our inductive assertion, which we refer to as $T(m)$, is again a statement about what must be true just before the test for termination of the loop. Informally, when `i` has the value m, we have selected m of the smallest elements and sorted them at the beginning of the array. More precisely, we prove the following statement $T(m)$ by induction on m.

STATEMENT $T(m)$: If we reach the loop test $i < n-1$ of line (1) with the value of variable `i` equal to m, then

a) `A[0..m-1]` are in sorted order; that is, $A[0] \leq A[1] \leq \cdots \leq A[m-1]$.

b) All of `A[m..n-1]` are at least as great as any of `A[0..m-1]`.

BASIS. The basis case is $m = 0$. The basis is true for trivial reasons. If we look at the statement $T(0)$, part (a) says that `A[0..-1]` are sorted. But there are no elements in the range $A[0], \ldots, A[-1]$, and so (a) must be true. Similarly, part (b) of $T(0)$ says that all of `A[0..n-1]` are at least as large as any of `A[0..-1]`. Since there are no elements of the latter description, part (b) is also true.

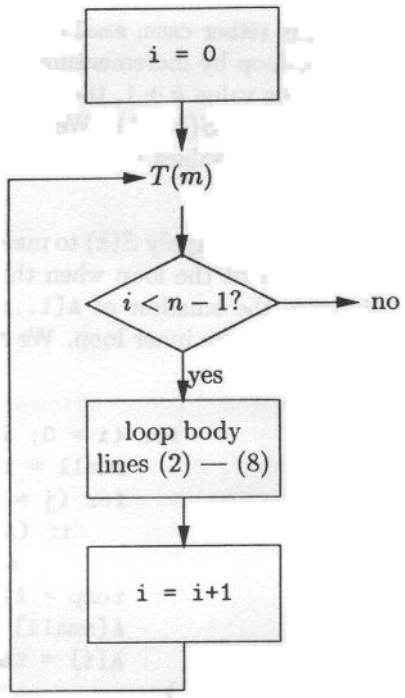


Fig. 2.12. Flow-chart for the entire selection sort function.

INDUCTION. For the inductive step, we assume that $T(m)$ is true for some $m \geq 0$, and we show that $T(m+1)$ holds. As in Example 2.12, we are trying to prove a statement of the form "if A then B," and such a statement is true whenever A is false. Thus, $T(m+1)$ is true if the assumption that we reach the for-loop test with `i` equal to $m+1$ is false. Thus, we may assume that we actually reach the test with `i` having the value $m+1$; that is, we may assume $m < n-1$.

When `i` has the value m, the body of the loop finds a smallest element in `A[m..n-1]` (as proved by the statement $S(m)$ of Example 2.12). This element is swapped with $A[m]$ in lines (6) through (8). Part (b) of the inductive hypothesis, $T(m)$, tells us the element chosen must be at least as large as any of `A[0..m-1]`. Moreover, those elements were sorted, so now all of `A[i..m]` are sorted. That proves part (a) of statement $T(m+1)$.

To prove part (b) of $T(m+1)$, we see that $A[m]$ was just selected to be as small as any of `A[m+1..n-1]`. Part (a) of $T(m)$ tells us that `A[0..m-1]` were already as small as any of `A[m+1..n-1]`. Thus, after executing the body of lines (2) through (8) and incrementing `i`, we know that all of `A[m+1..n-1]` are at least as large as any of `A[0..m]`. Since now the value of `i` is $m+1$, we have shown the truth of the statement $T(m+1)$ and thus have proved the inductive step.

Now, let $m = n-1$. We know that we exit the outer for-loop when `i` has the value $n-1$, so $T(n-1)$ will hold after we finish this loop. Part (a) of $T(n-1)$ says that all of `A[0..n-2]` are sorted, and part (b) says that $A[n-1]$ is as large as any of the other elements. Thus, after the program terminates the elements in `A` are in nonincreasing order; that is, they are sorted. ✦

Loop Invariants for While-Loops

When we have a while-loop of the form

```
while (<condition>)
    <body>
```

it usually makes sense to find the appropriate loop invariant for the point just before the test of the condition. Generally, we try to prove the loop invariant holds by induction on the number of times around the loop. Then, when the condition becomes false, we can use the loop invariant, together with the falsehood of the condition, to conclude something useful about what is true after the while-loop terminates.

However, unlike for-loops, there may not be a variable whose value counts the number of times around the while-loop. Worse, while the for-loop is guaranteed to iterate only up to the limit of the loop (for example, up to $n - 1$ for the inner loop of the `SelectionSort` program), there is no reason to believe that the condition of the while-loop will ever become false. Thus, part of the proof of correctness for a while-loop is a proof that it eventually terminates. We usually prove termination by identifying some expression E, involving the variables of the program, such that

While-loop termination

1. The value of E decreases by at least 1 each time around the loop, and
2. The loop condition is false if E is as low as some specified constant, such as 0.

✦ **Example 2.14.** The factorial function, written $n!$, is defined as the product of the integers $1 \times 2 \times \cdots \times n$. For example, $1! = 1$, $2! = 1 \times 2 = 2$, and

Factorial

$$5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$$

Figure 2.13 shows a simple program fragment to compute $n!$ for integers $n \geq 1$.

```
(1)     scanf("%d", &n);
(2)     i = 2;
(3)     fact = 1;
(4)     while (i <= n) {
(5)         fact = fact*i;
(6)         i++;
        }
(7)     printf("%d\n" fact);
```

Fig. 2.13. Factorial program fragment.

To begin, let us prove that the while-loop of lines (4) to (6) in Fig. 2.13 must terminate. We shall choose E to be the expression $n - i$. Notice that each time around the while-loop, `i` is increased by 1 at line (6) and `n` remains unchanged. Therefore, E decreases by 1 each time around the loop. Moreover, when E is -1 or less, we have $n - i \leq -1$, or $i \geq n + 1$. Thus, when E becomes negative, the loop condition $i \leq n$ will be false and the loop will terminate. We don't know how large E is initially, since we don't know what value of `n` will be read. Whatever that value is, however, E will eventually reach as low as -1, and the loop will terminate.

Now we must prove that the program of Fig. 2.13 does what it is intended to do. The appropriate loop-invariant statement, which we prove by induction on the value of the variable `i`, is

STATEMENT $S(j)$: If we reach the loop test $i \leq n$ with the variable `i` having the value j, then the value of the variable `fact` is $(j-1)!$.

BASIS. The basis is $S(2)$. We reach the test with `i` having value 2 only when we enter the loop from the outside. Prior to the loop, lines (2) and (3) of Fig. 2.13 set `fact` to 1 and `i` to 2. Since $1 = (2-1)!$, the basis is proved.

INDUCTION. Assume $S(j)$, and prove $S(j+1)$. If $j > n$, then we break out of the while-loop when `i` has the value j or earlier, and thus we never reach the loop test with `i` having the value $j+1$. In that case, $S(j+1)$ is trivially true, because it is of the form "If we reach $\cdots$."

Thus, assume $j \leq n$, and consider what happens when we execute the body of the while-loop with `i` having the value j. By the inductive hypothesis, before line (5) is executed, `fact` has value $(j-1)!$, and `i` has the value j. Thus, after line (5) is executed, `fact` has the value $j \times (j-1)!$, which is $j!$.

At line (6), `i` is incremented by 1 and so attains the value $j+1$. Thus, when we reach the loop test with `i` having value $j+1$, the value of `fact` is $j!$. The statement $S(j+1)$ says that when `i` equals $j+1$, `fact` equals $((j+1)-1)!$, or $j!$. Thus, we have proved statement $S(j+1)$, and completed the inductive step.

We already have shown that the while-loop will terminate. Evidently, it terminates when `i` first attains a value greater than n. Since `i` is an integer and is incremented by 1 each time around the loop, `i` must have the value $n+1$ when the loop terminates. Thus, when we reach line (7), statement $S(n+1)$ must hold. But that statement says that `fact` has the value $n!$. Thus, the program prints $n!$, as we wished to prove.

As a practical matter, we should point out that on any computer the factorial program in Fig. 2.13 will print $n!$ as an answer for very few values of n. The problem is that the factorial function grows so rapidly that the size of the answer quickly exceeds the maximum size of an integer on any real computer. ✦

EXERCISES

2.5.1: What is an appropriate loop invariant for the following program fragment, which sets `sum` equal to the sum of the integers from 1 to n?

```
scanf("%d",&n);
sum = 0;
for (i = 1; i <= n; i++)
    sum = sum + i;
```

Prove your loop invariant by induction on i, and use it to prove that the program works as intended.

2.5.2: The following fragment computes the sum of the integers in array `A[0..n-1]`:

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

What is an appropriate loop invariant? Use it to show that the fragment works as intended.

2.5.3*: Consider the following fragment:

```
scanf("%d", &n);
x = 2;
for (i = 1; i <= n; i++)
    x = x * x;
```

An appropriate loop invariant for the point just before the test for $i \leq n$ is that if we reach that point with the value k for variable i, then $x = 2^{2^{k-1}}$. Prove that this invariant holds, by induction on k. What is the value of x after the loop terminates?

```
sum = 0;
scanf("%d", &x);
while (x >= 0) {
    sum = sum + x;
    scanf("%d", &x);
}
```

Fig. 2.14. Summing a list of integers terminated by a negative integer.

2.5.4*: The fragment in Fig. 2.14 reads integers until it finds a negative integer, and then prints the accumulated sum. What is an appropriate loop invariant for the point just before the loop test? Use the invariant to show that the fragment performs as intended.

2.5.5: Find the largest value of n for which the program in Fig. 2.13 works on your computer. What are the implications of fixed-length integers for proving programs correct?

2.5.6: Show by induction on the number of times around the loop of Fig. 2.10 that $j > i + 1$ after the first time around.

2.6 Recursive Definitions

Inductive definition

In a *recursive*, or *inductive*, definition, we define one or more classes of closely related objects (or facts) in terms of the objects themselves. The definition must not be meaningless, like "a widget is a widget of some color," or paradoxical, like "something is a glotz if and only if it is not a glotz." Rather, a recursive definition involves

1. One or more *basis rules*, in which some simple objects are defined, and
2. One or more *inductive rules*, whereby larger objects are defined in terms of smaller ones in the collection.

✦ **Example 2.15.** In the previous section we defined the factorial function by an iterative algorithm: multiply $1 \times 2 \times \cdots \times n$ to get $n!$. However, we can also define the value of $n!$ recursively, as follows.

BASIS. $1! = 1$.

INDUCTION. $n! = n \times (n-1)!$.

For example, the basis tells us that $1! = 1$. We can use this fact in the inductive step with $n = 2$ to find

$$2! = 2 \times 1! = 2 \times 1 = 2$$

With $n = 3$, 4, and 5, we get

$$3! = 3 \times 2! = 3 \times 2 = 6$$
$$4! = 4 \times 3! = 4 \times 6 = 24$$
$$5! = 5 \times 4! = 5 \times 24 = 120$$

and so on. Notice that, although it appears that the term "factorial" is defined in terms of itself, in practice, we can get the value of $n!$ for progressively higher values of n in terms of the factorials for lower values of n only. Thus, we have a meaningful definition of "factorial."

Strictly speaking, we should prove that our recursive definition of $n!$ gives the same result as our original definition,

$$n! = 1 \times 2 \times \cdots \times n$$

To do so, we shall prove the following statement.

STATEMENT $S(n)$: $n!$, as defined recursively above, equals $1 \times 2 \times \cdots \times n$.

The proof will be by induction on n.

BASIS. $S(1)$ clearly holds. The basis of the recursive definition tells us that $1! = 1$, and the product $1 \times \cdots \times 1$ (i.e., the product of the integers "from 1 to 1") is evidently 1 as well.

INDUCTION. Assume that $S(n)$ holds; that is, $n!$, as given by the recursive definition, equals $1 \times 2 \times \cdots \times n$. Then the recursive definition tells us that

$$(n+1)! = (n+1) \times n!$$

If we use the commutative law for multiplication, we see that

$$(n+1)! = n! \times (n+1) \tag{2.11}$$

By the inductive hypothesis,

$$n! = 1 \times 2 \times \cdots \times n$$

Thus, we may substitute $1 \times 2 \times \cdots \times n$ for $n!$ in Equation (2.11) to get

$$(n+1)! = 1 \times 2 \times \cdots \times n \times (n+1)$$

which is the statement $S(n+1)$. We have thereby proved the inductive hypothesis and shown that our recursive definition of $n!$ is the same as our iterative definition.✦

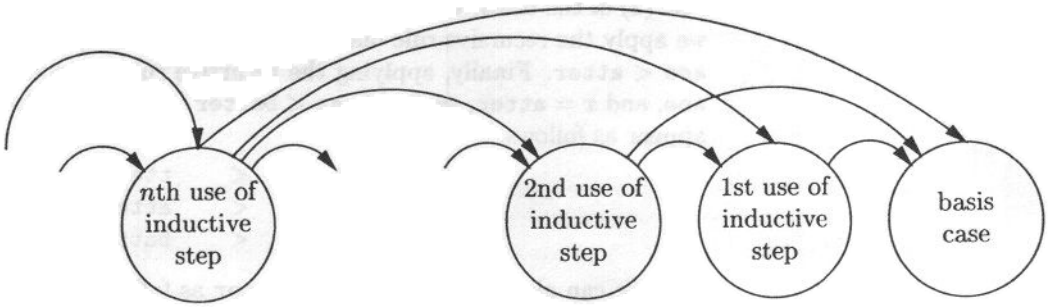


Fig. 2.15. In a recursive definition, we construct objects in rounds, where the objects constructed in one round may depend on objects constructed in all previous rounds.

Figure 2.15 suggests the general nature of a recursive definition. It is similar in structure to a complete induction, in that there is an infinite sequence of cases, each of which can depend on any or all of the previous cases. We start by applying the basis rule or rules. On the next round, we apply the inductive rule or rules to what we have already obtained, to construct new facts or objects. On the following round, we again apply the inductive rules to what we have, obtaining new facts or objects, and so on.

In Example 2.15, where we were defining the factorial, we discovered the value of 1! by the basis case, 2! by one application of the inductive step, 3! by two applications, and so on. Here, the induction had the form of an "ordinary" induction, where we used in each round only what we had discovered in the previous round.

✦ **Example 2.16.** In Section 2.2 we defined the notion of lexicographic order of strings, and our definition was iterative in nature. Roughly, we test whether string $c_1 \cdots c_n$ precedes string $d_1 \cdots d_m$ by comparing corresponding symbols c_i and d_i from the left, until we either find an i for which $c_i \neq d_i$ or come to the end of one of the strings. The following recursive definition defines those pairs of strings w and x such that w precedes x in lexicographic order. Intuitively, the induction is on the number of pairs of equal characters at the beginnings of the two strings involved.

Lexicographic order

BASIS. The basis covers those pairs of strings for which we can immediately resolve the question of which comes first in lexicographic order. There are two parts of the basis.

1. $\epsilon < w$ for any string w other than ϵ itself. Recall that ϵ is the empty string, or the string with no characters.
2. If $c < d$, where c and d are characters, then for any strings w and x, we have $cw < dx$.

INDUCTION. If $w < x$ for strings w and x, then for any character c we have $cw < cx$.

For instance, we can use the above definition to show that `base` < `batter`. By rule (2) of the basis, with $c =$ `s`, $d =$ `t`, $w =$ `e`, and $x =$ `ter`, we have `se` < `tter`. If we apply the recursive rule once, with $c =$ `a`, $w =$ `se`, and $x =$ `tter`, we infer that `ase` < `atter`. Finally, applying the recursive rule a second time with $c =$ `b`, $w =$ `ase`, and $x =$ `atter`, we find `base` < `batter`. That is, the basis and inductive steps appear as follows:

`se`	<	`tter`
`ase`	<	`atter`
`base`	<	`batter`

We can also show that `bat` < `batter` as follows. Part (1) of the basis tells us that ϵ < `ter`. If we apply the recursive rule three times — with c equal to `t`, `a`, and `b`, in turn — we make the following sequence of inferences:

ϵ	<	`ter`
`t`	<	`tter`
`at`	<	`atter`
`bat`	<	`batter`

Now we should prove, by induction on the number of characters that two strings have in common at their left ends, that one string precedes the other according to the definition in Section 2.2 if and only if it precedes according to the recursive definition just given. We leave these two inductive proofs as exercises. ✦

In Example 2.16, the groups of facts suggested by Fig. 2.15 are large. The basis case gives us all facts $w < x$ for which either $w = \epsilon$ or w and x begin with different letters. One use of the inductive step gives us all $w < x$ facts where w and x have exactly one initial letter in common, the second use covers those cases where w and x have exactly two initial letters in common, and so on.

Expressions

Arithmetic expressions of all kinds are naturally defined recursively. For the basis of the definition, we specify what the atomic operands can be. For example, in C, atomic operands are either variables or constants. Then, the induction tells us what operators may be applied, and to how many operands each is applied. For instance, in C, the operator < can be applied to two operands, the operator symbol − can be applied to one or two operands, and the function application operator, represented by a pair of parenthesis with as many commas inside as necessary, can be applied to one or more operands, as $f(a_1, \ldots, a_n)$.

✦ **Example 2.17.** It is common to refer to the following set of expressions as "arithmetic expressions."

BASIS. The following types of atomic operands are arithmetic expressions:

1. Variables
2. Integers
3. Real numbers

INDUCTION. If E_1 and E_2 are arithmetic expressions, then the following are also arithmetic expressions:

1. $(E_1 + E_2)$
2. $(E_1 - E_2)$
3. $(E_1 \times E_2)$
4. $(E_1 \ / \ E_2)$

Infix operator

The operators $+$, $-$, $\times$, and $/$ are said to be binary operators, because they take two arguments. They are also said to be *infix* operators, because they appear between their two arguments.

Additionally, we allow a minus sign to imply negation (change of sign), as well as subtraction. That possibility is reflected in the fifth and last recursive rule:

5. If E is an arithmetic expression, then so is $(-E)$.

Unary, prefix operator

An operator like $-$ in rule (5), which takes only one operand, is said to be a *unary* operator. It is also said to be a *prefix* operator, because it appears before its argument.

Figure 2.16 illustrates some arithmetic expressions and explains why each is an expression. Note that sometimes parentheses are not needed, and we can omit them. In the final expression (*vi*) of Fig. 2.16, the outer parentheses and the parentheses around $-(x + 10)$ can be omitted, and we could write $y \times -(x + 10)$. However, the remaining parentheses are essential, since $y \times -x + 10$ is conventionally interpreted as $(y \times -x) + 10$, which is not an equivalent expression (try $y = 1$ and $x = 0$, for instance).[7] ✦

i)	x	Basis rule (1)
ii)	10	Basis rule (2)
iii)	$(x + 10)$	Recursive rule (1) on (*i*) and (*ii*)
iv)	$\big(-(x + 10)\big)$	Recursive rule (5) on (*iii*)
v)	y	Basis rule (1)
vi)	$\Big(y \times \big(-(x + 10)\big)\Big)$	Recursive rule (3) on (*v*) and (*iv*)

Fig. 2.16. Some sample arithmetic expressions.

[7] Parentheses are redundant when they are implied by the conventional precedences of operators (unary minus highest, then multiplication and division, then addition and subtraction) and by the convention of "left associativity," which says that we group operators at the same precedence level (e.g., a string of plusses and minuses) from the left. These conventions should be familiar from C, as well as from ordinary arithmetic.

More Operator Terminology

Postfix operator

A unary operator that appears after its argument, as does the factorial operator ! in expressions like $n!$, is said to be a *postfix* operator. Operators that take more than one operand can also be prefix or postfix operators, if they appear before or after all their arguments, respectively. There are no examples in C or ordinary arithmetic of operators of these types, although in Section 5.4 we shall discuss notations in which all operators are prefix or postfix operators.

Ternary operator

An operator that takes three arguments is a *ternary* operator. In C, `?:` is a ternary operator, as in the expression `c?x:y` meaning "if c then x else y." In general, if an operator takes k arguments, it is said to be *k-ary*.

Balanced Parentheses

Strings of parentheses that can appear in expressions are called *balanced parentheses*. For example, the pattern `(((())))` appears in expression (*vi*) of Fig. 2.16, and the expression

$$\Big((a+b)\times\big((c+d)-e\big)\Big)$$

has the pattern `(()(()))`. The empty string, ϵ, is also a string of balanced parentheses; it is the pattern of the expression x, for example. In general, what makes a string of parentheses balanced is that it is possible to match each left parenthesis with a right parenthesis that appears somewhere to its right. Thus, a common definition of "balanced parenthesis strings" consists of two rules:

1. A balanced string has an equal number of left and right parentheses.

Profile

2. As we move from left to right along the string, the profile of the string never becomes negative, where the *profile* is the running total of the number of left parentheses seen minus the number of right parentheses seen.

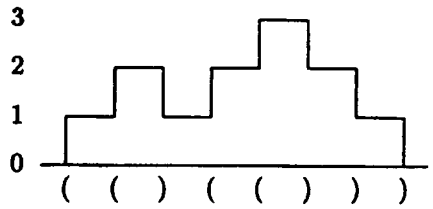
Note that the profile must begin and end at 0. For example, Fig. 2.17(a) shows the profile of `(()(()))`, and Fig. 2.17(b) shows the profile of `()(())()`.

There are a number of recursive definitions for the notion of "balanced parentheses." The following is a bit subtle, but we shall prove that it is equivalent to the preceding, nonrecursive definition involving profiles.

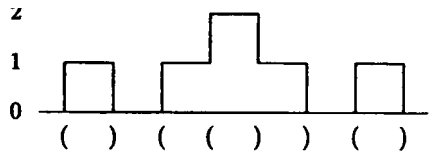
BASIS. The empty string is a string of balanced parentheses.

INDUCTION. If x and y are strings of balanced parentheses, then $(x)y$ is also a string of balanced parentheses.

✦ **Example 2.18.** By the basis, ϵ is a balanced-parenthesis string. If we apply the recursive rule, with x and y both equal to ϵ, then we infer that `()` is balanced. Notice that when we substitute the empty string for a variable, such as x or y, that variable "disappears." Then we may apply the recursive rule with:



(a) Profile of (()(())).



(b) Profile of ()(())().

Fig. 2.17. Profiles of two strings of parentheses.

1. $x =$ () and $y = \epsilon$, to discover that (()) is balanced.
2. $x = \epsilon$ and $y =$ (), to find that ()() is balanced.
3. $x = y =$ () to infer that (())() is balanced.

As a final example, since we now know that (()) and ()() are balanced, we may let these be x and y in the recursive rule, respectively, and show that ((()))()() is balanced. ✦

Profile-balanced

We can show that the two definitions of "balanced" specify the same sets of strings. To make things clearer, let us refer to strings that are balanced according to the recursive definition simply as *balanced* and refer to those balanced according to the nonrecursive definition as *profile-balanced*. That is, the profile-balanced strings are those whose profile ends at 0 and never goes negative. We need to show two things:

1. Every balanced string is profile-balanced.
2. Every profile-balanced string is balanced.

These are the aims of the inductive proofs in the next two examples.

✦ **Example 2.19.** First, let us prove part (1), that every balanced string is profile-balanced. The proof is a complete induction that mirrors the induction by which the class of balanced strings is defined. That is, we prove

STATEMENT $S(n)$: If string w is defined to be balanced by n applications of the recursive rule, then w is profile-balanced.

BASIS. The basis is $n = 0$. The only string that can be shown to be balanced without any application of the recursive rule is ϵ, which is balanced according to the basis rule. Evidently, the profile of the empty string ends at 0 and does not go negative, so ϵ is profile-balanced.

INDUCTION. Assume that $S(i)$ is true for $i = 0, 1, \ldots, n$, and consider an instance of $S(n+1)$, that is, a string w whose proof of balance requires $n+1$ uses of the recursive rule. Consider the last such use, in which we took two strings x and y, already known to be balanced, and formed w as $(x)y$. We used the recursive rule $n+1$ times to form w, and one use was the last step, which helped form neither x nor y. Thus, neither x nor y requires more than n uses of the recursive rule. Therefore, the inductive hypothesis applies to both x and y, and we can conclude that x and y are profile-balanced.

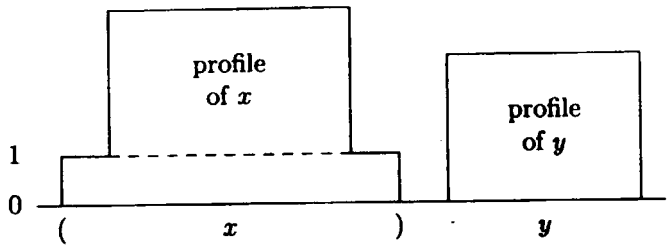


Fig. 2.18. Constructing the profile of $w = (x)y$.

The profile of w is as suggested in Fig. 2.18. It first goes up one level, in response to the first left parenthesis. Then comes the profile of x, raised one level, as indicated by the dashed line. We used the inductive hypothesis to conclude that x is profile-balanced; therefore, its profile begins and ends at level 0 and never goes negative. As the x portion of w's profile is raised one level in Fig. 2.18, that portion begins and ends at level 1 and never goes below level 1.

The explicitly shown right parenthesis between x and y lowers the profile of w to 0. Then comes the profile of y. By the inductive hypothesis, y is profile-balanced. Thus, the y portion of w's profile does not go below 0, and it ends the profile of w at 0.

We have now constructed the profile of w and see that it meets the condition for a profile-balanced string. That is, it begins and ends at 0, and it never becomes negative. Thus, we have proved that if a string is balanced, it is profile-balanced. ✦

Now we shall address the second direction of the equivalence between the two definitions of "balanced parentheses." We show in the next example that a profile-balanced string is balanced.

✦ **Example 2.20.** We prove part (2), that "profile-balanced" implies "balanced," by complete induction on the length of the string of parentheses. The formal statement is

Proofs About Recursive Definitions

Notice that Example 2.19 proves an assertion about a class of recursively defined objects (the balanced strings of parentheses) by induction on the number of times the recursive rule is used to establish that the object is in the defined class. That is a very common way to deal with recursively defined concepts; in fact, it is one of the reasons recursive definitions are useful. As another illustration, in Example 2.15, we showed a property of the recursively defined factorial values (that $n!$ is the product of the integers from 1 to n) by induction on n. But n is also 1 plus the number of times we used the recursive rule in the definition of $n!$, so the proof could also be considered an induction on the number of applications of the recursive rule.

STATEMENT $S(n)$: If a string w of length n is profile-balanced, then it is balanced.

BASIS. If $n = 0$, then the string must be ϵ. We know that ϵ is balanced by the basis rule of the recursive definition.

INDUCTION. Suppose that profile-balanced strings of length equal to or less than n are balanced. We must prove $S(n+1)$, that profile-balanced strings of length $n+1$ are also balanced.[8] Consider such a string w. Since w is profile-balanced, it cannot start with a right parenthesis, or its profile would immediately go negative. Thus, w begins with a left parenthesis.

Let us break w into two parts. The first part starts at the beginning of w and ends where the profile of w first becomes 0. The second part is the remainder of w. For example, the profile of Fig. 2.17(a) first becomes 0 at the end, so if $w =$ (()(())), then the first part is the entire string and the second part is ϵ. In Fig. 2.17(b), where $w =$ ()(())(), the first part is (), and the second part is (())().

The first part can never end in a left parenthesis, because then the profile would be negative at the position just before. Thus, the first part begins with a left parenthesis and ends with a right parenthesis. We can therefore write w as $(x)y$, where (x) is the first part and y is the second part. Both x and y are shorter than w, so if we can show they are profile-balanced, then we can use the inductive hypothesis to infer that they are balanced. Then we can use the recursive rule in the definition of "balanced" to show that $w = (x)y$ is balanced.

It is easy to see that y is profile-balanced. Figure 2.18 also illustrates the relationship between the profiles of w, x, and y here. That is, the profile of y is a tail of the profile of w, beginning and ending at height 0. Since w is profile-balanced, we can conclude that y is also. Showing that x is profile-balanced is almost the same. The profile of x is a part of the profile of w; it begins and ends at level 1 in the profile of w, but we can lower it by one level to get the profile of x. We know that the profile of w never reaches 0 during the extent of x, because we picked (x) to be the shortest prefix of w that ends with the profile of w at level 0. Hence, the profile of x within w never reaches level 0, and the profile of x itself never becomes negative.

We have now shown both x and y to be profile-balanced. Since they are each

[8] Note that all profile-balanced strings happen to be of even length, so if $n+1$ is odd, we are not saying anything. However, we do not need the evenness of n for the proof.

shorter than w, the inductive hypothesis applies to them, and they are each balanced. The recursive rule defining "balanced" says that if x and y are balanced, then so is $(x)y$. But $w = (x)y$, and so w is balanced. We have now completed the inductive step and shown statement $S(n)$ to be true for all $n \geq 0$. ✦

EXERCISES

2.6.1*: Prove that the definitions of lexicographic order given in Example 2.16 and in Section 2.2 are the same. *Hint*: The proof consists of two parts, and each is an inductive proof. For the first part, suppose that $w < x$ according to the definition in Example 2.16. Prove the following statement $S(i)$ by induction on i: "If it is necessary to apply the recursive rule i times to show that $w < x$, then w precedes x according to the definition of 'lexicographic order' in Section 2.2." The basis is $i = 0$. The second part of the exercise is to show that if w precedes x in lexicographic order according to the definition in Section 2.2, then $w < x$ according to the definition in Example 2.16. Now the induction is on the number of initial positions that w and x have in common.

2.6.2: Draw the profiles of the following strings of parentheses:

a) (()(()))
b) ()())(()
c) (((())()()))
d) (()(()(())))

Which are profile-balanced? For those that are profile-balanced, use the recursive definition in Section 2.6 to show that they are balanced.

2.6.3*: Show that every string of balanced parentheses (according to the recursive definition in Section 2.6) is the string of parentheses in some arithmetic expression (see Example 2.17 for a definition of arithmetic expressions). *Hint*: Use a proof by induction on the number of times the recursive rule of the definition of "balanced parentheses" is used to construct the given string of balanced parentheses.

2.6.4: Tell whether each of the following C operators is prefix, postfix, or infix, and whether they are unary, binary, or k-ary for some $k > 2$:

a) <
b) &
c) %

2.6.5: If you are familiar with the UNIX file system or a similar system, give a recursive definition of the possible directory/file structures.

2.6.6*: A certain set S of integers is defined recursively by the following rules.

BASIS. 0 is in S.

INDUCTION. If i is in S, then $i + 5$ and $i + 7$ are in S.

a) What is the largest integer *not* in S?

b) Let j be your answer to part (a). Prove that all integers $j+1$ and greater are in S. *Hint*: Note the similarity to Exercise 2.4.8 (although here we are dealing with only nonnegative integers).

2.6.7*: Define recursively the set of even-parity strings, by induction on the length of the string. *Hint*: It helps to define two concepts simultaneously, both the even-parity strings and the odd-parity strings.

2.6.8*: We can define sorted lists of integers as follows.

BASIS. A list consisting of a single integer is sorted.

INDUCTION. If L is a sorted list in which the last element is a, and if $b \geq a$, then L followed by b is a sorted list.

Prove that this recursive definition of "sorted list" is equivalent to our original, nonrecursive definition, which is that the list consist of integers

$$a_1 \leq a_2 \leq \cdots \leq a_n$$

Remember, you need to prove two parts: (a) If a list is sorted by the recursive definition, then it is sorted by the nonrecursive definition, and (b) if a list is sorted by the nonrecursive definition, then it is sorted by the recursive definition. Part (a) can use induction on the number of times the recursive rule is used, and (b) can use induction on the length of the list.

2.6.9**: As suggested by Fig. 2.15, whenever we have a recursive definition, we can classify the objects defined according to the "round" on which each is generated, that is, the number of times the inductive step is applied before we obtain each object. In Examples 2.15 and 2.16, it was fairly easy to describe the results generated on each round. Sometimes it is more challenging to do so. How do you characterize the objects generated on the nth round for each of the following?

a) Arithmetic expressions like those described in Example 2.17. *Hint*: If you are familiar with trees, which are the subject of Chapter 5, you might consider the tree representation of expressions.

b) Balanced parenthesis strings. Note that the "number of applications used," as discussed in Example 2.19, is not the same as the round on which a string is discovered. For example, `(())()` uses the inductive rule three times but is discovered on round 2.

2.7 Recursive Functions

Direct and indirect recursion

A recursive function is one that is called from within its own body. Often, the call is *direct;* for example, a function F has a call to F within itself. Sometimes, however, the call is *indirect*: some function F_1 calls a function F_2 directly, which calls F_3 directly, and so on, until some function F_k in the sequence calls F_1.

There is a common belief that it is easier to learn to program iteratively, or to use nonrecursive function calls, than it is to learn to program recursively. While we cannot argue conclusively against that point of view, we do believe that recursive programming is easy once one has had the opportunity to practice the style.

More Truth in Advertising

A potential disadvantage of using recursion is that function calls on some machines are time-consuming, so that a recursive program may take more time to run than an iterative program for the same problem. However, on many modern machines function calls are quite efficient, and so this argument against using recursive programs is becoming less important.

Profiling

Even on machines with slow function-calling mechanisms, one can *profile* a program to find how much time is spent on each part of the program. One can then rewrite the parts of the program in which the bulk of its time is spent, replacing recursion by iteration if necessary. That way, one gets the advantages of recursion throughout most of the program, except for a small fraction of the code where speed is most critical.

Recursive programs are often more succinct or easier to understand than their iterative counterparts. More importantly, some problems are more easily attacked by recursive programs than by iterative programs.[9]

Often, we can develop a recursive algorithm by mimicking a recursive definition in the specification of a program we are trying to implement. A recursive function that implements a recursive definition will have a basis part and an inductive part. Frequently, the basis part checks for a simple kind of input that can be solved by the basis of the definition, with no recursive call needed. The inductive part of the function requires one or more recursive calls to itself and implements the inductive part of the definition. Some examples should clarify these points.

✦ **Example 2.21.** Figure 2.19 gives a recursive function that computes $n!$ given a positive integer n. This function is a direct transcription of the recursive definition of $n!$ in Example 2.15. That is, line (1) of Fig. 2.19 distinguishes the basis case from the inductive case. We assume that $n \geq 1$, so the test of line (1) is really asking whether $n = 1$. If so, we apply the basis rule, $1! = 1$, at line (2). If $n > 1$, then we apply the inductive rule, $n! = n \times (n-1)!$, at line (3).

```
            int fact(int n)
            {
(1)             if (n <= 1)
(2)                 return 1; /* basis */
                else
(3)                 return n*fact(n-1); /* induction */
            }
```

Fig. 2.19. Recursive function to compute $n!$ for $n \geq 1$.

For instance, if we call `fact(4)`, the result is a call to `fact(3)`, which calls

[9] Such problems often involve some kind of search. For instance, in Chapter 5 we shall see some recursive algorithms for searching trees, algorithms that have no convenient iterative analog (although there are equivalent iterative algorithms using stacks).

Defensive Programming

The program of Fig. 2.19 illustrates an important point about writing recursive programs so that they do not run off into infinite sequences of calls. We tacitly assumed that `fact` would never be called with an argument less than 1. Best, of course, is to begin `fact` with a test that $n \geq 1$, printing an error message and returning some particular value such as 0 if it is not. However, even if we believe very strongly that `fact` will never be called with $n < 1$, we shall be wise to include in the basis case all these "error cases." Then, the function `fact` called with erroneous input will simply return the value 1, which is wrong, but not a disaster (in fact, 1 is even correct for $n = 0$, since 0! is conventionally defined to be 1).

However, suppose we were to ignore the error cases and write line (1) of Fig. 2.19 as

```
if (n == 1)
```

Then if we called `fact(0)`, it would look like an instance of the inductive case, and we would next call `fact(-1)`, then `fact(-2)`, and so on, terminating with failure when the computer ran out of space to record the recursive calls.

`fact(2)`, which calls `fact(1)`. At that point, `fact(1)` applies the basis rule, because $n \leq 1$, and returns the value 1 to `fact(2)`. That call to `fact` completes line (3), returning 2 to `fact(3)`. In turn, `fact(3)` returns 6 to `fact(4)`, which completes line (3) by returning 24 as the answer. Figure 2.20 suggests the pattern of calls and returns. ✦

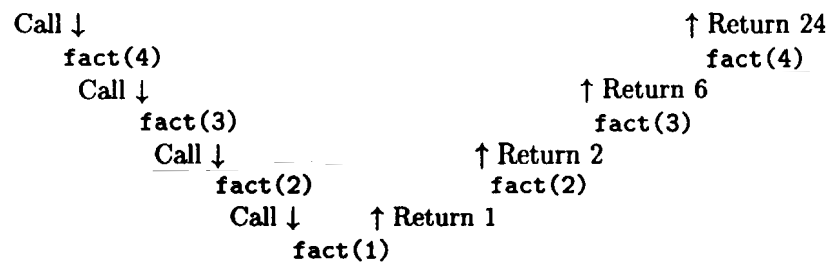


Fig. 2.20. Calls and returns resulting from call to `fact(4)`.

Size of arguments

We can picture a recursion much as we have pictured inductive proofs and definitions. In Fig. 2.21 we have assumed that there is a notion of the "size" of arguments for a recursive function. For example, for the function *fact* in Example 2.21 the value of the argument n itself is the appropriate size. We shall say more about the matter of size in Section 2.9. However, let us note here that it is essential for a recursion to make only calls involving arguments of smaller size. Also, we must reach the basis case — that is, we must terminate the recursion — when we reach some particular size, which in Fig. 2.21 is size 0.

In the case of the function `fact`, the calls are not as general as suggested by Fig. 2.21. A call to `fact(n)` results in a direct call to `fact(n-1)`, but `fact(n)` does

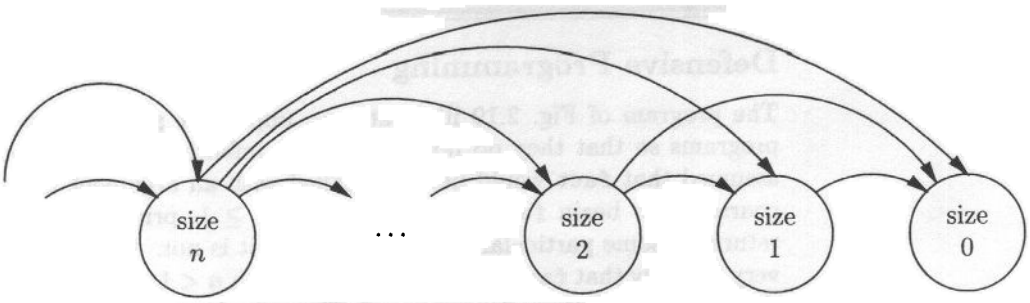


Fig. 2.21. A recursive function calls itself with arguments of smaller size.

not call `fact` with any smaller argument directly.

✦ **Example 2.22.** We can turn the function `SelectionSort` of Fig. 2.2 into a recursive function `recSS`, if we express the underlying algorithm as follows. Assume the data to be sorted is in `A[0..n-1]`.

1. Pick a smallest element from the tail of the array `A`, that is, from `A[i..n-1]`.
2. Swap the element selected in step (1) with `A[i]`.
3. Sort the remainder of the array, `A[i+1..n-1]`.

We can express selection sort as the following recursive algorithm.

BASIS. If $i = n - 1$, then only the last element of the array remains to be sorted. Since any one element is already sorted, we need not do anything.

INDUCTION. If $i < n - 1$, then find the smallest element in `A[i..n-1]`, swap it with `A[i]`, and recursively sort `A[i+1..n-1]`.

The entire algorithm is to perform the above recursion starting with $i = 0$.

Backward induction

If we see i as the parameter in the preceding induction, it is a case of *backward induction*, where we start with a high basis and by the inductive rule solve instances with smaller values of the parameter in terms of instances with higher values. That is a perfectly good style of induction, although we have not previously mentioned its possibility. However, we can also see this induction as an ordinary, or "forward" induction on a parameter $k = n - i$ that represents the number of elements in the tail of the array waiting to be sorted.

In Fig. 2.22, we see the program for `recSS(A,i,n)`. The second parameter `i` is the index of the first element in the unsorted tail of the array `A`. The third parameter `n` is the total number of elements in the array `A` to be sorted. Presumably, n is less than or equal to the maximum size of `A`. Thus, a call to `recSS(A,0,n)` will sort the entire array `A[0..n-1]`.

In terms of Fig. 2.21, $s = n - i$ is the appropriate notion of "size" for arguments of the function `recSS`. The basis is $s = 1$ — that is, sorting one element, in which case no recursive calls occur. The inductive step tells us how to sort s elements by picking the smallest and then sorting the remaining $s - 1$ elements.

At line (1) we test for the basis case, in which there is only one element remaining to be sorted (again, we are being defensive, so that if we somehow make a

```
    void recSS(int A[], int i, int n)
    {
        int j, small, temp;
(1)     if (i < n-1) {/* basis is when i = n-1, in which case */
                /* the function returns without changing A */
                /* induction follows */
(2)             small = i;
(3)             for (j = i+1; j < n; j++)
(4)                 if (A[j] < A[small])
(5)                     small = j;
(6)             temp = A[small];
(7)             A[small] = A[i];
(8)             A[i] = temp;
(9)             recSS(A, i+1, n);
            }
    }
```

Fig. 2.22. Recursive selection sort.

call with $i \geq n$, we shall not go into an infinite sequence of calls). In the basis case, we have nothing to do, so we just return.

The remainder of the function is the inductive case. Lines (2) through (8) are copied directly from the iterative version of selection sort. Like that program, these lines set `small` to the index of the array `A[i..n-1]` that holds a smallest element and then swap this element with `A[i]`. Finally, line (9) is the recursive call, which sorts the remainder of the array. ✦

EXERCISES

2.7.1: We can define n^2 recursively as follows.

BASIS. For $n = 1$, $1^2 = 1$.

INDUCTION. If $n^2 = m$, then $(n+1)^2 = m + 2n + 1$.

a) Write a recursive C function to implement this recursion.

b) Prove by induction on n that this definition correctly computes n^2.

2.7.2: Suppose that we are given array `A[0..4]`, with elements 10, 13, 4, 7, 11, in that order. What are the contents of the array `A` just before each recursive call to `recSS`, according to the recursive function of Fig. 2.22?

2.7.3: Suppose we define cells for a linked list of integers, as discussed in Section 1.3, using the macro `DefCell(int, CELL, LIST)` of Section 1.6. Recall, this macro expands to be the following type definition:

Divide-and-Conquer

One way of attacking a problem is to try to break it into subproblems and then solve the subproblems and combine their solutions into a solution for the problem as a whole. The term *divide-and-conquer* is used to describe this problem-solving technique. If the subproblems are similar to the original, then we may be able to use the same function to solve the subproblems recursively.

There are two requirements for this technique to work. The first is that the subproblems must be simpler than the original problem. The second is that after a finite number of subdivisions, we must encounter a subproblem that can be solved outright. If these criteria are not met, a recursive algorithm will continue subdividing the problem forever, without finding a solution.

Note that the recursive function `recSS` in Fig. 2.22 satisfies both criteria. Each time it is invoked, it is on a subarray that has one fewer element, and when it is invoked on a subarray containing a single element, it returns without invoking itself again. Similarly, the factorial program of Fig. 2.19 involves calls with a smaller integer value at each call, and the recursion stops when the argument of the call reaches 1. Section 2.8 discusses a more powerful use of the divide-and-conquer technique, called "merge sort." There, the size of the arrays being sorted diminishes very rapidly, because merge sort works by dividing the size in half, rather than subtracting 1, at each recursive call.

```
typedef struct CELL *LIST;
struct CELL {
    int element;
    LIST next;
};
```

Write a recursive function `find` that takes an argument of type LIST and returns TRUE if some cell of the list contains the integer 1698 as its element and returns FALSE if not.

2.7.4: Write a recursive function `add` that takes an argument of type LIST, as defined in Exercise 2.7.3, and returns the sum of the elements on the list.

2.7.5: Write a version of recursive selection sort that takes as argument a list of integers, using the cells mentioned in Exercise 2.7.3.

2.7.6: In Exercise 2.2.8 we suggested that one could generalize selection sort to use arbitrary *key* and *lt* functions to compare elements. Rewrite the recursive selection sort algorithm to incorporate this generality.

2.7.7*: Give a recursive algorithm that takes an integer i and produces the binary representation of i as a sequence of 0's and 1's, low-order bit first.

GCD

2.7.8*: The *greatest common divisor* (GCD) of two integers i and j is the largest integer that divides both i and j evenly. For example, $gcd(24, 30) = 6$, and $gcd(24, 35) = 1$. Write a recursive function that takes two integers i and j, with $i > j$, and returns $gcd(i, j)$. *Hint*: You may use the following recursive definition of *gcd*. It assumes that $i > j$.

BASIS. If j divides i evenly, then j is the GCD of i and j.

INDUCTION. If j does not divide i evenly, let k be the remainder when i is divided by j. Then $gcd(i, j)$ is the same as $gcd(j, k)$.

2.7.9**: Prove that the recursive definition of GCD given in Exercise 2.7.8 gives the same result as the nonrecursive definition (largest integer dividing both i and j evenly).

2.7.10: Often, a recursive definition can be turned into an algorithm fairly directly. For example, consider the recursive definition of "less than" on strings given in Example 2.16. Write a recursive function that tests whether the first of two given strings is "less than" the other. Assume that strings are represented by linked lists of characters.

2.7.11*: From the recursive definition of a sorted list given in Exercise 2.6.8, create a recursive sorting algorithm. How does this algorithm compare with the recursive selection sort of Example 2.22?

2.8 Merge Sort: A Recursive Sorting Algorithm

Divide and conquer

We shall now consider a sorting algorithm, called *merge sort*, which is radically different from selection sort. Merge sort is best described recursively, and it illustrates a powerful use of the *divide-and-conquer* technique, in which we sort a list $(a_1, a_2, \ldots, a_n)$ by "dividing" the problem into two similar problems of half the size. In principle, we could begin by dividing the list into two arbitrarily chosen equal-sized lists, but in the program we develop, we shall make one list out of the odd-numbered elements, $(a_1, a_3, a_5, \ldots)$ and the other out of the even-numbered elements, $(a_2, a_4, a_6, \ldots)$.[10] We then sort each of the half-sized lists separately. To complete the sorting of the original list of n elements, we merge the two sorted, half-sized lists by an algorithm to be described in the next example.

In the next chapter, we shall see that the time required for merge sort grows much more slowly, as a function of the length n of the list to be sorted, than does the time required by selection sort. Thus, even if recursive calls take some extra time, merge sort is greatly preferable to selection sort when n is large. In Chapter 3 we shall examine the relative performance of these two sorting algorithms.

Merging

To "merge" means to produce from two sorted lists a single sorted list containing all the elements of the two given lists and no other elements. For example, given the lists $(1, 2, 7, 7, 9)$ and $(2, 4, 7, 8)$, the merger of these lists is $(1, 2, 2, 4, 7, 7, 7, 8, 9)$. Note that it does not make sense to talk about "merging" lists that are not already sorted.

One simple way to merge two lists is to examine them from the front. At each step, we find the smaller of the two elements at the current fronts of the lists, choose that element as the next element on the combined list, and remove the chosen element from its list, exposing a new "first" element on that list. Ties can be broken

[10] Remember that "odd-numbered" and "even-numbered" refer to the positions of the elements on the list, and not to the values of these elements.

L_1	L_2	M
1,2,7,7,9	2,4,7,8	empty
2,7,7,9	2,4,7,8	1
7,7,9	2,4,7,8	1,2
7,7,9	4,7,8	1,2,2
7,7,9	7,8	1,2,2,4
7,9	7,8	1,2,2,4,7
9	7,8	1,2,2,4,7,7
9	8	1,2,2,4,7,7,7
9	empty	1,2,2,4,7,7,7,8
empty	empty	1,2,2,4,7,7,7,8,9

Fig. 2.23. Example of merging.

arbitrarily, although we shall take from the first list when the leading elements of both lists are the same.

✦ **Example 2.23.** Consider merging the two lists

$$L_1 = (1,2,7,7,9) \text{ and } L_2 = (2,4,7,8)$$

The first elements of the lists are 1 and 2, respectively. Since 1 is smaller, we choose it as the first element of the merged list M and remove 1 from L_1. The new L_1 is thus $(2,7,7,9)$. Now, both L_1 and L_2 have 2 as their first elements. We can pick either. Suppose we adopt the policy that we always pick the element from L_1 in case of a tie. Then merged list M becomes $(1,2)$, list L_1 becomes $(7,7,9)$, and L_2 remains $(2,4,7,8)$. The table in Fig. 2.23 shows the merging steps until lists L_1 and L_2 are both exhausted. ✦

We shall find it easier to design a recursive merging algorithm if we represent lists in the linked form suggested in Section 1.3. Linked lists will be reviewed in more detail in Chapter 6. In what follows, we shall assume that list elements are integers. Thus, each element can be represented by a "cell," or structure of the type `struct CELL`, and the list by a type `LIST`, which is a pointer to a `CELL`. These definitions are provided by the macro `DefCell(int, CELL, LIST)`, which we discussed in Section 1.6. This use of macro `DefCell` expands into:

```
typedef struct CELL *LIST;
struct CELL {
    int element;
    LIST next;
};
```

The `element` field of each cell contains an integer, and the `next` field contains a pointer to the next cell on the list. If the element at hand is the last on the list, then the `next` field contains the value `NULL`, which represents a null pointer. A list of integers is then represented by a pointer to the first cell on the list, that is, by a variable of type `LIST`. An empty list is represented by a variable with the value

```
    LIST merge(LIST list1, LIST list2)
    {
(1)     if (list1 == NULL) return list2;
(2)     else if (list2 == NULL) return list1;
(3)     else if (list1->element <= list2->element) {
            /* Here, neither list is empty, and the first list
               has the smaller first element. The answer is the
               first element of the first list followed by the
               merge of the remaining elements. */
(4)         list1->next = merge(list1->next, list2);
(5)         return list1;
        }
        else { /* list2 has smaller first element */
(6)         list2->next = merge(list1, list2->next);
(7)         return list2;
        }
    }
```

Fig. 2.24. Recursive merge.

NULL, in place of a pointer to the first element.

Figure 2.24 is a C implementation of a recursive merging algorithm. The function merge takes two lists as arguments and returns the merged list. That is, the formal parameters list1 and list2 are pointers to the two given lists, and the return value is a pointer to the merged list. The recursive algorithm can be described as follows.

BASIS. If either list is empty, then the other list is the desired result. This rule is implemented by lines (1) and (2) of Fig. 2.24. Note that if both lists are empty, then list2 will be returned. But that is correct, since the value of list2 is then NULL and the merger of two empty lists is an empty list.

INDUCTION. If neither list is empty, then each has a first element. We can refer to the two first elements as list1->element and list2->element, that is, the element fields of the cells pointed to by list1 and list2, respectively. Fig 2.25 is a picture of the data structure. The list to be returned begins with the cell of the smallest element. The remainder of the list is formed by merging all but that element.

For example, lines (4) and (5) handle the case in which the first element of list 1 is smallest. Line (4) is a recursive call to merge. The first argument of this call is list1->next, that is, a pointer to the second element on the first list (or NULL if the first list only has one element). Thus, the recursive call is passed the list consisting of all but the first element of the first list. The second argument is the entire second list. As a consequence, the recursive call to merge at line (4) will return a pointer to the merged list of all the remaining elements and store a pointer to this merged list in the next field of the first cell on list 1. At line (5), we return a pointer to that cell, which is now the first cell on the merged list of all the elements.

Figure 2.25 illustrates the changes. Dotted arrows are present when merge is

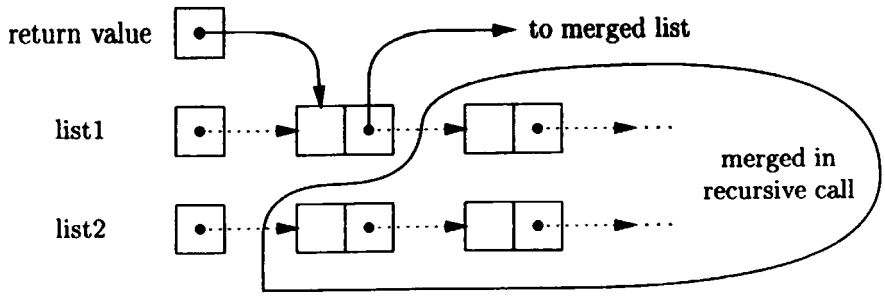


Fig. 2.25. Inductive step of merging algorithm.

called. Solid arrows are created by `merge`. Specifically, the return value of `merge` is a pointer to the cell of the smallest element, and the `next` field of that element is shown pointing to the list returned by the recursive call to `merge` at line (4).

Finally, lines (6) and (7) handle the case where the second list has the smallest element. The behavior of the algorithm is exactly as in lines (4) and (5), but the roles of the two lists are reversed.

✦ **Example 2.24.** Suppose we call `merge` on the lists $(1,2,7,7,9)$ and $(2,4,7,8)$ of Example 2.23. Figure 2.26 illustrates the sequence of calls made to `merge`, if we read the first column downward. We omit the commas separating list elements, but commas are used to separate the arguments of `merge`.

CALL	RETURN
`merge(12779, 2478)`	122477789
`merge(2779, 2478)`	22477789
`merge(779, 2478)`	2477789
`merge(779, 478)`	477789
`merge(779, 78)`	77789
`merge(79, 78)`	7789
`merge(9, 78)`	789
`merge(9, 8)`	89
`merge(9, NULL)`	9

Fig. 2.26. Recursive calls to `merge`.

For instance, since the first element of list 1 is less than the first element of list 2, line (4) of Fig. 2.24 is executed and we recursively merge all but the first element of list 1. That is, the first argument is the tail of list 1, or $(2,7,7,9)$, and the second argument is the full list 2, or $(2,4,7,8)$. Now the leading elements of both lists are the same. Since the test of line (3) in Fig. 2.24 favors the first list, we remove the 2 from list 1, and our next call to `merge` has first argument $(7,7,9)$ and second argument $(2,4,7,8)$.

The returned lists are indicated in the second column, read upward. Notice that, unlike the iterative description of merging suggested by Fig. 2.23, the recursive

algorithm assembles the merged list from the rear, whereas the iterative algorithm assembles it from the front. ✦

Splitting Lists

Another important task required for merge sort is splitting a list into two equal parts, or into parts whose lengths differ by 1 if the original list is of odd length. One way to do this job is to count the number of elements on the list, divide by 2, and break the list at the midpoint. Instead, we shall give a simple recursive function `split` that "deals" the elements into two lists, one consisting of the first, third, and fifth elements, and so on, and the other consisting of the elements at the even positions. More precisely, the function `split` removes the even-numbered elements from the list it is given as an argument and returns a new list consisting of the even-numbered elements.

The C code for function `split` is shown in Fig. 2.27. Its argument is a list of the type `LIST` that was defined in connection with the `merge` function. Note that the local variable `pSecondCell` is defined to be of type `LIST`. We really use `pSecondCell` as a pointer to the second cell on a list, rather than as a list itself; but of course type `LIST` is, in fact, a pointer to a cell.

It is important to observe that `split` is a function with a side effect. It removes the cells in the even positions from the list it is given as an argument, and it assembles these cells into a new list, which becomes the return value of the function.

```
     LIST split(LIST list)
     {
         LIST pSecondCell;

(1)      if (list == NULL) return NULL;
(2)      else if (list->next == NULL) return NULL;
         else { /* there are at least two cells */
(3)          pSecondCell = list->next;
(4)          list->next = pSecondCell->next;
(5)          pSecondCell->next = split(pSecondCell->next);
(6)          return pSecondCell;
         }
     }
```

Fig. 2.27. Splitting a list into two equal pieces.

The splitting algorithm can be described inductively, as follows. It uses an induction on the length of a list; that induction has a multiple basis case.

BASIS. If the list is of length 0 or 1, then we do nothing. That is, an empty list is "split" into two empty lists, and a list of a single element is split by leaving the element on the given list and returning an empty list of the even-numbered elements, of which there are none. The basis is handled by lines (1) and (2) of Fig. 2.27. Line (1) handles the case where `list` is empty, and line (2) handles the case where it is a single element. Notice that we are careful not to examine `list->next` in line (2) unless we have previously determined, at line (1), that `list` is not `NULL`.

INDUCTION. The inductive step applies when there are at least two elements on `list`. At line (3), we keep a pointer to the second cell of the list in the local variable `pSecondCell`. Line (4) makes the `next` field of the first cell skip over the second cell and point to the third cell (or become `NULL` if there are only two cells on the list). At line (5), we call `split` recursively, on the list consisting of all but the first two elements. The value returned by that call is a pointer to the fourth element (or `NULL` if the list is shorter than four elements), and we place this pointer in the `next` field of the second cell, to complete the linking of the even-numbered elements. A pointer to the second cell is returned by `split` at line (6); that pointer gives us access to the linked list of all the even-numbered elements of the original list.

The changes made by `split` are suggested in Fig. 2.28. Original pointers are dotted, and new pointers are solid. We also indicate the number of the line that creates each of the new pointers.

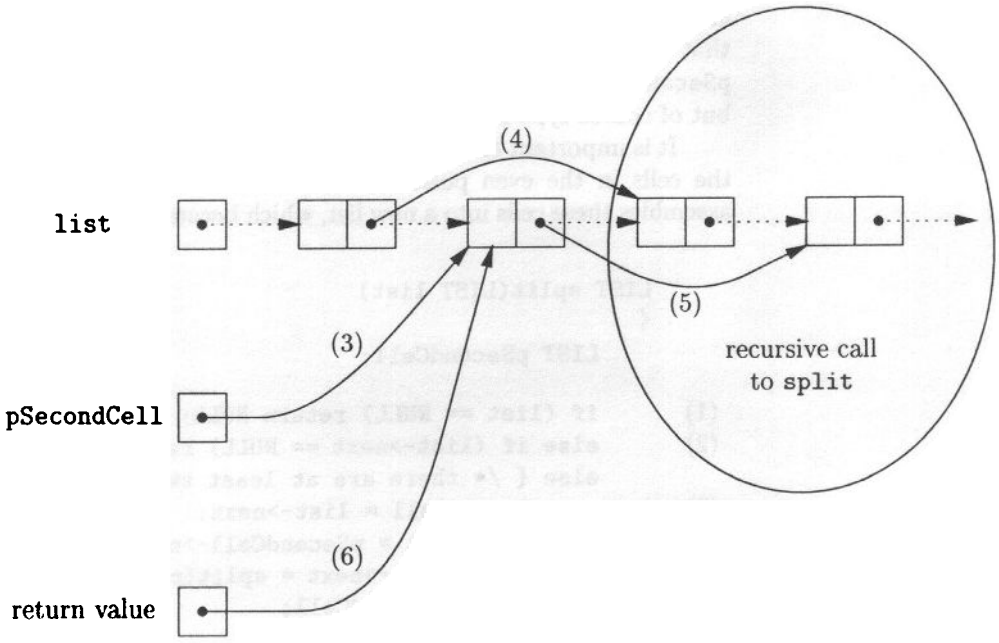


Fig. 2.28. Action of function `split`.

The Sorting Algorithm

The recursive sorting algorithm is shown in Fig. 2.29. The algorithm can be described by the following basis and inductive step.

BASIS. If the list to be sorted is empty or of length 1, just return the list; it is already sorted. The basis is taken care of by lines (1) and (2) of Fig. 2.29.

INDUCTION. If the list is of length at least 2, use the function `split` at line (3) to remove the even-numbered elements from `list` and use them to form another list, pointed to by local variable `SecondList`. Line (4) recursively sorts the half-sized lists, and returns the merger of the two lists.

```
        LIST MergeSort(LIST list)
        {
            LIST SecondList;

(1)         if (list == NULL) return NULL;
(2)         else if (list->next == NULL) return list;
            else {
                /* at least two elements on list */
(3)             SecondList = split(list);
                /* Note that as a side effect, half
                   the elements are removed from list */
(4)             return merge(MergeSort(list), MergeSort(SecondList));
            }
        }
```

Fig. 2.29. The merge sort algorithm.

✦ **Example 2.25.** Let us use merge sort on the list of single-digit numbers

742897721

We again omit commas between digits for succinctness. First, the list is split into two, by the call to `split` at line (3) of `MergeSort`. One of the resulting lists consists of the odd positions, and the other the evens; that is, `list` = 72971 and `SecondList` = 4872. At line (4), these lists are sorted, resulting in lists 12779 and 2478, and then merged to produce the sorted list 122477789.

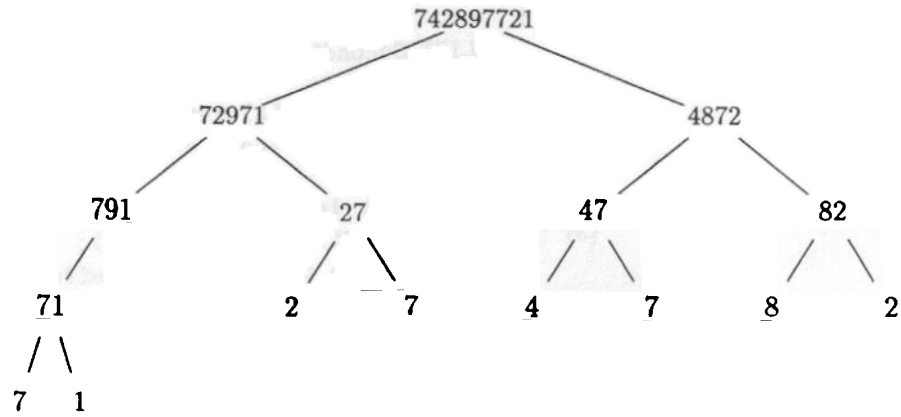
However, the sorting of the two half-sized lists does not occur by magic, but rather by the methodical application of the recursive algorithm. Initially, `MergeSort` splits the list on which it is called, if the list has length greater than 1. Figure 2.30(a) shows the recursive splitting of the lists until each list is of length 1. Then the split lists are merged, in pairs, going up the tree, until the entire list is sorted. This process is suggested in Fig. 2.30(b). However, it is worth noting that the splits and merges occur in a mixed order; not all splits are followed by all merges. For example, the first half list, 72971, is completely split and merged before we begin on the second half list, 4872. ✦

The Complete Program

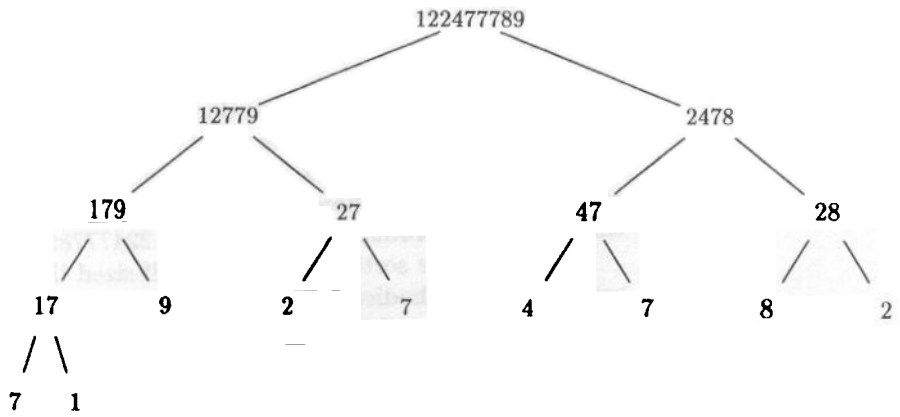
Figure 2.31 contains the complete merge sort program. It is analogous to the program in Fig. 2.3 that was based on selection sort. The function `MakeList` on line (1) reads each integer from the input and puts it into a linked list by a simple recursive algorithm, which we shall describe in detail in the next section. Line (2) of the main program contains the call to `MergeSort`, which returns a sorted list to `PrintList`. The function `PrintList` marches down the sorted list, printing each element.

EXERCISES

2.8.1: Show the result of applying the function `merge` to the lists $(1, 2, 3, 4, 5)$ and $(2, 4, 6, 8, 10)$.



(a) Splitting.



(b) Merging.

Fig. 2.30. Recursive splitting and merging.

2.8.2: Suppose we start with the list $(8, 7, 6, 5, 4, 3, 2, 1)$. Show the sequence of calls to `merge`, `split`, and `MergeSort` that result.

Multiway merge sort

2.8.3*: A *multiway* merge sort divides a list into k pieces of equal (or approximately equal) size, sorts them recursively, and then merges all k lists by comparing all their respective first elements and taking the smallest. The merge sort described in this section is for the case $k = 2$. Modify the program in Fig. 2.31 so that it becomes a multiway merge sort for the case $k = 3$.

2.8.4*: Rewrite the merge sort program to use the functions *lt* and *key*, described in Exercise 2.2.8, to compare elements of arbitrary type.

2.8.5: Relate each of the functions (a) `merge` (b) `split` (c) `MakeList` to Fig. 2.21. What is the appropriate notion of size for each of these functions?

```
#include <stdio.h>
#include <stdlib.h>

typedef struct CELL *LIST;
struct CELL {
    int element;
    LIST next;
};

LIST merge(LIST list1, LIST list2);
LIST split(LIST list);
LIST MergeSort(LIST list);
LIST MakeList();
void PrintList(LIST list);

main()
{
    LIST list;

(1)     list = MakeList();
(2)     PrintList(MergeSort(list));
}

LIST MakeList()
{
    int x;
    LIST pNewCell;

(3)     if (scanf("%d", &x) == EOF) return NULL;
        else {
(4)         pNewCell = (LIST) malloc(sizeof(struct CELL));
(5)         pNewCell->next = MakeList();
(6)         pNewCell->element = x;
(7)         return pNewCell,
        }
}

void PrintList(LIST list)
{
(8)     while (list != NULL) {
(9)         printf("%d\n", list->element);
(10)        list = list->next;
        }
}
```

Fig. 2.31(a). A sorting program using merge sort (start).

```
LIST MergeSort(LIST list)
{
    LIST SecondList;

    if (list == NULL) return NULL;
    else if (list->next == NULL) return list;
    else {
        SecondList = split(list);
        return merge(MergeSort(list), MergeSort(SecondList));
    }
}

LIST merge(LIST list1, LIST list2)
{
    if (list1 == NULL) return list2;
    else if (list2 == NULL) return list1;
    else if (list1->element <= list2->element) {
        list1->next = merge(list1->next, list2);
        return list1;
    }
    else {
        list2->next = merge(list1, list2->next);
        return list2;
    }
}

LIST split(LIST list)
{
    LIST pSecondCell;

    if (list == NULL) return NULL;
    else if (list->next == NULL) return NULL;
    else {
        pSecondCell = list->next;
        list->next = pSecondCell->next;
        pSecondCell->next = split(pSecondCell->next);
        return pSecondCell;
    }
}
```

Fig. 2.31(b). A sorting program using merge sort (conclusion).

2.9 Proving Properties of Recursive Programs

If we want to prove a certain property of a recursive function, we generally need to prove a statement about the effect of one call to that function. For example, that effect might be a relationship between the arguments and the return value, such as "the function, called with argument i, returns $i!$." Frequently, we define a notion of the "size" of the arguments of a function and perform a proof by induction on this

Size of arguments

size. Some of the many possible ways in which size of arguments could be defined are

1. The value of some argument. For instance, for the recursive factorial program of Fig. 2.19, the appropriate size is the value of the argument n.

2. The length of a list pointed to by some argument. The recursive function `split` of Fig. 2.27 is an example where the length of the list is the appropriate size.

3. Some function of the arguments. For instance, we mentioned that the recursive selection sort of Fig. 2.22 performs an induction on the number of elements in the array that remain to be sorted. In terms of the arguments n and i, this function is $n - i + 1$. As another example, the appropriate size for the `merge` function of Fig. 2.24 is the sum of the lengths of the lists pointed to by the two arguments of the function.

Whatever notion of size we pick, it is essential that when a function is called with arguments of size s, it makes only function calls with arguments of size $s - 1$ or less. That requirement is so we can do an induction on the size to prove a property of the program. Further, when the size falls to some fixed value — say 0 — the function must make no recursive calls. This condition is so we can start off our inductive proof with a basis case.

✦ **Example 2.26.** Consider the factorial program of Fig. 2.19 in Section 2.7. The statement to prove by induction on i, for $i \geq 1$, is

STATEMENT $S(i)$: When called with the value i for the argument `n`, `fact` returns $i!$.

BASIS. For $i = 1$, the test at line (1) of Fig. 2.19 causes the basis, line (2), to be executed. That results in the return value 1, which is 1!.

INDUCTION. Assume $S(i)$ to be true, that is, when called with some argument $i \geq 1$, `fact` returns $i!$. Now, consider what happens when `fact` is called with $i + 1$ as the value of variable `n`. If $i \geq 1$, then $i + 1$ is at least 2, so the inductive case, line (3), applies. The return value is thus $n \times fact(n - 1)$; or, since the variable `n` has the value $i + 1$, the result $(i + 1) \times fact(i)$ is returned. By the inductive hypothesis, `fact(i)` returns $i!$. Since $(i + 1) \times i! = (i + 1)!$, we have proved the inductive step, that `fact`, with argument $i + 1$, returns $(i + 1)!$. ✦

✦ **Example 2.27.** Now, let us examine the function `MakeList`, one of the auxiliary routines in Fig. 2.31(a), in Section 2.8. This function creates a linked list to hold the input elements and returns a pointer to this list. We shall prove the following statement by induction on $n \geq 0$, the number of elements in the input sequence.

STATEMENT $S(n)$: If $x_1, x_2, ..., x_n$ is the sequence of input elements, `Makelist` creates a linked list that contains $x_1, x_2, ..., x_n$ and returns a pointer to this list.

BASIS. The basis is $n = 0$, that is, when the input sequence is empty. The test for `EOF` in line (3) of `MakeList` causes the return value to be set to `NULL`. Thus, `MakeList` correctly returns an empty list.

INDUCTION. Suppose that $S(n)$ is true for $n \geq 0$, and consider what happens when `MakeList` is called on an input sequence of $n + 1$ elements. Suppose we have just read the first element x_1.

Line (4) of `MakeList` creates a pointer to a new cell c. Line (5) recursively calls `Makelist` to create, by the inductive hypothesis, a pointer to a linked list for the remaining n elements, $x_2, x_3, \ldots, x_n$. This pointer is put into the next field of c at line (5). Line (6) puts x_1 into the element field of c. Line (7) returns the pointer created by line (4). This pointer points to a linked list for the $n+1$ input elements, $x_1, x_2, \ldots, x_n$.

We have proved the inductive step and conclude that `MakeList` works correctly on all inputs. ✦

✦ **Example 2.28.** For our last example, let us prove the correctness of the merge-sort program of Fig. 2.29, assuming that the functions `split` and `merge` perform their respective tasks correctly. The induction will be on the length of the list that `MergeSort` is given as an argument. The statement to be proved by complete induction on $n \geq 0$ is

STATEMENT $S(n)$: If `list` is a list of length n when `MergeSort` is called, then `MergeSort` returns a sorted list of the same elements.

BASIS. We take the basis to be both $S(0)$ and $S(1)$. When `list` is of length 0, its value is `NULL`, and so the test of line (1) in Fig. 2.29 succeeds and the entire function returns `NULL`. Likewise, if `list` is of length 1, the test of line (2) succeeds, and the function returns `list`. Thus, `MergeSort` returns `list` when n is 0 or 1. This observation proves statements $S(0)$ and $S(1)$, because a list of length 0 or 1 is already sorted.

INDUCTION. Suppose $n \geq 1$ and $S(i)$ is true for all $i = 0, 1, \ldots, n$. We must prove $S(n + 1)$. Thus, consider a list of length $n + 1$. Since $n \geq 1$, this list is of length at least 2, so we reach line (3) in Fig. 2.29. There, `split` divides the list into two lists of length $(n + 1)/2$ if $n + 1$ is even, and of lengths $(n/2) + 1$ and $n/2$ if $n + 1$ is odd. Since $n \geq 1$, none of these lists can be as long as $n + 1$. Thus, the inductive hypothesis applies to them, and we can conclude that the half-sized lists are correctly sorted by the recursive calls to `MergeSort` at line (4). Finally, the two sorted lists are merged into one list, which becomes the return value. We have assumed that `merge` works correctly, and so the resulting returned list is sorted. ✦

```
DefCell(int, CELL, LIST);

int sum(LIST L)
{
    if (L == NULL) sum = 0;
    else sum = L->element + sum(L->next);
}

int find0(LIST L)
{
    if (L == NULL) find0 = FALSE;
    else if (L->element == 0) find0 = TRUE;
    else find0 = find0(L->next);
}
```

Fig. 2.32. Two recursive functions, `sum` and `find0`.

EXERCISES

2.9.1: Prove that the function `PrintList` in Fig. 2.31(b) prints the elements on the list that it is passed as an argument. What statement $S(i)$ do you prove inductively? What is the basis value for i?

2.9.2: The function `sum` in Fig. 2.32 computes the sum of the elements on its given list (whose cells are of the usual type as defined by the macro `DefCell` of Section 1.6 and used in the merge-sort program of Section 2.8) by adding the first element to the sum of the remaining elements; the latter sum is computed by a recursive call on the remainder of the list. Prove that `sum` correctly computes the sum of the list elements. What statement $S(i)$ do you prove inductively? What is the basis value for i?

2.9.3: The function `find0` in Fig. 2.32 returns `TRUE` if at least one of the elements on its list is 0, and returns `FALSE` otherwise. It returns `FALSE` if the list is empty, returns `TRUE` if the first element is 0 and otherwise, makes a recursive call on the remainder of the list, and returns whatever answer is produced for the remainder. Prove that `find0` correctly determines whether 0 is present on the list. What statement $S(i)$ do you prove inductively? What is the basis value for i?

2.9.4*: Prove that the functions (a) `merge` of Fig. 2.24 and (b) `split` of Fig. 2.27 perform as claimed in Section 2.8.

2.9.5: Give an intuitive "least counterexample" proof of why induction starting from a basis including both 0 and 1 is valid.

2.9.6**: Prove the correctness of (your C implementation of) the recursive GCD algorithm of Exercise 2.7.8.

2.10 Summary of Chapter 2

Here are the important ideas we should take from Chapter 2.

- Inductive proofs, recursive definitions, and recursive programs are closely related ideas. Each depends on a basis and an inductive step to "work."
- In "ordinary" or "weak" inductions, successive steps depend only on the previous step. We frequently need need to perform a proof by complete induction, in which each step depends on all the previous steps.
- There are several different ways to sort. Selection sort is a simple but slow sorting algorithm, and merge sort is a faster but more complex algorithm.
- Induction is essential to prove that a program or program fragment works correctly.
- Divide-and-conquer is a useful technique for designing some good algorithms, such as merge sort. It works by dividing the problem into independent subparts and then combining the results.
- Expressions are defined in a natural, recursive way in terms of their operands and operators. Operators can be classified by the number of arguments they take: unary (one argument), binary (two arguments), and k-ary (k arguments). Also, a binary operator appearing between its operands is infix, an operator appearing before its operands is prefix, and one appearing after its operands is postfix.

2.11 Bibliographic Notes for Chapter 2

An excellent treatment of recursion is Roberts [1986]. For more on sorting algorithms, the standard source is Knuth [1973]. Berlekamp [1968] tells about techniques — of which the error detection scheme in Section 2.3 is the simplest — for detecting and correcting errors in streams of bits.

Berlekamp, E. R. [1968]. *Algebraic Coding Theory*, McGraw-Hill, New York.

Knuth, D. E. [1973]. *The Art of Computer Programming*, Vol. III: *Sorting and Searching*, Addison-Wesley, Reading, Mass.

Roberts, E. [1986]. *Thinking Recursively*, Wiley, New York.

CHAPTER 6

The List Data Model

CHAPTER 6

The List Data Model

Like trees, lists are among the most basic of data models used in computer programs. Lists are, in a sense, simple forms of trees, because one can think of a list as a binary tree in which every left child is a leaf. However, lists also present some aspects that are not special cases of what we have learned about trees. For instance, we shall talk about operations on lists, such as pushing and popping, that have no common analog for trees, and we shall talk of character strings, which are special and important kinds of lists requiring their own data structures.

6.1 What This Chapter Is About

We introduce list terminology in Section 6.2. Then in the remainder of the chapter we present the following topics:

- The basic operations on lists (Section 6.3).
- Implementations of abstract lists by data structures, especially the linked-list data structure (Section 6.4) and an array data structure (Section 6.5).
- The stack, a list upon which we insert and delete at only one end (Section 6.6).
- The queue, a list upon which we insert at one end and delete at the other (Section 6.8).
- Character strings and the special data structures we use to represent them (Section 6.10).

Further, we shall study in detail two applications of lists:

- The run-time stack and the way C and many other languages implement recursive functions (Section 6.7).

✦ The problem of finding longest common subsequences of two strings, and its solution by a "dynamic programming," or table-filling, algorithm (Section 6.9).

6.2 Basic Terminology

List

A *list* is a finite sequence of zero or more elements. If the elements are all of type T, then we say that the type of the list is "list of T." Thus we can have lists of integers, lists of real numbers, lists of structures, lists of lists of integers, and so on. We generally expect the elements of a list to be of some one type. However, since a type can be the union of several types, the restriction to a single "type" can be circumvented.

A list is often written with its elements separated by commas and enclosed in parentheses:

$$(a_1, a_2, \ldots, a_n)$$

where the a_i's are the elements of the list.

Character string

In some situations we shall not show commas or parentheses explicitly. In particular, we shall study *character strings*, which are lists of characters. Character strings are generally written with no comma or other separating marks and with no surrounding parentheses. Elements of character strings will normally be written in typewriter font. Thus `foo` is the list of three characters of which the first is `f` and the second and third are `o`.

✦ **Example 6.1.** Here are some examples of lists.

1. The list of prime numbers less than 20, in order of size:

   (2, 3, 5, 7, 11, 13, 17, 19)

2. The list of noble gasses, in order of atomic weight:

   (helium, neon, argon, krypton, xenon, radon)

3. The list of the numbers of days in the months of a non-leap year:

   (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)

As this example reminds us, the same element can appear more than once on a list. ✦

✦ **Example 6.2.** A line of text is another example of a list. The individual characters making up the line are the elements of this list, so the list is a character string. This character string usually includes several occurrences of the blank character, and normally the last character in a line of text is the "newline" character.

As another example, a document can be viewed as a list. Here the elements of the list are the lines of text. Thus a document is a list whose elements that are themselves lists, character strings in particular. ✦

✦ **Example 6.3.** A point in n-dimensional space can be represented by a list of n real numbers. For example, the vertices of the unit cube can be represented by the triples shown in Fig. 6.1. The three elements on each list represent the coordinates of a point that is one of the eight corners (or "vertices") of the cube. The first component represents the x-coordinate (horizontal), the second represents the y-coordinate (into the page), and the third represents the z-coordinate (vertical). ✦

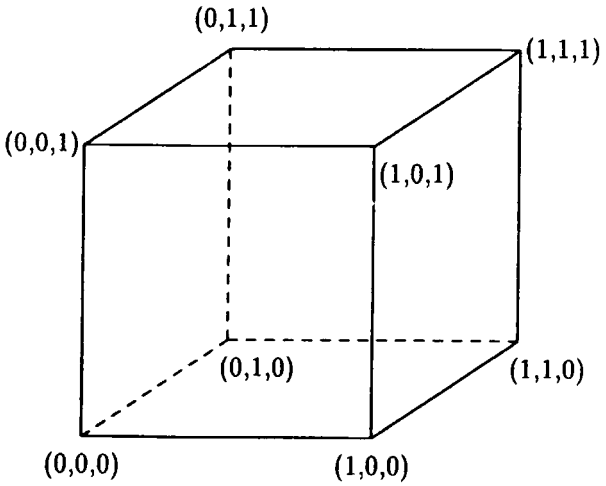


Fig. 6.1. The vertices of the unit cube represented as triples.

The Length of a List

Empty list

The *length* of a list is the number of occurrences of elements on the list. If the number of elements is zero, then the list is said to be *empty*. We use the Greek letter ϵ (epsilon) to represent the empty list. We can also represent the empty list by a pair of parentheses surrounding nothing: (). It is important to remember that length counts positions, not distinct symbols, and so a symbol appearing k times on a list adds k to the length of the list.

✦ **Example 6.4.** The length of list (1) in Example 6.1 is 8, and the length of list (2) is 6. The length of list (3) is 12, since there is one position for each month. The fact that there are only three different numbers on the list is irrelevant as far as the length of the list is concerned. ✦

Parts of a List

Head and tail of a list

If a list is not empty, then it consists of a first element, called the *head* and the remainder of the list, called the *tail*. For instance, the head of list (2) in Example 6.1 is helium, while the tail is the list consisting of the remaining five elements,

(neon, argon, krypton, xenon, radon)

Elements and Lists of Length 1

It is important to remember that the head of a list is an element, while the tail of a list is a list. Moreover, we should not confuse the head of a list — say a — with the list of length 1 containing only the element a, which would normally be written with parentheses as (a). If the element a is of type T, then the list (a) is of type "list of T."

Failure to recognize the difference leads to programming errors when we implement lists by data structures. For example, we may represent lists by linked cells, which are typically structures with an **element** field of some type T, holding an element, and a **next** field holding a pointer to the next cell. Then element a is of type T, while the list (a) is a cell with **element** field holding a and **next** field holding **NULL**.

Sublist

If $L = (a_1, a_2, \ldots, a_n)$ is a list, then for any i and j such that $1 \leq i \leq j \leq n$, $(a_i, a_{i+1}, \ldots, a_j)$ is said to be a *sublist* of L. That is, a sublist is formed by starting at some position i, and taking all the elements up to some position j. We also say that ϵ, the empty list, is a sublist of any list.

Subsequence

A *subsequence* of the list $L = (a_1, a_2, \ldots, a_n)$ is a list formed by striking out zero or more elements of L. The remaining elements, which form the subsequence, must appear in the same order in which they appear in L, but the elements of the subsequence need not be consecutive in L. Note that ϵ and the list L itself are always subsequences, as well as sublists, of L.

✦ **Example 6.5.** Let L be the character string **abc**. The sublists of L are

ϵ, **a**, **b**, **c**, **ab**, **bc**, **abc**

These are all subsequences of L as well, and in addition, **ac** is a subsequence, but not a sublist.

For another example, let L be the character string **abab**. Then the sublists are

ϵ, **a**, **b**, **ab**, **ba**, **aba**, **bab**, **abab**

These are also subsequences of L, and in addition, L has the subsequences **aa**, **bb**, **aab**, and **abb**. Notice that a character string like **bba** is not a subsequence of L. Even though L has two **b**'s and an **a**, they do not appear in such an order in L that we can form **bba** by striking out elements of L. That is, there is no **a** after the second **b** in L. ✦

Prefix and suffix

A *prefix* of a list is any sublist that starts at the beginning of the list. A *suffix* is a sublist that terminates at the end of the list. As special cases, we regard ϵ as both a prefix and a suffix of any list.

✦ **Example 6.6.** The prefixes of the list **abc** are ϵ, **a**, **ab**, and **abc**. Its suffixes are ϵ, **c**, **bc**, and **abc**. ✦

Car and Cdr

In the programming language Lisp, the head is called the *car* and the tail is called the *cdr* (pronounced "cudder"). The terms "*car*" and "*cdr*" arose from the names given to two fields of a machine instruction on an IBM 709, the computer on which Lisp was first implemented. Car stands for "contents of the address register," and cdr stands for "contents of the decrement register." In a sense, memory words were seen as cells with **element** and **next** fields, corresponding to the car and cdr, respectively.

The Position of an Element on a List

Each element on a list is associated with a *position*. If $(a_1, a_2, \ldots, a_n)$ is a list and $n \geq 1$, then a_1 is said to be *first* element, a_2 the second, and so on, with a_n the *last* element. We also say that element a_i occurs at *position* i. In addition, a_i is said to *follow* a_{i-1} and to *precede* a_{i+1}. A position holding element a is said to be an *occurrence* of a.

Occurrence of an element

The number of positions on a list equals the length of the list. It is possible for the same element to appear at two or more positions. Thus it is important not to confuse a position with the element at that position. For instance, list (3) in Example 6.1 has twelve positions, seven of which hold 31 — namely, positions 1, 3, 5, 7, 8, 10, and 12.

EXERCISES

6.2.1: Answer the following questions about the list $(2, 7, 1, 8, 2)$.

a) What is the length?
b) What are all the prefixes?
c) What are all the suffixes?
d) What are all the sublists?
e) How many subsequences are there?
f) What is the head?
g) What is the tail?
h) How many positions are there?

6.2.2: Repeat Exercise 6.2.1 for the character string **banana**.

6.2.3**: In a list of length $n \geq 0$, what are the largest and smallest possible numbers of (a) prefixes (b) sublists (c) subsequences?

6.2.4: If the tail of the tail of the list L is the empty list, what is the length of L?

6.2.5*: Bea Fuddled wrote a list whose elements are themselves lists of integers, but omitted the parentheses: 1,2,3. There are many lists of lists that could have been represented, such as $((1), (2, 3))$. What are all the possible lists that do not have the empty list as an element?

6.3 Operations on Lists

Sorting and merging

A great variety of operations can be performed on lists. In Chapter 2, when we discussed merge sort, the basic problem was to sort a list, but we also needed to split a list into two, and to merge two sorted lists. Formally, the operation of *sorting* a list $(a_1, a_2, \ldots, a_n)$ amounts to replacing the list with a list consisting of a permutation of its elements, $(b_1, b_2, \ldots, b_n)$, such that $b_1 \leq b_2 \leq \cdots \leq b_n$. Here, as before, $\leq$ represents an ordering of the elements, such as "less than or equal to" on integers or reals, or lexicographic order on strings. The operation of *merging* two sorted lists consists of constructing from them a sorted list containing the same elements as the two given lists. Multiplicity must be preserved; that is, if there are k occurrences of element a among the two given lists, then the resulting list has k occurrences of a. Review Section 2.8 for examples of these two operations on lists.

Insertion, Deletion, and Lookup

Recall from Section 5.7 that a "dictionary" is a set of elements on which we perform the operations *insert*, *delete*, and *lookup*. There is an important difference between sets and lists. An element can never appear more than once in a set, although, as we have seen, an element can appear more than once on a list; this and other issues regarding sets are discussed in Chapter 7. However, lists can implement sets, in the sense that the elements in a set $\{a_1, a_2, \ldots, a_n\}$ can be placed in a list in any order, for example, the order $(a_1, a_2, \ldots, a_n)$, or the order $(a_n, a_{n-1}, \ldots, a_1)$. Thus it should be no surprise that there are operations on lists analogous to the dictionary operations on sets.

1. We can *insert* an element x onto a list L. In principle, x could appear anywhere on the list, and it does not matter if x already appears in L one or more times. We insert by adding one more occurrence of x. As a special case, if we make x the head of the new list (and therefore make L the tail), we say that we *push* x onto the list L. If $L = (a_1, \ldots, a_n)$, the resulting list is $(x, a_1, \ldots, a_n)$.

2. We can *delete* an element x from a list L. Here, we delete an occurrence of x from L. If there is more than one occurrence, we could specify which occurrence to delete; for example, we could always delete the first occurrence. If we want to delete all occurrences of x, we repeat the deletion until no more x's remain. If x is not present on list L, the deletion has no effect. As a special case, if we delete the head element of the list, so that the list $(x, a_1, \ldots, a_n)$ becomes $(a_1, \ldots, a_n)$, we are said to *pop* the list.

3. We can *lookup* an element x on a list L. This operation returns **TRUE** or **FALSE**, depending on whether x is or is not an element on the list.

✦ **Example 6.7.** Let L be the list $(1, 2, 3, 2)$. The result of *insert*$(1, L)$ could be the list $(1, 1, 2, 3, 2)$, if we chose to push 1, that is, to insert 1 at the beginning. We could also insert the new 1 at the end, yielding $(1, 2, 3, 2, 1)$. Alternatively, the new 1 could be placed in any of three positions interior to the list L.

The result of *delete*$(2, L)$ is the list $(1, 3, 2)$ if we delete the first occurrence of 2. If we ask *lookup*(x, L), the answer is **TRUE** if x is 1, 2, or 3, but **FALSE** if x is any other value. ✦

Concatenation

We *concatenate* two lists L and M by forming the list that begins with the elements of L and then continues with the elements of M. That is, if $L = (a_1, a_2, \ldots, a_n)$ and $M = (b_1, b_2, \ldots, b_k)$, then LM, the concatenation of L and M, is the list

$$(a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_k)$$

Identity for concatenation

Note that the empty list is the identity for concatenation. That is, for any list L, we have $\epsilon L = L\epsilon = L$.

✦ **Example 6.8.** If L is the list $(1, 2, 3)$ and M is the list $(3, 1)$, then LM is the list $(1, 2, 3, 3, 1)$. If L is the character string **dog** and M is the character string **house**, then LM is the character string **doghouse**. ✦

Other List Operations

Another family of operations on lists refers to particular positions of the list. For example,

First and last of lists

a) The operation $first(L)$ returns the first element (head) of list L, and $last(L)$ returns the last element of L. Both cause an error if L is an empty list.

Retrieve

b) The operation $retrieve(i, L)$ returns the element at the ith position of list L. It is an error if L has length less than i.

There are additional operations that involve the length of a list. Some common operations are:

c) $length(L)$, which returns the length of list L.

isEmpty

d) $isEmpty(L)$, which returns **TRUE** if L is an empty list and returns **FALSE** if not. Similarly, $isNotEmpty(L)$ would return the opposite.

EXERCISES

6.3.1: Let L be the list $(3, 1, 4, 1, 5, 9)$.

a) What is the value of $delete(5, L)$?
b) What is the value of $delete(1, L)$?
c) What is the result of popping L?
d) What is the result of pushing 2 onto list L?
e) What is returned if we perform *lookup* with the element 6 and list L?
f) If M is the list $(6, 7, 8)$, what is the value of LM (the concatenation of L and M)? What is ML?
g) What is $first(L)$? What is $last(L)$?
h) What is the result of $retrieve(3, L)$?
i) What is the value of $length(L)$?
j) What is the value of $isEmpty(L)$?

6.3.2**: If L and M are lists, under what conditions is $LM = ML$?

6.3.3**: Let x be an element and L a list. Under what conditions are the following equations true?

a) $delete(x, insert(x, L)) = L$
b) $insert(x, delete(x, L)) = L$
c) $first(L) = retrieve(1, L)$
d) $last(L) = retrieve(length(L) \quad L)$

6.4 The Linked-List Data Structure

The easiest way to implement a list is to use a linked list of cells. Each cell consists of two fields, one containing an element of the list, the other a pointer to the next cell on the linked list. In this chapter we shall make the simplifying assumption that elements are integers. Not only may we use the specific type `int` for the type of elements, but we can compare elements by the standard comparison operators ==, <, and so on. The exercises invite the reader to develop variants of our functions that work for arbitrary types, where comparisons are made by user-defined functions such as *eq* to test equality, $lt(x, y)$ to test if x precedes y in some ordering, and so on.

In what follows, we shall use our macro from Section 1.6:

```
DefCell(int, CELL, LIST);
```

which expands into our standard structure for cells and lists:

```
typedef struct CELL *LIST;
struct CELL {
    int element;
    LIST next;
};
```

Note that `LIST` is the type of a pointer to a cell. In effect, the `next` field of each cell points both to the next cell and to the entire remainder of the list.

Figure 6.2 shows a linked list that represents the abstract list

$$L = (a_1, a_2, \ldots, a_n)$$

There is one cell for each element; the element a_i appears in the `element` field of the ith cell. The pointer in the ith cell points to the $(i+1)$st cell, for $i = 1, 2, \ldots, n-1$, and the pointer in the last cell is `NULL`, indicating the end of the list. Outside the list is a pointer, named `L`, that points to the first cell of the list; `L` is of type `LIST`. If the list L were empty, then the value of `L` would be `NULL`.

Fig. 6.2. A linked list representing the list $L = (a_1, a_2, \ldots, a_n)$.

Implementation of Dictionary Operations by Linked Lists

Let us consider how we can implement the dictionary operations if we represent the dictionary by a linked list. The following operations on dictionaries were defined in Section 5.7.

Lists and Linked Lists

Remember that a list is an abstract, or mathematical model. The linked list is a simple data structure, which was mentioned in Chapter 1. A linked list is one way to implement the list data model, although, as we shall see, it is not the only way. At any rate, this is a good time to remember once more the distinction between models and the data structures that implement them.

1. *insert*(x, D), to insert element x into dictionary D,
2. *delete*(x, D), to delete element x from dictionary D, and
3. *lookup*(x, D), to determine whether element x is in dictionary D,

We shall see that the linked list is a simpler data structure for implementing dictionaries than the binary search tree that we discussed in the previous chapter. However, the running time of the dictionary operations when using the linked-list representation is not as good as when using the binary search tree. In Chapter 7 we shall see an even better implementation for dictionaries — the hash table — which makes use, as subroutines, of the dictionary operations on lists.

We shall assume that our dictionary contains integers, and cells are defined as they were at the beginning of this section. Then the type of a dictionary is `LIST`, also as defined at the beginning of this section. The dictionary containing the set of elements $\{a_1, a_2, \ldots, a_n\}$ could be represented by the linked list of Fig. 6.2. There are many other lists that could represent the same set, since order of elements is not important in sets.

Lookup

To perform *lookup*(x, D), we examine each cell of the list representing D to see whether it holds the desired element x. If so, we return `TRUE`. If we reach the end of the list without finding x, we return `FALSE`. As before, the defined constants `TRUE` and `FALSE` stand for the constants 1 and 0, and `BOOLEAN` for the defined type `int`. A recursive function `lookup(x,D)` is shown in Fig. 6.3.

```
BOOLEAN lookup(int x, LIST L)
{
    if (L == NULL)
        return FALSE;
    else if (x == L->element)
        return TRUE;
    else
        return lookup(x, L->next);
}
```

Fig. 6.3. Lookup on a linked list.

If the list has length n, we claim that the function of Fig. 6.3 takes $O(n)$ time. Except for the recursive call at the end, `lookup` takes $O(1)$ time. When the call is

made, the length of the remaining list is 1 less than the length of the list L. Thus it should surprise no one that `lookup` on a list of length n takes $O(n)$ time. More formally, the following recurrence relation gives the running time $T(n)$ of `lookup` when the list L pointed to by the second argument has length n.

BASIS. $T(0)$ is $O(1)$, because there is no recursive call when L is `NULL`.

INDUCTION. $T(n) = T(n-1) + O(1)$.

The solution to this recurrence is $T(n) = O(n)$, as we saw several times in Chapter 3. Since a dictionary of n elements is represented by a list of length n, *lookup* takes $O(n)$ time on a dictionary of size n.

Unfortunately, the average time for a successful lookup is also proportional to n. For if we are looking for an element x known to be in D, the expected value of the position of x in the list is $(n+1)/2$. That is, x could be anywhere from the first to the nth element, with equal probability. Thus, the expected number of recursive calls to `lookup` is $(n+1)/2$. Since each takes $O(1)$ time, the average successful lookup takes $O(n)$ time. Of course, if the lookup is unsuccessful, we make all n calls before reaching the end of the list and returning `FALSE`.

Deletion

A function to delete an element x from a linked list is shown in Fig. 6.4. The second parameter `pL` is a pointer to the list L (rather than the list L itself). We use the "call by reference" style here because we want `delete` to remove the cell containing x from the list. As we move down the list, `pL` holds a pointer to a pointer to the "current" cell. If we find x in the current cell C, at line (2), then we change the pointer to cell C at line (3), so that it points to the cell following C on the list. If C happens to be last on the list, the former pointer to C becomes `NULL`. If x is not the current element, then at line (4) we recursively delete x from the tail of the list.

Note that the test at line (1) causes the function to return with no action if the list is empty. That is because x is not present on an empty list, and we need not do anything to remove x from the dictionary. If D is a linked list representing a dictionary, then a call to `delete(x, &D)` initiates the deletion of x from the dictionary D.

```
     void delete(int x, LIST *pL)
     {
(1)      if ((*pL) != NULL)
(2)          if (x == (*pL)->element)
(3)              (*pL) = (*pL)->next;
             else
(4)              delete(x, &((*pL)->next));
     }
```

Fig. 6.4. Deleting an element.

If the element x is not on the list for the dictionary D, then we run down to the end of the list, taking $O(1)$ time for each element. The analysis is similar to

that for the `lookup` function of Fig. 6.3, and we leave the details for the reader. Thus the time to delete an element not in D is $O(n)$ if D has n elements. If x is in the dictionary D, then, on the average, we shall encounter x halfway down the list. Therefore we search $(n+1)/2$ cells on the average, and the running time of a successful deletion is also $O(n)$.

Insertion

A function to insert an element x into a linked list is shown in Fig. 6.5. To insert x, we need to check that x is not already on the list (if it is, we do nothing). If x is not already present, we must add it to the list. It does not matter where in the list we add x, but the function in Fig. 6.5 adds x to the end of the list. When at line (1) we detect the `NULL` at the end, we are therefore sure that x is not already on the list. Then, lines (2) through (4) append x to the end of the list.

If the list is not `NULL`, line (5) checks for x at the current cell. If x is not there, line (6) makes a recursive call on the tail. If x is found at line (5), then function `insert` terminates with no recursive call and with no change to the list L. A call to `insert(x, &D)` initiates the insertion of x into dictionary D.

```
     void insert(int x, LIST *pL)
     {
(1)      if ((*pL) == NULL) {
(2)          (*pL) = (LIST) malloc(sizeof(struct CELL));
(3)          (*pL)->element = x;
(4)          (*pL)->next = NULL;
         }
(5)      else if (x != (*pL)->element)
(6)          insert(x, &((*pL)->next));
     }
```

Fig. 6.5. Inserting an element.

As in the case of lookup and deletion, if we do not find x on the list, we travel to the end, taking $O(n)$ time. If we do find x, then on the average we travel halfway[1] down the list, and we still take $O(n)$ time on the average.

A Variant Approach with Duplicates

We can make insertion run faster if we do not check for the presence of x on the list before inserting it. However, as a consequence, there may be several copies of an element on the list representing a dictionary.

To execute the dictionary operation *insert*(x, D) we simply create a new cell, put x in it, and push that cell onto the front of the list for D. This operation takes $O(1)$ time.

The lookup operation is exactly the same as in Fig. 6.3. The only nuance is that we may have to search a longer list, because the length of the list representing dictionary D may be greater than the number of members of D.

[1] In the following analyses, we shall use terms like "halfway" or "$n/2$" when we mean the middle of a list of length n. Strictly speaking, $(n+1)/2$ is more accurate.

Abstraction Versus Implementation Again

It may be surprising to see us using duplicates in lists that represent dictionaries, since the abstract data type DICTIONARY is defined as a set, and sets do not have duplicates. However, it is not the dictionary that has duplicates. Rather the data structure implementing the dictionary is allowed to have duplicates. But even when x appears several times on a linked list, it is only present once in the dictionary that the linked list represents.

Deletion is slightly different. We cannot stop our search for x when we encounter a cell with element x, because there could be other copies of x. Thus we must delete x from the tail of a list L, even when the head of L contains x. As a result, not only do we have longer lists to contend with, but to achieve a successful deletion we must search every cell rather than an average of half the list, as we could for the case in which no duplicates were allowed on the list. The details of these versions of the dictionary operations are left as an exercise.

In summary, by allowing duplicates, we make insertion faster, $O(1)$ instead of $O(n)$. However, successful deletions require search of the entire list, rather than an average of half the list, and for both lookup and deletion, we must contend with lists that are longer than when duplicates are not allowed, although how much longer depends on how often we insert an element that is already present in the dictionary.

Which method to choose is a bit subtle. Clearly, if insertions predominate, we should allow duplicates. In the extreme case, where we do only insertions but never lookup or deletion, we get performance of $O(1)$ per operation, instead of $O(n)$.[2] If we can be sure, for some reason, that we shall never insert an element already in the dictionary, then we can use both the fast insertion and the fast deletion, where we stop when we find one occurrence of the element to be deleted. On the other hand, if we may insert duplicate elements, and lookups or deletions predominate, then we are best off checking for the presence of x before inserting it, as in the *insert* function of Fig. 6.5.

Sorted Lists to Represent Dictionaries

Another alternative is to keep elements sorted in increasing order on the list representing a dictionary D. Then, if we wish to lookup element x, we have only to go as far as the position in which x would appear; on the average, that is halfway down the list. If we meet an element greater than x, then there is no hope of finding x later in the list. We thus avoid going all the way down the list on unsuccessful searches. That saves us about a factor of 2, although the exact factor is somewhat clouded because we have to ask whether x follows in sorted order each of the elements we meet on the list, which is an additional step at each cell. However, the same factor in savings is gained on unsuccessful searches during insertion and deletion.

A lookup function for sorted lists is shown in Fig. 6.6. We leave to the reader the exercise of modifying the functions of Figs. 6.4 and 6.5 to work on sorted lists.

[2] But why bother inserting into a dictionary if we never look to see what is there?

```
BOOLEAN lookup(int x, LIST L)
{
    if (L == NULL)
        return FALSE;
    else if (x > L->element)
        return lookup(x, L->next);
    else if (x == L->element)
        return TRUE;
    else /* here x < L->element, and so x could not be
            on the sorted list L */
        return FALSE;
}
```

Fig. 6.6. Lookup on a sorted list.

Comparison of Methods

The table in Fig. 6.7 indicates the number of cells we must search for each of the three dictionary operations, for each of the three list-based representations of dictionaries we have discussed. We take n to be the number of elements in the dictionary, which is also the length of the list if no duplicates are allowed. We use m for the length of the list when duplicates are allowed. We know that $m \geq n$, but we do not know how much greater m is than n. Where we use $n/2 \rightarrow n$ we mean that the number of cells is an average of $n/2$ when the search is successful, and n when unsuccessful. The entry $n/2 \rightarrow m$ indicates that on a successful lookup we shall see $n/2$ elements of the dictionary, on the average, before seeing the one we want,[3] but on an unsuccessful search, we must go all the way to the end of a list of length m.

	INSERT	DELETE	LOOKUP
No duplicates	$n/2 \rightarrow n$	$n/2 \rightarrow n$	$n/2 \rightarrow n$
Duplicates	0	m	$n/2 \rightarrow m$
Sorted	$n/2$	$n/2$	$n/2$

Fig. 6.7. Number of cells searched by three methods of representing dictionaries by linked lists.

Notice that all of these running times, except for insertion with duplicates, are worse than the average running times for dictionary operations when the data structure is a binary search tree. As we saw in Section 5.8, dictionary operations take only $O(\log n)$ time on the average when a binary search tree is used.

[3] In fact, since there may be duplicates, we may have to examine somewhat more than $n/2$ cells before we can expect to see $n/2$ different elements.

Judicious Ordering of Tests

Notice the order in which the three tests of Fig. 6.6 are made. We have no choice but to test that L is not `NULL` first, since if L is `NULL` the other two tests will cause an error. Let y be the value of `L->element`. Then in all but the last cell we visit, we shall have $x < y$. The reason is that if we have $x = y$, we terminate the lookup successfully, and if we have $x > y$, we terminate with failure to find x. Thus we make the test $x < y$ first, and only if that fails do we need to separate the other two cases. That ordering of tests follows a general principle: we want to test for the most common cases first, and thus save in the total number of tests we perform, on the average.

If we visit k cells, then we test k times whether L is `NULL` and we test k times whether x is less than y. Once, we shall test whether $x = y$, making a total of $2k + 1$ tests. That is only one more test than we make in the `lookup` function of Fig. 6.3 — which uses unsorted lists — in the case that the element x is found. If the element is not found, we shall expect to use many fewer tests in Fig. 6.6 than in Fig. 6.3, because we can, on the average, stop after examining only half the cells with Fig. 6.6. Thus although the big-oh running times of the dictionary operations using either sorted or unsorted lists is $O(n)$, there is usually a slight advantage in the constant factor if we use sorted lists.

Doubly Linked Lists

In a linked list it is not easy to move from a cell toward the beginning of the list. The doubly linked list is a data structure that facilitates movement both forward and backward in a list. The cells of a doubly linked list of integers contain three fields:

```
typdef struct CELL *LIST;
struct CELL {
    LIST previous;
    int element;
    LIST next;
};
```

The additional field contains a pointer to the previous cell on the list. Figure 6.8 shows a doubly linked list data structure that represents the list $L = (a_1, a_2, \ldots, a_n)$.

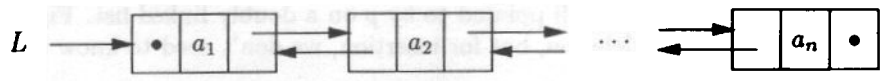


Fig. 6.8. A doubly linked list representing the list $L = (a_1, a_2, \ldots, a_n)$.

Dictionary operations on a doubly linked list structure are essentially the same as those on a singly linked list. To see the advantage of doubly linked lists, consider the operation of deleting an element a_i, given only a pointer to the cell containing that element. With a singly linked list, we would have to find the previous cell by searching the list from the beginning. With a doubly linked list, we can do the task in $O(1)$ time by a sequence of pointer manipulations, as shown in Fig. 6.9.

```
    void delete(LIST p, LIST *pL)
    {
        /* p is a pointer to the cell to be deleted,
          and pL points to the list */
(1)     if (p->next != NULL)
(2)         p->next->previous = p->previous;
(3)     if (p->previous == NULL) /* p points to first cell */
(4)         (*pL) = p->next;
        else
(5)         p->previous->next = p->next;
    }
```

Fig. 6.9. Deleting a cell from a doubly linked list.

The function **delete(p,pL)** shown in Fig. 6.9 takes as arguments a pointer p to the cell to be deleted, and pL, which is a pointer to the list L itself. That is, pL is the address of a pointer to the first cell on the list. In line (1) of Fig. 6.9 we check that p does not point to the last cell. If it does not, then at line (2) we make the backward pointer of the following cell point to the cell before p (or we make it equal to **NULL** if p happens to point to the first cell).

Line (3) tests whether p is the first cell. If so, then at line (4) we make L point to the second cell. Note that in this case, line (2) has made the **previous** field of the second cell **NULL**. If p does not point to the first cell, then at line (5) we make the forward pointer of the previous cell point to the cell following p. That way, the cell pointed to by p has effectively been spliced out of the list; the previous and next cells point to each other.

EXERCISES

6.4.1: Set up the recurrence relations for the running times of (a) **delete** in Fig. 6.4 (b) **insert** in Fig. 6.5. What are their solutions?

6.4.2: Write C functions for dictionary operations *insert*, *lookup* and *delete* using linked lists with duplicates.

6.4.3: Write C functions for *insert* and *delete*, using sorted lists as in Fig. 6.6.

6.4.4: Write a C function that inserts an element x into a new cell that follows the cell pointed to by p on a doubly linked list. Figure 6.9 is a similar function for deletion, but for insertion, we don't need to know the list header L.

6.4.5: If we use the doubly linked data structure for lists, an option is to represent a list not by a pointer to a cell, but by a cell with the element field unused. Note that this "header" cell is not itself a part of the list. The **next** field of the header points to the first true cell of the list, and the **previous** field of the first cell points to the header cell. We can then delete the cell (not the header) pointed to by pointer p without knowing the header L, as we needed to know in Fig. 6.9. Write a C function to delete from a doubly linked list using the format described here.

6.4.6: Write recursive functions for (a) *retrieve*(i, L) (b) *length*(L) (c) *last*(L) using the linked-list data structure.

6.4.7: Extend each of the following functions to cells with an arbitrary type **ETYPE** for elements, using functions $eq(x, y)$ to test if x and y are equal and $lt(x, y)$ to tell if x precedes y in an ordering of the elements of **ETYPE**.

a) *lookup* as in Fig. 6.3.

b) *delete* as in Fig. 6.4.

c) *insert* as in Fig. 6.5.

d) *insert*, *delete*, and *lookup* using lists with duplicates.

e) *insert*, *delete*, and *lookup* using sorted lists.

6.5 Array-Based Implementation of Lists

Another common way to implement a list is to create a structure consisting of

1. An array to hold the elements and
2. A variable `length` to keep track of the count of the number of elements currently in the list.

Figure 6.10 shows how we might represent the list $(a_0, a_1, \ldots, a_{n-1})$ using an array `A[0..MAX-1]`. Elements $a_0, a_1, \ldots, a_{n-1}$ are stored in `A[0..n-1]`, and $length = n$.

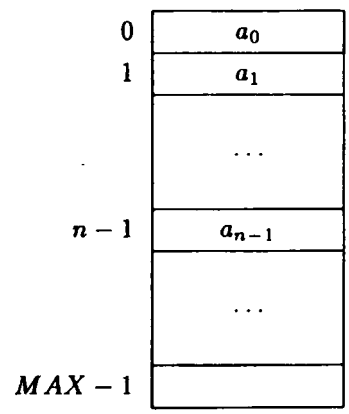


Fig. 6.10. The array `A` holding the list $(a_0, a_1, \ldots, a_{n-1})$.

As in the previous section, we assume that list elements are integers and invite the reader to generalize the functions to arbitrary types. The structure declaration for this array-based implementation of lists is:

```
typedef struct {
    int A[MAX];
    int length;
} LIST;
```

Here, LIST is a structure of two fields; the first is an array A that stores the elements, the second an integer **length** that contains the number of elements currently on the list. The quantity **MAX** is a user-defined constant that bounds the number of elements that will ever be stored on the list.

The array-based representation of lists is in many ways more convenient than the linked-list representation. It does suffer, however, from the limitation that lists cannot grow longer than the array, which can cause an insertion to fail. In the linked-list representation, we can grow lists as long as we have available computer memory.

We can perform the dictionary operations on array-based lists in roughly the same time as on lists in the linked-list representation. To insert x, we look for x, and if we do not find it, we check whether $length < MAX$. If not, there is an error condition, as we cannot fit the new element into the array. Otherwise, we store x in **A[length]** and then increase *length* by 1. To delete x, we again lookup x, and if found, we shift all following elements of **A** down by one position; then, we decrement *length* by 1. The details of functions to implement *insert* and *delete* are left as exercises. We shall concentrate on the details of *lookup* below.

Lookup with Linear Search

Figure 6.11 is a function that implements the operation *lookup*. Because the array **A** may be large, we choose to pass a pointer **pL** to the structure of type LIST as a formal parameter to *lookup*. Within the function, the two fields of the structure can then be referred to as **pL->A[i]** and **pL->length**.

Starting with $i = 0$, the for-loop of lines (1) to (3) examines each location of the array in turn, until it either reaches the last occupied location or finds x. If it finds x, it returns **TRUE**. If it has examined each element in the list without finding x, it returns **FALSE** at line (4). This method of lookup is called *linear* or *sequential* search.

```
    BOOLEAN lookup(int x, LIST *pL)
    {
        int i;

(1)     for (i = 0; i < pL->length; i++)
(2)         if (x == pL->A[i])
(3)             return TRUE;
(4)     return FALSE;
    }
```

Fig. 6.11. Function that does lookup by linear search.

It is easy to see that, on the average, we search half the array **A[0..length-1]** before finding x if it is present. Thus letting n be the value of **length**, we take $O(n)$ time to perform a lookup. If x is not present, we search the whole array **A[0..length-1]**, again requiring $O(n)$ time. This performance is the same as for a linked-list representation of a list.

The Importance of Constant Factors in Practice

Throughout Chapter 3, we emphasized the importance of big-oh measures of running time, and we may have given the impression that big-oh is all that matters or that any $O(n)$ algorithm is as good as any other $O(n)$ algorithm for the same job. Yet here, in our discussion of sentinels, and in other sections, we have examined rather carefully the constant factor hidden by the $O(n)$. The reason is simple. While it is true that big-oh measures of running time dominate constant factors, it is also true that everybody studying the subject learns that fairly quickly. We learn, for example, to use an $O(n \log n)$ running time sort whenever n is large enough to matter. A competitive edge in the performance of software frequently comes from improving the constant factor in an algorithm that already has the right "big-oh" running time, and this edge frequently translates into the success or failure of a commercial software product.

Look with Sentinels

We can simplify the code in the for-loop of Fig 6.11 and speed up the program by temporarily inserting x at the end of the list. This x at the end of the list is called a *sentinel*. The technique was first mentioned in a box on "More Defensive Programming" in Section 3.6, and it has an important application here. Assuming that there always is an extra slot at the end of the list, we can use the program in Fig. 6.12 to search for x. The running time of the program is still $O(n)$, but the constant of proportionality is smaller because the number of machine instructions required by the body and test of the loop will typically be smaller for Fig. 6.12 than for Fig. 6.11.

```
    BOOLEAN lookup(int x, LIST *pL)
    {
        int i;

(1)     pL->A[pL->length] = x;
(2)     i = 0;
(3)     while (x != pL->A[i])
(4)         i++;
(5)     return (i < pL->length);
    }
```

Fig. 6.12. Function that does lookup with a sentinel.

The sentinel is placed just beyond the list by line (1). Note that since *length* does not change, this x is not really part of the list. The loop of lines (3) and (4) increases i until we find x. Note that we are guaranteed to find x even if the list is empty, because of the sentinel. After finding x, we test at line (5) whether we have found a real occurrence of x (that is, $i <$ *length*), or whether we have found the sentinel (that is, $i =$ *length*). Note that if we are using a sentinel, it is essential that *length* be kept strictly less than *MAX*, or else there will be no place to put the sentinel.

Lookup on Sorted Lists with Binary Search

Suppose L is a list in which the elements $a_0, a_1, \ldots, a_{n-1}$ have been sorted in nondecreasing order. If the sorted list is stored in an array `A[0..n-1]`, we can speed lookups considerably by using a technique known as *binary search*. We must first find the index m of the middle element; that is, $m = \lfloor (n-1)/2 \rfloor$.[4] Then we compare element x with $A[m]$. If they are equal, we have found x. If $x < A[m]$, we recursively repeat the search on the sublist `A[0..m-1]`. If $x > A[m]$, we recursively repeat the search on the sublist `A[m+1..n-1]`. If at any time we try to search an empty list, we report failure. Figure 6.13 illustrates the division process.

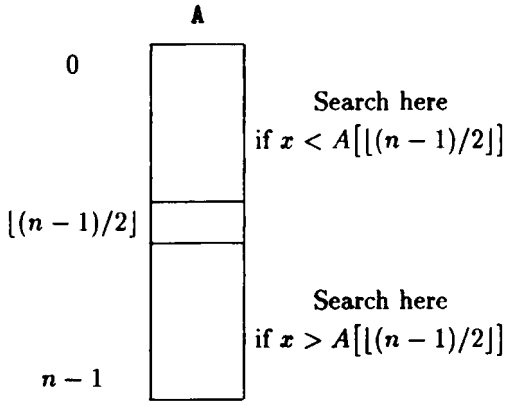


Fig. 6.13. Binary search divides a range in two.

The code for a function *binsearch* to locate x in a sorted array A is shown in Fig. 6.14. The function uses the variables `low` and `high` for the lower and upper bounds of the range in which x might lie. If the lower range exceeds the upper, we have failed to find x; the function terminates and returns `FALSE`.

Otherwise, *binsearch* computes the midpoint of the range, by

$$mid = \lfloor (low + high)/2 \rfloor$$

Then the function examines $A[mid]$, the element at the middle of the range, to determine whether x is there. If not, it continues the search in the lower or upper half of the range, depending on whether x is less than or greater than $A[mid]$. This idea generalizes the division suggested in Fig. 6.13, where *low* was 0 and *high* was $n-1$.

The function `binsearch` can be proved correct using the inductive assertion that if x is in the array, then it must lie within the range `A[low..high]`. The proof is by induction on the difference *high* − *low* and is left as an exercise.

At each iteration, *binsearch* either

1. Finds the element x when it reaches line (8) or

[4] The notation $\lfloor a \rfloor$, the *floor* of a, is the integer part of a. Thus $\lfloor 6.5 \rfloor = 6$ and $\lfloor 6 \rfloor = 6$. Also, $\lceil a \rceil$, the *ceiling* of a, is the smallest integer greater than or equal to a. For instance, $\lceil 6.5 \rceil = 7$ and $\lceil 6 \rceil = 6$.

```
    BOOLEAN binsearch(int x, int A[], int low, int high)
    {
        int mid;

(1)     if (low > high)
(2)         return FALSE;
        else {
(3)         mid = (low + high)/2;
(4)         if (x < A[mid])
(5)             return binsearch(x, A, low, mid-1);
(6)         else if (x > A[mid])
(7)             return binsearch(x, A, mid+1, high);
            else /* x == A[mid] */
(8)             return TRUE;
        }
    }
```

Fig. 6.14. Function that does lookup using binary search.

2. Calls itself recursively at line (5) or line (7) on a sublist that is at most half as long as the array `A[low..high]` that it was given to search.

Starting with an array of length n, we cannot divide the length of the array to be searched in half more than $\log_2 n$ times before it has length 1, whereupon we either find x at `A[mid]`, or we fail to find x at all after a call on the empty list.

To look for x in an array `A` with n elements, we call `binsearch(x,A,0,n-1)`. We see that `binsearch` calls itself $O(\log n)$ times at most. At each call, we spend $O(1)$ time, plus the time of the recursive call. The running time of binary search is therefore $O(\log n)$. That compares favorably with the linear search, which takes $O(n)$ time on the average, as we have seen.

EXERCISES

6.5.1: Write functions to (a) insert x and (b) delete x from a list L, using linear search of an array.

6.5.2: Repeat Exercise 6.5.1 for an array with sentinels.

6.5.3: Repeat Exercise 6.5.1 for a sorted array.

6.5.4: Write the following functions assuming that list elements are of some arbitrary type `ETYPE`, for which we have functions $eq(x, y)$ that tells whether x and y are equal and $lt(x, y)$ telling whether x precedes y in the order of elements of type `ETYPE`.

a) Function *lookup* of Fig. 6.11.

b) Function *lookup* of Fig. 6.12.

c) Function *binsearch* of Fig. 6.14.

Probes in binary search

6.5.5**: Let $P(k)$ be the length $(high - low + 1)$ of the longest array such that the binary search algorithm of Fig. 6.14 never makes more than k *probes* [evaluations of `mid` at line (3)]. For example, $P(1) = 1$, and $P(2) = 3$. Write a recurrence relation for $P(k)$. What is the solution to your recurrence relation? Does it demonstrate that binary search makes $O(\log n)$ probes?

6.5.6*: Prove by induction on the difference between *low* and *high* that if x is in the range `A[low..high]`, then the binary search algorithm of Fig. 6.14 will find x.

6.5.7: Suppose we allowed arrays to have duplicates, so insertion could be done in $O(1)$ time. Write *insert*, *delete*, and *lookup* functions for this data structure.

6.5.8: Rewrite the binary search program to use iteration rather than recursion.

6.5.9**: Set up and solve a recurrence relation for the running time of binary search on an array of n elements. *Hint*: To simplify, it helps to take $T(n)$ as an upper bound on the running time of binary search on any array of n or fewer elements (rather than on exactly n elements, as would be our usual approach).

Ternary search

6.5.10: In *ternary search*, given a range *low* to *high*, we compute the approximate 1/3 point of the range,

$$first = \lfloor (2 \times low + high)/3 \rfloor$$

and compare the lookup element x with $A[first]$. If $x > A[first]$, we compute the approximate 2/3 point,

$$second = \lceil (low + 2 \times high)/3 \rceil$$

and compare x with $A[second]$. Thus we isolate x to within one of three ranges, each no more than one third the range *low* to *high*. Write a function to perform ternary search.

6.5.11**: Repeat Exercise 6.5.5 for ternary search. That is, find and solve a recurrence relation for the largest array that requires no more than k probes during ternary search. How do the number of probes required for binary and ternary search compare? That is, for a given k, can we handle larger arrays by binary search or by ternary search?

6.6 Stacks

Top of stack

A *stack* is an abstract data type based on the list data model in which all operations are performed at one end of the list, which is called the *top* of the stack. The term "LIFO (for last-in first-out) list" is a synonym for stack.

The abstract model of a stack is the same as that of a list — that is, a sequence of elements $a_1, a_2, \ldots, a_n$ of some one type. What distinguishes stacks from general lists is the particular set of operations permitted. We shall give a more complete set of operations later, but for the moment, we note that the quintessential stack operations are *push* and *pop*, where *push*(x) puts the element x on top of the stack and *pop* removes the topmost element from the stack. If we write stacks with the top at the right end, the operation *push*(x) applied to the list $(a_1, a_2, \ldots, a_n)$ yields the list $(a_1, a_2, \ldots, a_n, x)$. Popping the list $(a_1, a_2, \ldots, a_n)$ yields the list $(a_1, a_2, \ldots, a_{n-1})$; popping the empty list, ϵ, is impossible and causes an error condition,

Push and pop

✦ **Example 6.9.** Many compilers begin by turning the infix expressions that appear in programs into equivalent postfix expressions. For example, the expression $(3+4) \times 2$ is $3\ 4\ +\ 2\ \times$ in postfix notation. A stack can be used to evaluate postfix expressions. Starting with an empty stack, we scan the postfix expression from left to right. Each time we encounter an argument, we push it onto the stack. When we encounter an operator, we pop the stack twice, remembering the operands popped. We then apply the operator to the two popped values (with the second as the left operand) and push the result onto the stack. Figure 6.15 shows the stack after each step in the processing of the postfix expression $3\ 4\ +\ 2\ \times$. The result, 14, remains on the stack after processing. ✦

SYMBOL PROCESSED	STACK	ACTIONS
initial	ϵ	
3	3	*push* 3
4	3, 4	*push* 4
+	ϵ	*pop* 4; *pop* 3
		compute $7 = 3 + 4$
	7	*push* 7
2	7, 2	*push* 2
×	ϵ	*pop* 2; *pop* 7
		compute $14 = 7 \times 2$
	14	*push* 14

Fig. 6.15. Evaluating a postfix expression using a stack.

Operations on a Stack

The ADT stack

The two previous ADT's we discussed, the dictionary and the priority queue, each had a definite set of associated operations. The stack is really a family of similar ADT's with the same underlying model, but with some variation in the set of allowable operations. In this section, we shall discuss the common operations on stacks and show two data structures that can serve as implementations for the stack, one based on linked lists and the other on arrays.

In any collection of stack operations we expect to see *push* and *pop*, as we mentioned. There is another common thread to the operations chosen for the stack ADT(s): they can all be implemented simply in $O(1)$ time, independent of the number of elements on the stack. You can check as an exercise that for the two data structures suggested, all operations require only constant time.

Clear stack

In addition to *push* and *pop*, we generally need an operation *clear* that initializes the stack to be empty. In Example 6.9, we tacitly assumed that the stack started out empty, without explaining how it got that way. Another possible operation is a test to determine whether the stack is currently empty.

Full and empty stacks

The last of the operations we shall consider is a test whether the stack is "full." Now in our abstract model of a stack, there is no notion of a full stack, since a stack is a list and lists can grow as long as we like, in principle. However, in any implementation of a stack, there will be some length beyond which it cannot grow. The most common example is when we represent a list or stack by an array. As

seen in the previous section, we had to assume the list would not grow beyond the constant **MAX**, or our implementation of *insert* would not work.

The formal definitions of the operations we shall use in our implementation of stacks are the following. Let S be a stack of type **ETYPE** and x an element of type **ETYPE**.

1. *clear*(S). Make the stack S empty.
2. *isEmpty*(S). Return **TRUE** if S is empty, **FALSE** otherwise.
3. *isFull*(S). Return **TRUE** if S is full, **FALSE** otherwise.
4. *pop*(S, x). If S is empty, return **FALSE**; otherwise, set x to the value of the top element on stack S, remove this element from S, and return **TRUE**.
5. *push*(x, S). If S is full, return **FALSE**; otherwise, add the element x to the top of S and return **TRUE**.

There is a common variation of *pop* that assumes S is nonempty. It takes only S as an argument and returns the element x that is popped. Yet another alternative version of *pop* does not return a value at all; it just removes the element at the top of the stack. Similarly, we may write *push* with the assumption that S is not "full." In that case, *push* does not return any value.

Array Implementation of Stacks

The implementations we used for lists can also be used for stacks. We shall discuss an array-based implementation first, followed by a linked-list representation. In each case, we take the type of elements to be **int**. Generalizations are left as exercises.

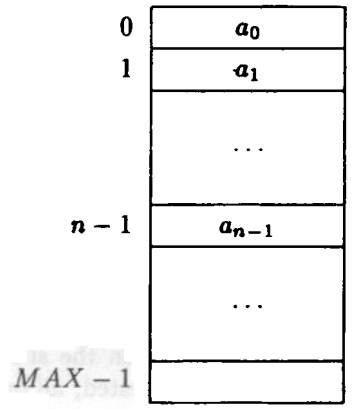


Fig. 6.16. An array representing a stack.

The declaration for an array-based stack of integers is

```
typedef struct {
    int A[MAX]
    int top;
} STACK
```

```
void clear(STACK *pS)
{
    pS->top = -1;
}

BOOLEAN isEmpty(STACK *pS)
{
    return (pS->top < 0);
}

BOOLEAN isFull(STACK *pS)
{
    return (pS->top >= MAX-1);
}

BOOLEAN pop(STACK *pS, int *px)
{
    if (isEmpty(pS))
        return FALSE;
    else {
        (*px) = pS->A[(pS->top)--];
        return TRUE;
    }
}

BOOLEAN push(int x, STACK *pS)
{
    if (isFull(pS))
        return FALSE;
    else {
        pS->A[++(pS->top)] = x;
        return TRUE;
    }
}
```

Fig. 6.17. Functions to implement stack operations on arrays.

With an array-based implementation, the stack can grow either upward (from lower locations to higher) or downward (from higher locations to lower). We choose to have the stack grow upward;[5] that is, the oldest element a_0 in the stack is in location 0, the next-to-oldest element a_1 is in location 1, and the most recently inserted element a_{n-1} is in the location $n-1$.

The field `top` in the array structure indicates the position of the top of stack. Thus, in Fig. 6.16, `top` has the value $n-1$. An empty stack is represented by having $top = -1$. In that case, the content of array `A` is irrelevant, there being no elements on the stack.

The programs for the five stack operations defined earlier in this section are

[5] Thus the "top" of the stack is physically shown at the bottom of the page, an unfortunate but quite standard convention.

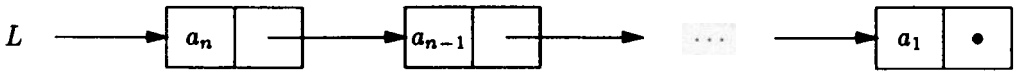
```
void clear(STACK *pS)
{
    (*pS) = NULL;
}

BOOLEAN isEmpty(STACK *pS)
{
    return ((*pS) == NULL);
}

BOOLEAN isFull(STACK *pS)
{
    return FALSE;
}

BOOLEAN pop(STACK *pS, int *px)
{
    if ((*pS) == NULL)
        return FALSE;
    else {
        (*px) = (*pS)->element;
        (*pS) = (*pS)->next;
        return TRUE;
    }
}

BOOLEAN push(int x, STACK *pS)
{
    STACK newCell;

    newCell = (STACK) malloc(sizeof(struct CELL));
    newCell->element = x;
    newCell->next = (*pS);
    (*pS) = newCell;
    return TRUE;
}
```

Fig. 6.18. Functions to implement stacks by linked lists.

shown in Fig. 6.17. We pass stacks by reference to avoid having to copy large arrays that are arguments of the functions.

Linked-List Implementation of a Stack

We can represent a stack by a linked-list data structure, like any list. However, it is convenient if the top of the stack is the front of the list. That way, we can push and pop at the head of the list, which takes only $O(1)$ time. If we had to find the end of the list to push or pop, it would take $O(n)$ time to do these operations on a stack of length n. However, as a consequence, the stack $S = (a_1, a_2, \ldots, a_n)$ must be represented "backward" by the linked list, as:



The type definition macro we have used for list cells can as well be used for stacks. The macro

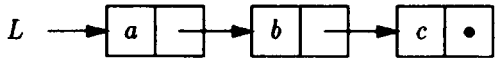
```
DefCell(int, CELL, STACK);
```

defines stacks of integers, expanding into

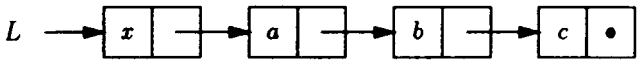
```
typdef struct CELL *STACK;
struct CELL {
    int element;
    STACK next;
};
```

With this representation, the five operations can be implemented by the functions in Fig. 6.18. We assume that `malloc` never runs out of space, which means that the *isFull* operation always returns **FALSE**, and the *push* operation never fails.

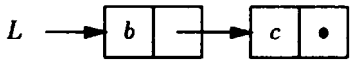
The effects of *push* and *pop* on a stack implemented as a linked list are illustrated in Fig. 6.19.



(a) List L.



(b) After executing $push(x, L)$.



(c) After executing $pop(L, x)$ on list L of (a).

Fig. 6.19. Push and pop operations on a stack implemented as a linked list.

EXERCISES

6.6.1: Show the stack that remains after executing the following sequence of operations, starting with an empty stack: *push*(*a*), *push*(*b*), *pop*, *push*(*c*), *push*(*d*), *pop*, *push*(*e*), *pop*, *pop*.

6.6.2: Using only the five operations on stacks discussed in this section to manipulate the stack, write a C program to evaluate postfix expressions with integer operands and the four usual arithmetic operators, following the algorithm suggested in Example 6.9. Show that you can use either the array or the linked-list implementation with your program by defining the data type **STACK** appropriately and including with your program first the functions of Fig. 6.17, and then the functions of Fig. 6.18.

6.6.3*: How would you use a stack to evaluate prefix expressions?

6.6.4: Compute the running time of each of the functions in Figs. 6.17 and 6.18. Are they all $O(1)$?

6.6.5: Sometimes, a stack ADT uses an operation *top*, where $top(S)$ returns the top element of stack S, which must be assumed nonempty. Write functions for *top* that can be used with

a) The array data structure
b) The linked-list data structure

that we defined for stacks in this section. Do your implementations of *top* all take $O(1)$ time?

6.6.6: Simulate a stack evaluating the following postfix expressions.

a) $ab + cd \times + e \times$
b) $abcde + + + +$
c) $ab + c + d + e +$

6.6.7*: Suppose we start with an empty stack and perform some push and pop operations. If the stack after these operations is $(a_1, a_2, \ldots, a_n)$ (top at the right), prove that a_i was pushed before a_{i+1} was pushed, for $i = 1, 2, \ldots, n-1$.

6.7 Implementing Function Calls Using a Stack

An important application of stacks is normally hidden from view: a stack is used to allocate space in the computer's memory to the variables belonging to the various functions of a program. We shall discuss the mechanism used in C, although a similar mechanism is used in almost every other programming language as well.

```
            int fact(int n)
            {
(1)             if (n <= 1)
(2)                 return 1; /* basis */
                else
(3)                 return n*fact(n-1); /* induction */
            }
```

Fig. 6.20. Recursive function to compute $n!$.

To see what the problem is, consider the simple, recursive factorial function `fact` from Section 2.7, which we reproduce here as Fig. 6.20. The function has a parameter `n` and a return value. As `fact` calls itself recursively, different calls are active at the same time. These calls have different values of the parameter `n` and produce different return values. Where are these different objects with the same names kept?

Run-time organization

To answer the question, we must learn a little about the *run-time organization* associated with a programming language. The run-time organization is the plan used to subdivide the computer's memory into regions to hold the various data items

Activation record

used by a program. When a program is run, each execution of a function is called an *activation*. The data objects associated with each activation are stored in the memory of the computer in a block called an *activation record* for that activation. The data objects include parameters, the return value, the return address, and any variables local to the function.

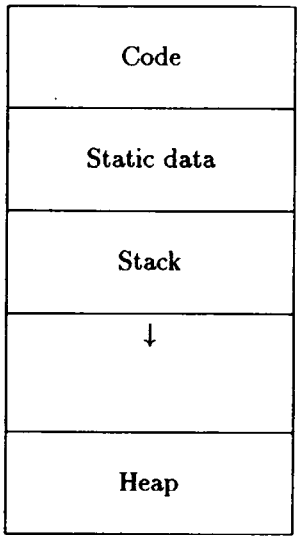


Fig. 6.21. Typical run-time memory organization.

Static data
Run-time stack

Figure 6.21 shows a typical subdivision of run-time memory. The first area contains the object code for the program being executed. The next area contains the static data for the program, such as the values of certain constants and external variables used by the program. The third area is the *run-time stack*, which grows downward toward the higher addresses in memory. At the highest-numbered memory locations is the *heap*, an area set aside for the objects that are dynamically allocated using `malloc`.[6]

The run-time stack holds the activation records for all the currently live activations. A stack is the appropriate structure, because when we call a function, we can push an activation record onto the stack. At all times, the currently executing activation A_1 has its activation record at the top of the stack. Just below the top of the stack is the activation record for the activation A_2 that called A_1. Below A_2's activation record is the record for the activation that called A_2, and so on. When a function returns, we pop its activation record off the top of stack, exposing the activation record of the function that called it. That is exactly the right thing to do, because when a function returns, control passes to the calling function.

✦ **Example 6.10.** Consider the skeletal program shown in Fig. 6.22. This program is nonrecursive, and there is never more than one activation for any one function.

[6] Do not confuse this use of the term "heap" with the heap data structure discussed in Section 5.9.

```
void P();
void Q();

main() {
    int x, y, z;

    P(); /* Here */
}

void P();
{
    int p1, p2;

    Q();
}

void Q()
{
    int q1, q2, q3;

}
```

Fig. 6.22. Skeletal program.

When the main function starts to execute, its activation record containing the space for the variables `x`, `y`, and `z` is pushed onto the stack. When function `P` is called, at the place marked `Here`, its activation record, which contains the space for the variables `p1` and `p2`, is pushed onto the stack.[7] When `P` calls `Q`, `Q`'s activation record is pushed onto the stack. At this point, the stack is as shown in Fig. 6.23.

When `Q` finishes executing, its activation record is popped off the stack. At that time, `P` is also finished, and so its activation record is popped. Finally, `main` too is finished and has its activation record popped off the stack. Now the stack is empty, and the program is finished. ✦

✦ **Example 6.11.** Consider the recursive function `fact` from Fig. 6.20. There may be many activations of `fact` live at any one time, but each one will have an activation record of the same form, namely

consisting of a word for the parameter `n`, which is filled initially, and a word for the return value, which we have denoted `fact`. The return value is not filled until the last step of the activation, just before the return.

[7] Notice that the activation record for P has two data objects, and so is of a "type" different from that of the activation record for the main program. However, we may regard all activation record forms for a program as variants of a single record type, thus preserving the viewpoint that a stack has all its elements of the same type.

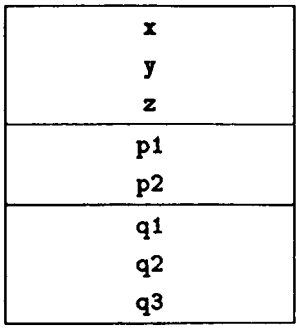
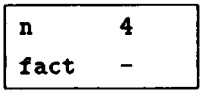
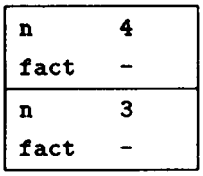


Fig. 6.23. Run-time stack when function `Q` is executing.

Suppose we call `fact(4)`. Then we create one activation record, of the form



As `fact(4)` calls `fact(3)`, we next push an activation record for that activation onto the run-time stack, which now appears as



Note that there are two locations named `n` and two named `fact`. There is no confusion, since they belong to different activations and only one activation record can be at the top of the stack at any one time: the activation record belonging to the currently executing activation.

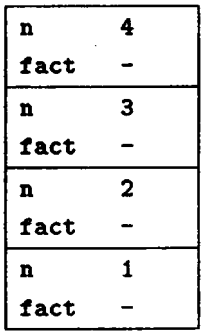


Fig. 6.24. Activation records during execution of `fact`.

Then `fact(3)` calls `fact(2)`, which calls `fact(1)`. At that point, the run-time stack is as in Fig. 6.24. Now `fact(1)` makes no recursive call, but assigns $fact = 1$. The value 1 is thus placed in the slot of the top activation record reserved for `fact`.

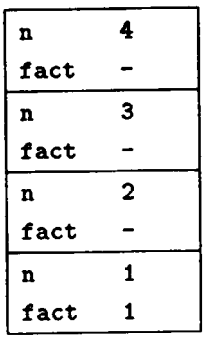


Fig. 6.25. After `fact(1)` computes its value.

The other slots labeled `fact` are unaffected, as shown in Fig. 6.25.

Then, `fact(1)` returns, exposing the activation record for `fact(2)` and returning control to the activation `fact(2)` at the point where `fact(1)` was called. The return value, 1, from `fact(1)` is multiplied by the value of n in the activation record for `fact(2)`, and the product is placed in the slot for `fact` of that activation record, as required by line (3) in Fig. 6.20. The resulting stack is shown in Fig. 6.26.

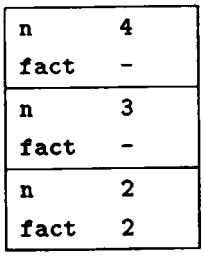
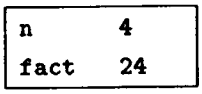


Fig. 6.26. After `fact(2)` computes its value.

Similarly, `fact(2)` then returns control to `fact(3)`, and the activation record for `fact(2)` is popped off the stack. The return value, 2, multiplies n of `fact(3)`, producing the return value 6. Then, `fact(3)` returns, and its return value multiplies n in `fact(4)`, producing the return value 24. The stack is now



At this point, `fact(4)` returns to some hypothetical calling function whose activation record (not shown) is below that of `fact(4)` on the stack. However, it would receive the return value 24 as the value of `fact(4)`, and would proceed with its own execution. ✦

EXERCISES

6.7.1: Consider the C program of Fig. 6.27. The activation record for `main` has a slot for the integer `i`. The important data in the activation record for `sum` is

```
#define MAX 4
int A[MAX];
int sum(int i);

main()
{
    int i;

(1)     for (i = 0; i < MAX; i++)
(2)         scanf("%d", &A[i]);
(3)     printf("\n"  sum(0));
}

int sum(int i)
{
(4)         if (i > MAX)
(5)             return 0;
            else
(6)             return A[i] + sum(i+1);
}
```

Fig. 6.27. Program for Exercise 6.7.1.

1. The parameter `i`.
2. The return value.
3. An unnamed temporary location, which we shall call `temp`, to store the value of `sum(i+1)`. The latter is computed in line (6) and then added to $A[i]$ to form the return value.

Show the stack of activation records immediately before and immediately after each call to `sum`, on the assumption that the value of $A[i]$ is $10i$. That is, show the stack immediately after we have pushed an activation record for `sum`, and just before we pop an activation record off the stack. You need not show the contents of the bottom activation record (for `main`) each time.

```
void delete(int x, LIST *pL)
{
    if ((*pL) != NULL)
        if (x == (*pL)->element)
            (*pL) = (*pL)->next;
        else
            delete(x, &((*pL)->next));
end
```

Fig. 6.28. Function for Exercise 6.7.2.

6.7.2*: The function **delete** of Fig. 6.28 removes the first occurrence of integer x from a linked list composed of the usual cells defined by

```
DefCell(int, CELL, LIST);
```

The activation record for **delete** consists of the parameters **x** and **pL**. However, since **pL** is a pointer to a list, the value of the second parameter in the activation record is not a pointer to the first cell on the list, but rather a pointer to a pointer to the first cell. Typically, an activation record will hold a pointer to the **next** field of some cell. Show the sequence of stacks when **delete(3,&L)** is called (from some other function) and L is a pointer to the first cell of a linked list containing elements 1, 2, 3, and 4, in that order.

6.8 Queues

Front and rear of queue

Another important ADT based on the list data model is the *queue*, a restricted form of list in which elements are inserted at one end, the *rear*, and removed from the other end, the *front*. The term "FIFO (first-in first-out) list" is a synonym for queue.

The intuitive idea behind a queue is a line at a cashier's window. People enter the line at the rear and receive service once they reach the front. Unlike a stack, there is fairness to a queue; people are served in the order in which they enter the line. Thus the person who has waited the longest is the one who is served next.

Operations on a Queue

Enqueue and dequeue

The abstract model of a queue is the same as that of a list (or a stack), but the operations applied are special. The two operations that are characteristic of a queue are *enqueue* and *dequeue*; *enqueue*(x) adds x to the rear of a queue, *dequeue* removes the element from the front of the queue. As is true of stacks, there are certain other useful operations that we may want to apply to queues.

Let Q be a queue whose elements are of type **ETYPE**, and let x be an element of type **ETYPE**. We shall consider the following operations on queues:

1. *clear*(Q). Make the queue Q empty.
2. *dequeue*(Q, x). If Q is empty, return **FALSE**; otherwise, set x to the value of the element at the front of Q, remove this element from Q, and return **TRUE**.
3. *enqueue*(x, Q). If Q is full, return **FALSE**; otherwise, add the element x to the rear of Q and return **TRUE**.
4. *isEmpty*(Q). Return **TRUE** if Q is empty and **FALSE** otherwise.
5. *isFull*(Q). Return **TRUE** if Q is full and **FALSE** otherwise.

As with stacks, we can have more "trusting" versions of *enqueue* and *dequeue* that do not check for a full or empty queue, respectively. Then *enqueue* does not return a value, and *dequeue* takes only Q as an argument and returns the value dequeued.

A Linked-List Implementation of Queues

A useful data structure for queues is based on linked lists. We start with the usual definition of cells given by the macro

```
void clear(QUEUE *pQ)
{
    pQ->front = NULL
}

BOOLEAN isEmpty(QUEUE *pQ)
{
    return (pQ->front == NULL);
}

BOOLEAN isFull(QUEUE *pQ)
{
    return FALSE;
}

BOOLEAN dequeue(QUEUE *pQ, int *px)
{
    if (isEmpty(pQ))
        return FALSE;
    else {
        (*px) = pQ->front->element;
        pQ->front = pQ->front->next;
        return TRUE;
    }
}

BOOLEAN enqueue(int x, QUEUE *pQ)
{
    if (isEmpty(pQ)) {
        pQ->front = (LIST) malloc(sizeof(struct CELL));
        pQ->rear = pQ->front;
    }
    else {
        pQ->rear->next = (LIST) malloc(sizeof(struct CELL));
        pQ->rear = pQ->rear->next;
    }
    pQ->rear->element = x;
    pQ->rear->next = NULL;
    return TRUE;
}
```

Fig. 6.29. Procedures to implement linked-list queue operations.

```
DefCell(int, CELL, LIST);
```

As previously in this chapter, we assume that elements of our queues are integers and invite the reader to generalize our functions to arbitrary element types.

The elements of a queue will be stored on a linked list of cells. A queue itself is a structure with two pointers — one to the front cell (the first on the linked list) and another to the rear cell (the last on the linked list). That is, we define

More Abstract Data Types

We can add the stack and queue to the table of ADT's that we started in Section 5.9. We covered two data structures for the stack in Section 6.6, and one data structure for the queue in Section 6.8. Exercise 6.8.3 covers another data structure, the "circular array," for the queue.

ADT	Stack	Queue
Abstract Implementation	List	List
Data Structures	1) Linked List 2) Array	1) Linked List 2) Circular Array

```
typedef struct {
    LIST front, rear;
} QUEUE;
```

If the queue is empty, `front` will be `NULL`, and the value of `rear` is then irrelevant.

Figure 6.29 gives programs for the queue operations mentioned in this section. Note that when a linked list is used there is no notion of a "full" queue, and so *isFull* returns `FALSE` always. However, if we used some sort of array-based implementation of a queue, there would be the possibility of a full queue.

EXERCISES

6.8.1: Show the queue that remains after executing the following sequence of operations, starting with an empty queue: *enqueue*(a), *enqueue*(b), *dequeue*, *enqueue*(c), *enqueue*(d), *dequeue*, *enqueue*(e), *dequeue*, *dequeue*.

6.8.2: Show that each of the functions in Fig. 6.29 can be executed in $O(1)$ time, regardless of the length of the queue.

Circular array

6.8.3*: We can represent a queue by an array, provided that the queue does not grow too long. In order to make the operations take $O(1)$ time, we must think of the array as *circular*. That is, the array `A[0..n-1]` can be thought of as having `A[1]` follow `A[0]`, `A[2]` follow `A[1]`, and so on, up to `A[n-1]` following `A[n-2]`, but also `A[0]` follows `A[n-1]`. The queue is represented by a pair of integers `front` and `rear` that indicate the positions of the front and rear elements of the queue. An empty queue is represented by having *front* be the position that follows *rear* in the circular sense; for example, $front = 23$ and $rear = 22$, or $front = 0$ and $rear = n - 1$. Note that therefore, the queue cannot have n elements, or that condition too would be represented by *front* immediately following *rear*. Thus the queue is full when it has $n-1$ elements, not when it has n elements. Write functions for the queue operations assuming the circular array data structure. Do not forget to check for full and empty queues.

6.8.4*: Show that if $(a_1, a_2, \ldots, a_n)$ is a queue with a_1 at the front, then a_i was enqueued before a_{i+1}, for $i = 1, 2, \ldots, n-1$.

6.9 Longest Common Subsequences

This section is devoted to an interesting problem about lists. Suppose we have two lists and we want to know what differences there are between them. This problem appears in many different guises; perhaps the most common occurs when the two lists represent two different versions of a text file and we want to determine which lines are common to the two versions. For notational convenience, throughout this section we shall assume that lists are character strings.

A useful way to think about this problem is to treat the two files as sequences of symbols, $x = a_1 \cdots a_m$ and $y = b_1 \cdots b_n$, where a_i represents the ith line of the first file and b_j represents the jth line of the second file. Thus an abstract symbol like a_i may really be a "big" object, perhaps a full sentence.

Diff command

There is a UNIX command `diff` that compares two text files for their differences. One file, x, might be the current version of a program and the other, y, might be the version of the program before a small change was made. We could use `diff` to remind ourselves of the changes that were made turning y into x. The typical changes that are made to a text file are

1. Inserting a line.
2. Deleting a line.

A modification of a line can be treated as a deletion followed by an insertion.

Usually, if we examine two text files in which a small number of such changes have been made when transforming one into the other, it is easy to see which lines correspond to which, and which lines have been deleted and which inserted. The `diff` command makes the assumption that one can identify what the changes are by first finding a *longest common subsequence*, or *LCS*, of the two lists whose elements are the lines of the two text files involved. An LCS represents those lines that have not been changed.

LCS

Recall that a subsequence is formed from a list by deleting zero or more elements, keeping the remaining elements in order. A *common subsequence* of two lists is a list that is a subsequence of both. A longest common subsequence of two lists is a common subsequence that is as long as any common subsequence of the two lists.

Common subsequence

✦ **Example 6.12.** In what follows, we can think of characters like `a`, `b`, or `c`, as standing for lines of a text file, or as any other type of elements if we wish. As an example, `baba` and `cbba` are both longest common subsequences of `abcabba` and `cbabac`. We see that `baba` is a subsequence of `abcabba`, because we may take positions 2, 4, 5, and 7 of the latter string to form `baba`. String `baba` is also a subsequence of `cbabac`, because we may take positions 2, 3, 4, and 5. Similarly, `cbba` is formed from positions 3, 5, 6, and 7 of `abcabba` and from positions 1, 2, 4, and 5 of `cbabac`. Thus `cbba` too is a common subsequence of these strings. We must convince ourselves that these are longest common subsequences; that is, there

are no common subsequences of length five or more. That fact will follow from the algorithm we describe next. ✦

A Recursion That Computes the LCS

We offer a recursive definition of the length of the LCS of two lists. This definition will let us calculate the length easily, and by examining the table it constructs, we can then discover one of the possible LCS's itself, rather than just its length. From the LCS, we can deduce what changes were made to the text files in question; essentially, everything that is not part of the LCS is a change.

To find the length of an LCS of lists x and y, we need to find the lengths of the LCS's of all pairs of prefixes, one from x and the other from y. Recall that a prefix is an initial sublist of a list, so that, for instance, the prefixes of `cbabac` are ϵ, `c`, `cb`, `cba`, and so on. Suppose that $x = (a_1, a_2, \ldots, a_m)$ and $y = (b_1, b_2, \ldots, b_n)$. For each i and j, where i is between 0 and m and j is between 0 and n, we can ask for an LCS of the prefix $(a_1, \ldots, a_i)$ from x and the prefix $(b_1, \ldots, b_j)$ from y.

If either i or j is 0, then one of the prefixes is ϵ, and the only possible common subsequence of the two prefixes is ϵ. Thus when either i or j is 0, the length of the LCS is 0. This observation is formalized in both the basis and rule (1) of the induction that follows our informal discussion of how the LCS is computed.

Now consider the case where both i and j are greater than 0. It helps to think of an LCS as a matching between certain positions of the two strings involved. That is, for each element of the LCS, we match the two positions of the two strings from which that element comes. Matched positions must have the same symbols, and the lines between matched positions must not cross.

✦ **Example 6.13.** Figure 6.30(a) shows one of two possible matchings between strings `abcabba` and `cbabac` corresponding to the common subsequence `baba` and Fig. 6.30(b) shows a matching corresponding to `cbba`. ✦

(a) For `baba`. (b) For `cbba`.

Fig. 6.30. LCS's as matchings between positions.

Thus let us consider any matching between prefixes $(a_1, \ldots, a_i)$ and $(b_1, \ldots, b_j)$. There are two cases, depending on whether or not the last symbols of the two lists are equal.

a) If $a_i \neq b_j$, then the matching cannot include both a_i and b_j. Thus an LCS of $(a_1, \ldots, a_i)$ and $(b_1, \ldots, b_j)$ must be either

   i) An LCS of $(a_1, \ldots, a_{i-1})$ and $(b_1, \ldots, b_j)$, or

   ii) An LCS of $(a_1, \ldots, a_i)$ and $(b_1, \ldots, b_{j-1})$.

If we have already found the lengths of the LCS's of these two pairs of prefixes, then we can take the larger to be the length of the LCS of $(a_1, \ldots, a_i)$ and $(b_1, \ldots, b_j)$. This situation is formalized in rule (2) of the induction that follows.

b) If $a_i = b_j$, we can match a_i and b_j, and the matching will not interfere with any other potential matches. Thus the length of the LCS of $(a_1, \ldots, a_i)$ and $(b_1, \ldots, b_j)$ is 1 greater than the length of the LCS of $(a_1, \ldots, a_{i-1})$ and $(b_1, \ldots, b_{j-1})$. This situation is formalized in rule (3) of the following induction.

These observations let us give a recursive definition for $L(i, j)$, the length of the LCS of $(a_1, \ldots, a_i)$ and $(b_1, \ldots, b_j)$. We use complete induction on the sum $i + j$.

BASIS. If $i + j = 0$, then both i and j are 0, and so the LCS is ϵ. Thus $L(0, 0) = 0$.

INDUCTION. Consider i and j, and suppose we have already computed $L(g, h)$ for any g and h such that $g + h < i + j$. There are three cases to consider.

1. If either i or j is 0, then $L(i, j) = 0$.
2. If $i > 0$ and $j > 0$, and $a_i \neq b_j$, then $L(i, j) = \max\big(L(i, j-1), L(i-1, j)\big)$.
3. If $i > 0$ and $j > 0$, and $a_i = b_j$, then $L(i, j) = 1 + L(i-1, j-1)$.

A Dynamic Programming Algorithm for the LCS

Ultimately what we want is $L(m, n)$, the length of an LCS for the two lists x and y. If we write a recursive program based on the preceding induction, it will take time that is exponential in the smaller of m and n. That is far too much time to make the simple recursive algorithm practical for, say, $n = m = 100$. The reason this recursion does so badly is a bit subtle. To begin, suppose there are no matches at all between characters in the lists x and y, and we call $L(3, 3)$. That results in calls to $L(2, 3)$ and $L(3, 2)$. But each of these calls results in a call to $L(2, 2)$. We thus do the work of $L(2, 2)$ twice. The number of times $L(i, j)$ is called increases rapidly as the arguments of L become smaller. If we continue the trace of calls, we find that $L(1, 1)$ is called 6 times, $L(0, 1)$ and $L(1, 0)$ are called 10 times each, and $L(0, 0)$ is called 20 times.

We can do much better if we build a two-dimensional table, or array, to store $L(i, j)$ for the various values of i and j. If we compute the values in order of the induction — that is, smallest values of $i + j$ first — then the needed values of L are always in the table when we compute $L(i, j)$. In fact, it is easier to compute L by rows, that is, for $i = 0, 1, 2$, and so on; within a row, compute by columns, for $j = 0, 1, 2$, and so on. Again, we can be sure of finding the needed values in the table when we compute $L(i, j)$, and no recursive calls are necessary. As a result, it takes only $O(1)$ time to compute each entry of the table, and a table for the LCS of lists of length m and n can be constructed in $O(mn)$ time.

In Fig. 6.31 we see C code that fills this table, working by row rather than by the sum $i + j$. We assume that the list x is stored in an array `a[1..m]` and y is stored in `b[1..n]`. Note that the 0th elements of these are unused; doing so simplifies the notation in Fig. 6.31. We leave it as an exercise to show that the running time of this program is $O(mn)$ on lists of length m and n.[8]

[8] Strictly speaking, we discussed only big-oh expressions that are a function of one variable. However, the meaning here should be clear. If $T(m, n)$ is the running time of the program

```
for (j = 0; j <= n; j++)
    L[0][j] = 0;
for (i = 1; i <= m; i++) {
    L[i][0] = 0;
    for (j = 1; j <= n; j++)
        if (a[i] != b[j])
            if (L[i-1][j] >= L[i][j-1])
                L[i][j] = L[i-1][j];
            else
                L[i][j] = L[i][j-1];
        else /* a[i] == b[j] */
            L[i][j] = 1 + L[i-1][j-1];
}
```

Fig. 6.31. C fragment to fill the LCS table.

Dynamic Programming

The term "dynamic programming" comes from a general theory developed by R. E. Bellman in the 1950's for solving problems in control systems. People who work in the field of artificial intelligence often speak of the technique under the name *memoing* or *tabulation*.

Memoing

A table-filling technique like this example is often called a *dynamic programming algorithm*. As in this case, it can be much more efficient than a straightforward implementation of a recursion that solves the same subproblem repeatedly.

Dynamic programming algorithm

✦ **Example 6.14.** Let x be the list cbabac and y the list abcabba. Figure 6.32 shows the table constructed for these two lists. For instance, $L(6,7)$ is a case where $a_6 \neq b_7$. Thus $L(6,7)$ is the larger of the entries just below and just to the left. Since these are 4 and 3, respectively, we set $L(6,7)$, the entry in the upper right corner, to 4. Now consider $L(4,5)$. Since both a_4 and b_5 are the symbol b, we add 1 to the entry $L(3,4)$ that we find to the lower left. Since that entry is 2, we set $L(4,5)$ to 3. ✦

Recovery of an LCS

We now have a table giving us the length of the LCS, not only for the lists in question, but for each pair of their prefixes. From this information we must deduce one of the possible LCS's for the two lists in question. To do so, we shall find the matching pairs of elements that form one of the LCS's. We shall find a path through the table, beginning at the upper right corner; this path will identify an LCS.

Suppose that our path, starting at the upper right corner, has taken us to row i and column j, the point in the table that corresponds to the pair of elements a_i

on lists of length m and n, then there are constants m_0, n_0, and c such that for all $m \geq m_0$ and $n \geq n_0$, $T(m,n) \leq cmn$.

c	6	0	1	2	3	3	3	3	4
a	5	0	1	2	2	3	3	3	4
b	4	0	1	2	2	2	3	3	3
a	3	0	1	1	1	2	2	2	3
b	2	0	0	1	1	1	2	2	2
c	1	0	0	0	1	1	1	1	1
	0	0	0	0	0	0	0	0	0
		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a

Fig. 6.32. Table of longest common subsequences for cbabac and abcabba.

and b_j. If $a_i = b_j$, then $L(i, j)$ was chosen to be $1 + L(i-1, j-1)$. We thus treat a_i and b_j as a matched pair of elements, and we shall include the symbol that is a_i (and also b_j) in the LCS, ahead of all the elements of the LCS found so far. We then move our path down and to the left, that is, to row $i-1$ and column $j-1$.

However, it is also possible that $a_i \neq b_j$. If so, then $L(i, j)$ must equal at least one of $L(i-1, j)$ and $L(i, j-1)$. If $L(i, j) = L(i-1, j)$, we shall move our path one row down, and if not, we know that $L(i, j) = L(i, j-1)$, and we shall move our path one column left.

When we follow this rule, we eventually arrive at the lower left corner. At that point, we have selected a certain sequence of elements for our LCS, and the LCS itself is the list of these elements, in the reverse of the order in which they were selected.

c	6	0	1	2	3	3	3	3	**4**
a	5	0	1	2	2	3	3	3	**4**
b	4	0	1	2	2	2	3	**3**	3
a	3	0	1	1	1	2	**2**	2	3
b	2	0	0	1	1	1	**2**	2	2
c	1	0	0	0	**1**	**1**	1	1	1
	0	**0**	**0**	**0**	0	0	0	0	0
		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a

Fig. 6.33. A path that finds the LCS caba.

✦ **Example 6.15.** The table of Fig. 6.32 is shown again in Fig. 6.33, with a path shown in bold. We start with $L(6, 7)$, which is 4. Since $a_6 \neq b_7$, we look immediately to the left and down to find the value 4, which must appear in at least one of these places. In this case, 4 appears only below, and so we go to $L(5, 7)$. Now $a_5 = b_7$; both are a. Thus a is the last symbol of the LCS, and we move southwest, to $L(4, 6)$.

Since a_4 and b_6 are both b, we include b, ahead of a, in the LCS being formed, and we again move southwest, to $L(3,5)$. Here, we find $a_3 \neq b_5$, but $L(3,5)$, which is 2, equals both the entry below and the entry to the left. We have elected in this situation to move down, so we next move to $L(2,5)$. There we find $a_2 = b_5 =$ b, and so we put a b ahead of the LCS being formed and move southwest to $L(1,4)$.

Since $a_1 \neq b_4$ and only the entry to the left has the same value (1) as $L(1,4)$, we move to $L(1,3)$. Now we have $a_1 = b_3 =$ c, and so we add c to the beginning of the LCS and move to $L(0,2)$. At this point, we have no choice but to move left to $L(0,1)$ and then $L(0,0)$, and we are done. The resulting LCS consists of the four characters we discovered, in the reverse order, or **cbba**. That happens to be one of the two LCS's we mentioned in Example 6.12. We can obtain other LCS's by choosing to go left instead of down when $L(i,j)$ equals both $L(i,j-1)$ and $L(i-1,j)$, and by choosing to go left or down when one of these equals $L(i,j)$, even in the situation when $a_i = b_j$ (i.e., by skipping certain matches in favor of matches farther to the left). ✦

We can prove that this path finding algorithm always finds an LCS. The statement that we prove by complete induction on the sum of the lengths of the lists is:

STATEMENT $S(k)$: If we find ourselves at row i and column j, where $i+j=k$, and if $L(i,j)=v$, then we subsequently discover v elements for our LCS.

BASIS. The basis is $k=0$. If $i+j=0$, then both i and j are 0. We have finished our path and find no more elements for the LCS. As we know $L(0,0)=0$, the inductive hypothesis holds for $i+j=0$.

INDUCTION. Assume the inductive hypothesis for sums k or less, and let $i+j=k+1$. Suppose we are at $L(i,j)$, which has value v. If $a_i = b_j$, then we find one match and move to $L(i-1,j-1)$. Since the sum $(i-1)+(j-1)$ is less than $i+j$, the inductive hypothesis applies. Since $L(i-1,j-1)$ must be $v-1$, we know that we shall find $v-1$ more elements for our LCS, which, with the one element just found, will give us v elements. That observation proves the inductive hypothesis in this case.

The only other case is when $a_i \neq b_j$. Then, either $L(i-1,j)$ or $L(i,j-1)$, or both, must have the value v, and we move to one of these positions that does have the value v. Since the sum of the row and column is $i+j-1$ in either case, the inductive hypothesis applies, and we conclude that we find v elements for the LCS. Again we can conclude that $S(k+1)$ is true. Since we have considered all cases, we are done and conclude that if we are at an entry $L(i,j)$, we always find $L(i,j)$ elements for our LCS.

EXERCISES

6.9.1: What is the length of the LCS of the lists

a) **banana** and **cabana**
b) **abaacbacab** and **bacabbcaba**

6.9.2*: Find all the LCS's of the pairs of lists from Exercise 6.9.1. *Hint*: After building the table from Exercise 6.9.1, trace backward from the upper right corner, following each choice in turn when you come to a point that could be explained in two or three different ways.

6.9.3**: Suppose we use the recursive algorithm for computing the LCS that we described first (instead of the table-filling program that we recommend). If we call $L(4,4)$ with two lists having no symbols in common, how many calls to $L(1,1)$ are made? *Hint*: Use a table-filling (dynamic programming) algorithm to compute a table giving the value of $L(i,j)$ for all i and j. Compare your result with Pascal's triangle from Section 4.5. What does this relationship suggest about a formula for the number of calls?

6.9.4**: Suppose we have two lists x and y, each of length n. For n below a certain size, there can be at most one string that is an LCS of x and y (although that string may occur in different positions of x and/or y). For example, if $n = 1$, then the LCS can only be ϵ, unless x and y are both the same symbol a, in which case a is the only LCS. What is the smallest value of n for which x and y can have two different LCS's?

6.9.5: Show that the program of Fig. 6.31 has running time $O(mn)$.

6.9.6: Write a C program to take a table, such as that computed by the program of Fig. 6.31, and find the positions, in each string, of one LCS. What is the running time of your program, if the table is m by n?

6.9.7: In the beginning of this section, we suggested that the length of an LCS and the size of the largest matching between positions of two strings were related.

a*) Prove by induction on k that if two strings have a common subsequence of length k, then they have a matching of length k.

b) Prove that if two strings have a matching of length k, then they have a common subsequence of length k.

c) Conclude from (a) and (b) that the lengths of the LCS and the greatest size of a matching are the same.

6.10 Representing Character Strings

Character strings are probably the most common form of list encountered in practice. There are a great many ways to represent strings
are rarely appropriate for other kinds of lists.
to the special issues regarding character strings.

First, we should realize that storing a single character string is rarely the whole problem. Often, we have a large number of character strings, each rather short. They may form a dictionary, meaning that we insert and delete strings from the population as time goes on, or they may be a *static* set of strings, unchanging over time. The following are two typical examples.

Concordance

1. A useful tool for studying texts is a *concordance,* a list of all the words used in the document and the places in which they occur. There will typically be tens of thousands of different words used in a large document, and each occurrence must be stored once. The set of words used is static; that is, once formed it does not change, except perhaps if there were errors in the original concordance.

2. The compiler that turns a C program into machine code must keep track of all the character strings that represent variables of the program. A large program may have hundreds or thousands of variable names, especially when we remember that two local variables named `i` that are declared in two functions are really two distinct variables. As the compiler scans the program, it finds new variable names and inserts them into the set of names. Once the compiler has finished compiling a function, the variables of that function are not available to subsequent functions, and so may be deleted.

In both of these examples, there will be many short character strings. Short words abound in English, and programmers like to use single letters such as `i` or `x` for variables. On the other hand, there is no limit on the length of words, either in English texts or in programs.

Character Strings in C

Null character

Character-string constants, as might appear in a C program, are stored as arrays of characters, followed by the special character `'\0'`, called the *null character*, whose value is 0. However, in applications such as the ones mentioned above, we need the facility to create and store new strings as a program runs. Thus, we need a data structure in which we can store arbitrary character strings. Some of the possibilities are:

Truncation

1. Use a fixed-length array to hold character strings. Strings shorter than the array are followed by a null character. Strings longer than the array cannot be stored in their entirety. They must be *truncated* by storing only their prefix of length equal to the length of the array.

2. A scheme similar to (1), but assume that every string, or prefix of a truncated string, is followed by the null character. This approach simplifies the reading of strings, but it reduces by one the number of string characters that can be stored.

3. A scheme similar to (1), but instead of following strings by a null character, use another integer *length* to indicate how long the string really is.

4. To avoid the restriction of a maximum string length, we can store the characters of the string as the elements of a linked list. Possibly, several characters can be stored in one cell.

5. We may create a large array of characters in which individual character strings are placed. A string is then represented by a pointer to a place in the array where the string begins. Strings may be terminated by a null character or they may have an associated length.

Fixed-Length Array Representations

Let us consider a structure of type (1) above, where strings are represented by fixed-length arrays. In the following example, we create structures that have a fixed-length array as one of their fields.

✦ **Example 6.16.** Consider the data structure we might use to hold one entry in a concordance, that is, a single word and its associated information. We need to hold

1. The word itself.
2. The number of times the word appears.
3. A list of the lines of the document in which there are one or more occurrences of the word.

Thus we might use the following structure:

```
typedef struct {
    char word[MAX];
    int occurrences;
    LIST lines;
} WORDCELL;
```

Here, `MAX` is the maximum length of a word. All `WORDCELL` structures have an array called `word` of `MAX` bytes, no matter how short the word happens to be.

The field `occurrences` is a count of the number of times the word appears, and `lines` is a pointer to the beginning of a linked list of cells. These cells are of the conventional type defined by the macro

```
DefCell(int, CELL, LIST);
```

Each cell holds one integer, representing a line on which there are one or more occurrences of the word in question. Note that `occurrences` could be larger than the length of the list, if the word appeared several times on one line.

In Fig. 6.34 we see the structure for the word `earth` in the first chapter of Genesis. We assume `MAX` is at least 6. The complete list of line (verse) numbers is (1, 2, 10, 11, 12, 15, 17, 20, 22, 24, 25, 26, 28, 29, 30).

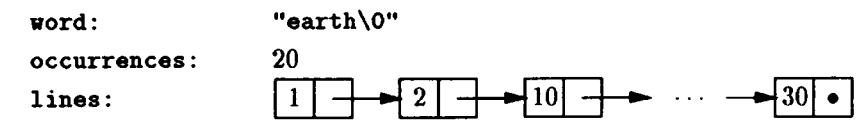


Fig. 6.34. Concordance entry for the word `earth` in the first chapter of Genesis.

The entire concordance might consist of a collection of structures of type `WORDCELL`. These might, for example, be organized in a binary search tree, with the < ordering of structures based on the alphabetic order of words. That structure would allow relatively fast access to words as we use the concordance. It would also allow us to create the concordance efficiently as we scan the text to locate and list the occurrences of the various words. To use the binary tree structure we would require left- and right-child fields in the type `WORDCELL`. We could also arrange these

structures in a linked list, by adding a "next" field to the type WORDCELL instead. That would be a simpler structure, but it would be less efficient if the number of words is large. We shall see, in the next chapter, how to arrange these structures in a hash table, which probably offers the best performance of all data structures for this problem. ✦

Linked Lists for Character Strings

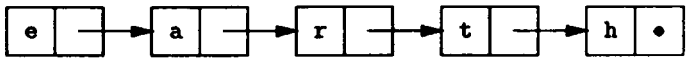
The limitation on the length of character strings, and the need to allocate a fixed amount of space no matter how short the string, are two disadvantages of the previous implementation of character strings. However, C and other languages allow us to build other, more flexible data structures to represent strings. For example, if we are concerned that there be no upper limit on the length of a character string, we can use conventional linked lists of characters to hold character strings. That is, we can declare a type

```
typedef struct CHARCELL *CHARSTRING;
struct CHARCELL {
    char character;
    CHARSTRING next;
};
```

In the type WORDCELL, CHARSTRING becomes the type of the field word, as

```
typedef {
    CHARSTRING word;
    int occurrences;
    LIST lines;
} WORDCELL;
```

For example, the word earth would be represented by



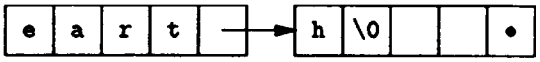
This scheme removes any upper limit on the length of words, but it is, in practice, not very economical of space. The reason is that each structure of type CHARCELL takes at least five bytes, assuming one for the character and a typical four for a pointer to the next cell on the list. Thus, the great majority of the space is used for the "overhead" of pointers rather than the "payload" of characters.

Packing characters into cells

We can be a bit more clever, however, if we pack several bytes into the data field of each cell. For example, if we put four characters into each cell, and pointers consume four bytes, then half our space will be used for "payload," compared with 20% payload in the one-character-per-cell scheme. The only caution is that we must have some character, such as the null character, that can serve as a string-terminating character, as is the case for character strings stored in arrays. In general, if CPC (characters per cell) is the number of characters that we are willing to place in one cell, we can declare cells by

```
typedef struct CHARCELL *CHARSTRING;
struct CHARCELL {
    char characters[CPC];
    CHARSTRING next;
};
```

For example, if CPC = 4, then we could store the word **earth** in two cells, as



We could also increase CPC above 4. As we do so, the fraction of space taken for pointers decreases, which is good; it means that the overhead of using linked lists rather than arrays is dropping. On the other hand, if we used a very large value for CPC, we would find that almost all words used only one cell, but that cell would have many unused locations in it, just as an array of length CPC would.

✦ **Example 6.17.** Let us suppose that in our population of character strings, 30% are between 1 and 4 characters long, 40% between 5 and 8 characters, 20% in the range 9–12, and 10% in the range 13–16. Then the table in Fig. 6.35 gives the number of bytes devoted to linked lists representing words in the four ranges, for four values of CPC, namely, 4, 8, 12, and 16. For our assumption about word-length frequencies, CPC = 8 comes out best, with an average usage of 15.6 bytes That is, we are best off using cells with room for 8 bytes, using a total of 12 bytes per cell, including the 4 bytes for the **next** pointer. Note that the total space cost, which is 19.6 bytes when we include a pointer to the front of the list, is not as good as using 16 bytes for a character array. However, the linked-list scheme can accommodate strings longer than 16 characters, even though our assumptions put a 0% probability on finding such strings. ✦

		CHARACTERS PER CELL			
RANGE	PROBABILITY	4	8	12	16
1-4	.3	8	12	16	20
5-8	.4	16	12	16	20
9-12	.2	24	24	16	20
13-16	.1	32	24	32	20
Avg.		16.8	15.6	17.6	20.0

Fig. 6.35. Numbers of bytes used for strings in various length ranges by different values of CPC.

Mass Storage of Character Strings

There is another approach to the storage of large numbers of character strings that combines the advantage of array storage (little overhead) with the advantages of linked-list storage (no wasted space due to padding, and no limit on string length). We create one very long array of characters, into which we shall store each character string. To tell where one string ends and the next begins, we need a special character called the *endmarker*. The endmarker character cannot appear as part of a legitimate character string. In what follows, we shall use * as the endmarker, for visibility, although it is more usual to choose a nonprinting character, such as the null character.

Endmarker

✦ **Example 6.18.** Suppose we declare an array **space** by

```
char space[MAX];
```

We can then store a word by giving a pointer to the first position of **space** devoted to that word. The **WORDCELL** structure, analogous to that of Example 6.16, would then be

```
typedef struct {
    char *word;
    int occurrences;
    LIST lines;
} WORDCELL;
```

In Fig. 6.36 we see the **WORDCELL** structure for the word **the** in a concordance based on the book of Genesis. The pointer **word** refers us to **space[3]**, where we see the beginning of the word **the**.

Note that the lowest elements of the array **space** might appear to contain the text itself. However, that would not continue to be the case for long. Even if the next elements contain the words **beginning**, **God**, and **created**, the second **the** would not appear again in the array **space**. Rather, that word would be accounted for by adding to the number of occurrences in the **WORDCELL** structure for **the**. As we proceeded through the book and found more repetitions of words, the entries in **space** would stop resembling the biblical text itself. ✦

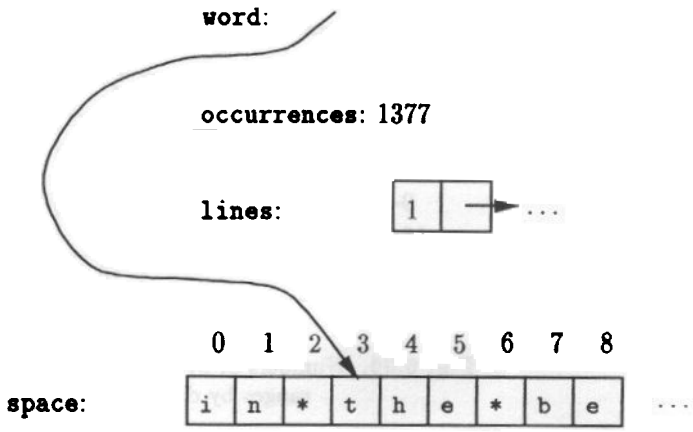


Fig. 6.36. Representing words by indices into string space.

As in Example 6.16, the structures of Example 6.18 can be formed into data structures such as binary search trees or linked lists by adding the appropriate pointer fields to the **WORDCELL** structure. The function $lt(W_1, W_2)$ that compares two **WORDCELL**'s W_1 and W_2 follows the **word** fields of these structures and compares them lexicographically.

To build a concordance using such a binary search tree, we maintain a pointer **available** to the first unoccupied position in the array **space**. Initially, **available** points to **space[0]**. Suppose we are scanning the text for which the concordance

What Happens When We Run Out of Space?

We have assumed that **space** is so large that there is always room to add a new word. Actually, each time we add a character we must be careful that the current position into which we write is less than **MAX**.

If we want to enter new words after running out of space, we need to be prepared to obtain new blocks of space when the old one runs out. Instead of creating just one array **space**, we can define a character-array type

```
typedef char SPACE[MAX];
```

We can then create a new array, the first character of which is pointed to by **available**, by

```
available = (char *) malloc(sizeof(SPACE));
```

It is useful to remember the end of this array by immediately assigning

```
last = available + MAX;
```

We then insert words into the array pointed to by **available**. If we can no longer fit words into this array, we call **malloc** to create another character array. Of course we must be careful not to write past the end of the array, and if we are presented with a string of length greater than MAX, there is no way we can store the word in this scheme.

is being built and we find the next word — say, **the**. We do not know whether or not **the** is already in the binary search tree. We thus temporarily add **the*** to the position indicated by **available** and the three following positions. We remember that the newly added word takes up 4 bytes.

Now we can search for the word **the** in the binary search tree. If found, we add 1 to its count of occurrences and insert the current line into the list of lines. If not found, we create a new node — which includes the fields of the **WORDCELL** structure, plus left- and right-child pointers (both **NULL**) — and insert it into the tree at the proper place. We set the **word** field in the new node to *available*, so that it refers to our copy of the word **the**. We set **occurrences** to 1 and create a list for the field **lines** consisting of only the current line of text. Finally, we must add 4 to **available**, since the word **the** has now been added permanently to the **space** array.

EXERCISES

6.10.1: For the structure type **WORDCELL** discussed in Example 6.16, write the following programs:

a) A function **create** that returns a pointer to a structure of type **WORDCELL**.

b) A function **insert(WORDCELL *pWC, int line)** that takes a pointer to the structure **WORDCELL** and a line number, adds 1 to the number of occurrences for that word, and adds that line to the list of lines if it is not already there.

6.10.2: Redo Example 6.17 under the assumption that any word length from 1 to 40 is equally likely; that is, 10% of the words are of length 1–4, 10% are of length 5–8, and so on, up to 10% in the range 37–40. What is the average number of bytes required if `CPC` is 4, 8, ..., 40?

6.10.3*: If, in the model of Example 6.17, all word lengths from 1 to n are equally likely, what value of `CPC`, as a function of n, minimizes the number of bytes used? If you cannot get the exact answer, a big-oh approximation is useful.

6.10.4*: One advantage of using the structure of Example 6.18 is that one can share parts of the space array among two or more words. For example, the structure for the word `he` could have `word` field equal to 5 in the array of Fig. 6.36. Compress the words `all`, `call`, `man`, `mania`, `maniac`, `recall`, `two`, `woman` into as few elements of the `space` array as you can. How much space do you save by compression?

6.10.5*: Another approach to storing words is to eliminate the endmarker character from the `space` array. Instead, we add a `length` field to the `WORDCELL` structures of Example 6.18, to tell us how many characters from the first character, as indicated by the `word` field, are included in the word. Assuming that integers take four bytes, does this scheme save or cost space, compared with the scheme described in Example 6.18? What if integers could be stored in one byte?

6.10.6**: The scheme described in Exercise 6.10.5 also gives us opportunities to compress the `space` array. Now words can overlap even if neither is a suffix of the other. How many elements of the space array do you need to store the words in the list of Exercise 6.10.4, using the scheme of Exercise 6.10.5?

6.10.7: Write a program to take two `WORDCELL`'s as discussed in Example 6.18 and determine which one's word precedes the other in lexicographic order. Recall that words are terminated by * in this example.

6.11 Summary of Chapter 6

The following points were covered in Chapter 6.

- Lists are an important data model representing sequences of elements.
- Linked lists and arrays are two data structures that can be used to implement lists.
- Lists are a simple implementation of dictionaries, but their efficiency does not compare with that of the binary search tree of Chapter 5 or the hash table to be covered in Chapter 7.
- Placing a "sentinel" at the end of an array to make sure we find the element we are seeking is a useful efficiency improver.
- Stacks and queues are important special kinds of lists.
- The stack is used "behind the scenes" to implement recursive functions.

- A character string is an important special case of a list, and we have a number of special data structures for representing character strings efficiently. These include linked lists that hold several characters per cell and large arrays shared by many character strings.
- The problem of finding longest common subsequences can be solved efficiently by a technique known as "dynamic programming," in which we fill a table of information in the proper order.

6.12 Bibliographic Notes for Chapter 6

Knuth [1968] is still the fundamental source on list data structures. While it is hard to trace the origins of very basic notions such as "list" or "stack," the first programming language to use lists as a part of its data model was IPL-V (Newell et al. [1961]), although among the early list-processing languages, only Lisp (McCarthy et al. [1962]) survives among the currently important languages. Lisp, by the way, stands for "LISt Processing."

The use of stacks in run-time implementation of recursive programs is discussed in more detail in Aho, Sethi, and Ullman [1986].

The longest-common-subsequence algorithm described in Section 6.9 is by Wagner and Fischer [1975]. The algorithm actually used in the UNIX `diff` command is described in Hunt and Szymanski [1977]. Aho [1990] surveys a number of algorithms involving the matching of character strings.

Dynamic programming as an abstract technique was described by Bellman [1957]. Aho, Hopcroft, and Ullman [1983] give a number of examples of algorithms using dynamic programming.

Aho, A. V. [1990]. "Algorithms for finding patterns in strings," in *Handbook of Theoretical Computer Science* Vol. A: *Algorithms and Complexity* (J. Van Leeuwen, ed.), MIT Press, Cambridge, Mass.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman [1983]. *Data Structures and Algorithms*, Addison-Wesley, Reading, Mass.

Aho, A. V., R. Sethi, and J. D. Ullman [1986]. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass.

Bellman, R. E. [1957]. *Dynamic Programming*, Princeton University Press, Princeton, NJ.

Hunt, J. W. and T. G. Szymanski [1977]. "A fast algorithm for computing longest common subsequences," *Comm. ACM* **20**:5, pp. 350–353.

Knuth, D. E. [1968]. *The Art of Computer Programming*, Vol. I, *Fundamental Algorithms*, Addison-Wesley, Reading, Mass.

McCarthy, J. et al. [1962]. *LISP 1.5 Programmer's Manual*, MIT Computation Center and Research Laboratory of Electronics, Cambridge, Mass.

Newell, A., F. M. Tonge, E. A. Feigenbaum, B. F. Green, and G. H. Mealy [1961]. *Information Processing Language-V Manual*, Prentice-Hall, Englewood Cliffs, New Jersey.

Wagner, R. A. and M. J. Fischer [1975]. "The string to string correction problem," *J. ACM* **21**:1, pp. 168–173.

CHAPTER 7

The Set Data Model

The set is the most fundamental data model of mathematics. Every concept in mathematics, from trees to real numbers, is expressible as a special kind of set. In this book, we have seen sets in the guise of events in a probability space. The dictionary abstract data type is a kind of set, on which particular operations — *insert*, *delete*, and *lookup* — are performed. Thus, it should not be surprising that sets are also a fundamental model of computer science. In this chapter, we learn the basic definitions concerning sets and then consider algorithms for efficiently implementing set operations.

7.1 What This Chapter Is About

This chapter covers the following topics:

- The basic definitions of set theory and the principal operations on sets (Sections 7.2–7.3).

  The three most common data structures used to implement sets: linked lists, characteristic vectors, and hash tables. We compare these data structures with respect to their relative efficiency in supporting various operations on sets (Sections 7.4–7.6).

- Relations and functions as sets of pairs (Section 7.7).
- Data structures for representing relations and functions (Sections 7.8–7.9).
- Special kinds of binary relations, such as partial orders and equivalence relations (Section 7.10).
- Infinite sets (Section 7.11).

7.2 Basic Definitions

In mathematics, the term "set" is not defined explicitly. Rather, like terms such as "point" and "line" in geometry, the term set is defined by its properties. Specifically, there is a notion of *membership* that makes sense only for sets. When S is a set and x is anything, we can ask the question, "Is x a member of set S?" The set S then consists of all those elements x for which x is a member of S. The following points summarize some important notations for talking about sets.

1. The expression $x \in S$ means that the element x is a member of the set S.
2. If $x_1, x_2, \ldots, x_n$ are all the members of set S, then we can write

   $$S = \{x_1, x_2, \ldots, x_n\}$$

   Here, each of the x's must be distinct; we cannot repeat an element twice in a set. However, the order in which the members of a set are listed is arbitrary.

Empty set

3. The empty set, denoted $\emptyset$, is the set that has no members. That is, $x \in \emptyset$ is false, no matter what x is.

✦ **Example 7.1.** Let $S = \{1, 3, 6\}$; that is, let S be the set that has the integers 1, 3, and 6, and nothing else, as members. We can say $1 \in S$, $3 \in S$, and $6 \in S$. However, the statement $2 \in S$ is false, as is the statement that any other thing is a member of S.

Sets can also have other sets as members. For example, let $T = \{\{1, 2\}, 3, \emptyset\}$. Then T has three members. First is the set $\{1, 2\}$, that is, the set with 1 and 2 as is sole members. Second is the integer 3. Third is the empty set. The following are true statements: $\{1, 2\} \in T$, $3 \in T$, and $\emptyset \in T$. However, $1 \in T$ is false. That is, the fact that 1 is a member of a member of T does not mean that 1 is a member of T itself. ✦

Atoms

In formal set theory, there really is nothing but sets. However, in our informal set theory, and in data structures and algorithms based on sets, it is convenient to assume the existence of certain *atoms*, which are elements that are not sets. An atom can be a member of a set, but nothing can be a member of an atom. It is important to remember that the empty set, like the atoms, has no members. However, the empty set is a set rather than an atom.

We shall generally assume that integers and lowercase letters denote atoms. When talking about data structures, it is often convenient to use complex data types as the types of atoms. Thus, atoms may be structures or arrays, and not be very "atomic" at all.

Definition of Sets by Abstraction

Enumeration of the members of a set is not the only way we may define sets. Often, it is more convenient to start with some set S and some property of elements P, and define the set of those elements in S that have property P. The notation for this operation, which is called *abstraction*, is

Sets and Lists

Although our notation for a list, such as $(x_1, x_2, \ldots, x_n)$, and our notation for a set, $\{x_1, x_2, \ldots, x_n\}$, look very much alike, there are important differences. First, the order of elements in a set is irrelevant. The set we write as $\{1, 2\}$ could just as well be written $\{2, 1\}$. In contrast, the list $(1, 2)$ is not the same as the list $(2, 1)$.

Second, a list may have repetitions. For example, the list $(1, 2, 2)$ has three elements; the first is 1, the second is 2, and the third is also 2. However, the set notation $\{1, 2, 2\}$ makes no sense. We cannot have an element, such as 2, occur as a member of a set more than once. If this notation means anything, it is the same as $\{1, 2\}$ or $\{2, 1\}$ — that is, the set with 1 and 2 as members, and no other members.

Multiset or bag

Sometimes we speak of a *multiset* or *bag*, which is a set whose elements are allowed to have a multiplicity greater than 1. For example, we could speak of the multiset that contains 1 once and 2 twice. Multisets are not the same as lists, because they still have no order associated with their elements.

$$\{x \mid x \in S \text{ and } P(x)\}$$

or "the set of elements x in S such that x has property P."

Set former

The preceding expression is called a *set former*. The variable x in the set former is local to the expression, and we could just as well have written

$$\{y \mid y \in S \text{ and } P(y)\}$$

to describe the same set.

✦ **Example 7.2.** Let S be the set $\{1, 3, 6\}$ from Example 7.1. Let $P(x)$ be the property "x is odd." Then

$$\{x \mid x \in S \text{ and } x \text{ is odd }\}$$

is another way of defining the set $\{1, 3\}$. That is, we accept the elements 1 and 3 from S because they are odd, but we reject 6 because it is not odd.

As another example, consider the set $T = \{\{1, 2\}, 3, \emptyset\}$ from Example 7.1. Then

$$\{A \mid A \in T \text{ and } A \text{ is a set }\}$$

denotes the set $\{\{1, 2\}, \emptyset\}$. ✦

Equality of Sets

We must not confuse what a set *is* with how it is represented. Two sets are *equal*, that is, they are really the same set, if they have exactly the same members. Thus, most sets have many different representations, including those that explicitly enumerate their elements in some order and representations that use abstraction.

✦ **Example 7.3.** The set $\{1, 2\}$ is the set that has exactly the elements 1 and 2 as members. We can present these elements in either order, so $\{1, 2\} = \{2, 1\}$. There are also many ways to express this set by abstraction. For example,

$$\{x \mid x \in \{1,2,3\} \text{ and } x < 3\}$$

is equal to the set $\{1,2\}$. ✦

Infinite Sets

It is comforting to assume that sets are *finite* — that is, that there is some particular integer n such that the set at hand has exactly n members. For example, the set $\{1,3,6\}$ has three members. However, some sets are *infinite*, meaning there is no integer that is the number of elements in the set. We are familiar with infinite sets such as

1. **N**, the set of nonnegative integers
2. **Z**, the set of nonnegative and negative integers
3. **R**, the set of real numbers
4. **C**, the set of complex numbers

From these sets, we can create other infinite sets by abstraction.

✦ **Example 7.4.** The set former

$$\{x \mid x \in \mathbf{Z} \text{ and } x < 3\}$$

stands for the set of all the negative integers, plus 0, 1, and 2. The set former

$$\{x \mid x \in \mathbf{Z} \text{ and } \sqrt{x} \in \mathbf{Z}\}$$

represents the set of integers that are perfect squares, that is, $\{0,1,4,9,16,\ldots\}$.

For a third example, let $P(x)$ be the property that x is prime (i.e., $x > 1$ and x has no divisors except 1 and x itself). Then the set of primes is denoted

$$\{x \mid x \in \mathbf{N} \text{ and } P(x)\}$$

This expression denotes the infinite set $\{2,3,5,7,11,\ldots\}$. ✦

There are some subtle and interesting properties of infinite sets. We shall take up the matter again in Section 7.11.

EXERCISES

7.2.1: What are the members of the set $\{\{a,b\},\{a\},\{b,c\}\}$?

7.2.2: Write set-former expressions for the following:

a) The set of integers greater than 1000.
b) The set of even integers.

7.2.3: Find two different representations for the following sets, one using abstraction, the other not.

a) $\{a,b,c\}$.
b) $\{0,1,5\}$.

Russell's Paradox

One might wonder why the operation of abstraction requires that we designate some other set from which the elements of the new set must come. Why can't we just use an expression like $\{x \mid P(x)\}$, for example,

$$\{x \mid x \text{ is blue }\}$$

to define the set of all blue things? The reason is that allowing such a general way to define sets gets us into a logical inconsistency discovered by Bertrand Russell and called *Russell's paradox*. We may have met this paradox informally when we heard about the town where the barber shaves everyone who doesn't shave himself, and then were asked whether the barber shaves himself. If he does, then he doesn't, and if he doesn't, he does. The way out of this anomaly is to realize that the statement "shaves everyone who doesn't shave himself," while it looks reasonable, actually makes no formal sense.

To understand Russell's paradox concerning sets, suppose we could define sets of the form $\{x \mid P(x)\}$ for any property P. Then let $P(x)$ be the property "x is not a member of x." That is, let P be true of a set x if x is not a member of itself. Let S be the set

$$S = \{x \mid x \text{ is not a member of } x\}$$

Now we can ask, "Is set S a member of itself?"

Case 1: Suppose that S is not a member of S. Then $P(S)$ is true, and so S is a member of the set $\{x \mid x \text{ is not a member of } x\}$. But that set is S, and so by assuming that S is not a member of itself, we prove that S is indeed a member of itself. Thus, it cannot be that S is not a member of itself.

Case 2: Suppose that S is a member of itself. Then S is not a member of

$$\{x \mid x \text{ is not a member of } x\}$$

But again, that set is S, and so we conclude that S is not a member of itself.

Thus, when we start by assuming that $P(S)$ is false, we prove that it is true, and when we start by assuming that $P(S)$ is true, we wind up proving that it is false. Since we arrive at a contradiction either way, we are forced to blame the notation. That is, the real problem is that it makes no sense to define the set S as we did.

Another interesting consequence of Russell's paradox is that it makes no sense to suppose there is a "set of all elements." If there were such a "universal set" — say U — then we could speak of

Universal set

$$\{x \mid x \in U \text{ and } x \text{ is not a member of } x\}$$

and we would again have Russell's paradox. We would then be forced to give up abstraction altogether, and that operation is far too useful to drop.

7.3 Operations on Sets

There are special operations that are commonly performed on sets, such as union and intersection. You are probably familiar with many of them, but we shall review the most important operations here. In the next sections we discuss some implementations of these operations.

Union, Intersection, and Difference

Perhaps the most common ways to combine sets are with the following three operations:

1. The *union* of two sets S and T, denoted $S \cup T$, is the set containing the elements that are in S or T, or both.
2. The *intersection* of sets S and T, written $S \cap T$, is the set containing the elements that are in both S and T.
3. The *difference* of sets S and T, denoted $S - T$, is the set containing those elements that are in S but not in T.

✦ **Example 7.5.** Let S be the set $\{1, 2, 3\}$ and T the set $\{3, 4, 5\}$. Then

$$S \cup T = \{1, 2, 3, 4, 5\}, S \cap T = \{3\}, \text{ and } S - T = \{1, 2\}$$

That is, $S \cup T$ contains all the elements appearing in either S or T. Although 3 appears in both S and T, there is, of course, only one occurrence of 3 in $S \cup T$, because elements cannot appear more than once in a set. $S \cap T$ contains only 3, because no other element appears in both S and T. Finally, $S - T$ contains 1 and 2, because these appear in S and do not appear in T. The element 3 is not present in $S - T$, because although it appears in S, it also appears in T. ✦

When the sets S and T are events in a probability space, the union, intersection, and difference have a natural meaning. $S \cup T$ is the event that either S or T (or both) occurs. $S \cap T$ is the event that both S and T occur. $S - T$ is the event that S, but not T occurs. However, if S is the set that is the entire probability space, then $S - T$ is the event "T does not occur," that is, the complement of T.

Venn Diagrams

It is often helpful to see operations involving sets as pictures called *Venn diagrams*. Figure 7.1 is a Venn diagram showing two sets, S and T, each of which is represented by an ellipse. The two ellipses divide the plane into four regions, which we have numbered 1 to 4.

1. Region 1 represents those elements that are in neither S nor T.
2. Region 2 represents $S - T$, those elements that are in S but not in T.
3. Region 3 represents $S \cap T$, those elements that are in both S and T.
4. Region 4 represents $T - S$, those elements that are in T but not in S.
5. Regions 2, 3, and 4 combined represent $S \cup T$, those elements that are in S or T, or both.

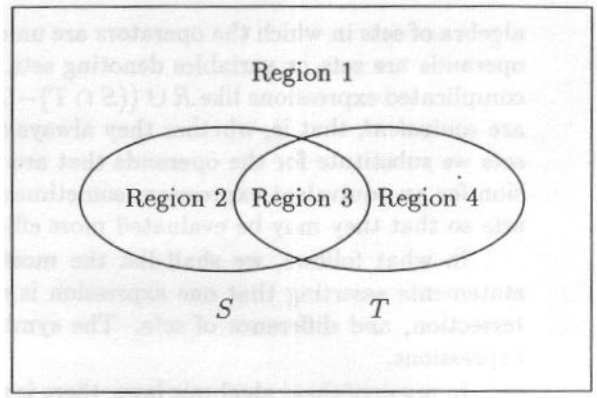


Fig. 7.1. Regions representing Venn diagrams for the basic set operations.

What Is an Algebra?

We may think that the term "algebra" refers to solving word problems, finding roots of polynomials, and other matters covered in a high school algebra course. To a mathematician, however, the term algebra refers to any sort of system in which there are operands and operators from which one builds expressions. For an algebra to be interesting and useful, it usually has special constants and *laws* that allow us to transform one expression into another "equivalent" expression.

The most familiar algebra is that in which operands are integers, reals, or perhaps complex numbers — or variables representing values from one of these classes — and the operators are the ordinary arithmetic operators: addition, multiplication, subtraction, and division. The constants 0 and 1 are special and satisfy laws like $x + 0 = x$. In manipulating arithmetic expressions, we use laws such as the distributive law, which lets us replace any expression of the form $a \times b + a \times c$ by an equivalent expression $a \times (b + c)$. Notice that by making this transformation, we reduce the number of arithmetic operations by 1. Often the purpose of algebraic manipulation of expressions, such as this one, is to find an equivalent expression whose evaluation takes less time than the evaluation of the original.

Throughout this book, we shall meet various kinds of algebras. Section 8.7 introduces relational algebra, a generalization of the algebra of sets that we discuss here. Section 10.5 talks about the algebra of *regular expressions* for describing patterns of character strings. Section 12.8 introduces the reader to the *Boolean* algebra of logic.

While we have suggested that Region 1 in Fig. 7.1 has finite extent, we should remember that this region represents everything outside S and T. Thus, this region is *not* a set. If it were, we could take its union with S and T to get the "universal set," which we know by Russell's paradox does not exist. Nevertheless, it is often convenient to draw as a region the elements that are not in any of the sets represented explicitly in the Venn diagram, as we did in Fig. 7.1.

Algebraic Laws For Union, Intersection, and Difference

Mirroring the algebra of arithmetic operations such as $+$ and $*$, one can define an

algebra of sets in which the operators are union, intersection, and difference and the operands are sets or variables denoting sets. Once we allow ourselves to build up complicated expressions like $R \cup ((S \cap T) - U)$, we can ask whether two expressions are *equivalent*, that is, whether they always denote the same set regardless of what sets we substitute for the operands that are variables. By substituting one expression for an equivalent expression, sometimes we can simplify expressions involving sets so that they may be evaluated more efficiently.

Equivalent expressions

In what follows, we shall list the most important *algebraic laws* — that is, statements asserting that one expression is equivalent to another — for union, intersection, and difference of sets. The symbol $\equiv$ is used to denote equivalence of expressions.

In many of these algebraic laws, there is an analogy between union, intersection, and difference of sets, on one hand, and addition, multiplication, and subtraction of integers on the other hand. We shall, however, point out those laws that do not have analogs for ordinary arithmetic.

a) *The commutative law of union*: $(S \cup T) \equiv (T \cup S)$. That is, it does not matter which of two sets appears first in a union. The reason this law holds is simple. The element x is in $S \cup T$ if x is in S or if x is in T, or both. That is exactly the condition under which x is in $T \cup S$.

b) *The associative law of union*: $(S \cup (T \cup R)) \equiv ((S \cup T) \cup R)$. That is, the union of three sets can be written either by first taking the union of the first two or the last two; in either case, the result will be the same. We can justify this law as we did the commutative law, by arguing that an element is in the set on the left if and only if it is in the set on the right. The intuitive reason is that both sets contain exactly those elements that are in either S, T, or R, or any two or three of them.

The commutative and associative laws of union together tell us that we can take the union of a collection of sets in any order. The result will always be the same set of elements, namely those elements that are in one or more of the sets. The argument is like the one we presented for addition, which is another commutative and associative operation, in Section 2.4. There, we showed that all ways to group a sum led to the same result.

c) *The commutative law of intersection*: $(S \cap T) \equiv (T \cap S)$. Intuitively, an element x is in the sets $S \cap T$ and $T \cap S$ under exactly the same circumstances: when x is in S and x is in T.

d) *The associative law of intersection*: $(S \cap (T \cap R)) \equiv ((S \cap T) \cap R)$. Intuitively, x is in either of these sets exactly when x is in all three of S, T, and R. Like addition or union, the intersection of any collection of sets may be grouped as we choose, and the result will be the same; in particular, it will be the set of elements in all the sets.

e) *Distributive law of intersection over union*: Just as we know that multiplication distributes over addition — that is, $a \times (b + c) = a \times b + a \times c$ — the law

$$(S \cap (T \cup R)) \equiv ((S \cap T) \cup (S \cap R))$$

holds for sets. Intuitively, an element x is in each of these sets exactly when x is in S and also in at least one of T and R. Similarly, by the commutativity of

union and intersection, we can distribute intersections from the right, as

$$((T \cup R) \cap S) \equiv ((T \cap S) \cup (R \cap S))$$

f) *Distributive law of union over intersection*: Similarly,

$$(S \cup (T \cap R)) \equiv ((S \cup T) \cap (S \cup R))$$

holds. Both the left and right sides are sets that contain an element x exactly when x is either in S, or is in both T and R. Notice that the analogous law of arithmetic, where union is replaced by addition and intersection by multiplication, is false. That is, $a+b\times c$ is generally not equal to $(a+b)\times(a+c)$. Here is one of several places where the analogy between set operations and arithmetic operations breaks down. However, as in (e), we can use the commutativity of union to get the equivalent law

$$((T \cap R) \cup S) \equiv ((T \cup S) \cap (R \cup S))$$

✦ **Example 7.6.** Let $S = \{1,2,3\}$, $T = \{3,4,5\}$, and $R = \{1,4,6\}$. Then

$$\begin{aligned} S \cup (T \cap R) &= \{1,2,3\} \cup (\{3,4,5\} \cap \{1,4,6\}) \\ &= \{1,2,3\} \cup \{4\} \\ &= \{1,2,3,4\} \end{aligned}$$

On the other hand,

$$\begin{aligned} (S \cup T) \cap (S \cup R) &= (\{1,2,3\} \cup \{3,4,5\}) \cap (\{1,2,3\} \cup \{1,4,6\}) \\ &= \{1,2,3,4,5\} \cap \{1,2,3,4,6\} \\ &= \{1,2,3,4\} \end{aligned}$$

Thus, the distributive law of union over intersection holds in this case. That doesn't prove that the law holds in general, of course, but the intuitive argument we gave with rule (f) should be convincing. ✦

g) *Associative law of union and difference*: $(S-(T \cup R)) \equiv ((S-T)-R)$. Both sides contain an element x exactly when x is in S but in neither T nor R. Notice that this law is analogous to the arithmetic law $a-(b+c) = (a-b)-c$.

h) *Distributive law of difference over union*: $((S \cup T)-R) \equiv ((S-R) \cup (T-R))$. In justification, an element x is in either set when it is not in R, but is in either S or T, or both. Here is another point at which the analogy with addition and subtraction breaks down; it is not true that $(a+b)-c = (a-c)+(b-c)$, unless $c = 0$.

i) *The empty set is the identity for union.* That is, $(S \cup \emptyset) \equiv S$, and by commutativity of union, $(\emptyset \cup S) \equiv S$. Informally, an element x can be in $S \cup \emptyset$ only when x is in S, since x cannot be in $\emptyset$.

Note that there is no identity for intersection. We might imagine that the set of "all elements" could serve as the identity for intersection, since the intersection of a set S with this "set" would surely be S. However, as mentioned in connection with Russell's paradox, there cannot be a "set of all elements."

Idempotence

j) *Idempotence of union.* An operator is said to be *idempotent* if, when applied to two copies of the same value, the result is that value. We see that $(S \cup S) \equiv S$. That is, an element x is in $S \cup S$ exactly when it is in S. Again the analogy with arithmetic fails, since $a + a$ is generally not equal to a.

k) *Idempotence of intersection.* Similarly, we have $(S \cap S) \equiv S$.

There are a number of laws relating the empty set to operations besides union. We list them here.

l) $(S - S) \equiv \emptyset$.

m) $(\emptyset - S) \equiv \emptyset$.

n) $(\emptyset \cap S) \equiv \emptyset$, and by commutativity of intersection, $(S \cap \emptyset) \equiv \emptyset$.

Proving Equivalences by Venn Diagrams

Figure 7.2 illustrates the distributive law for intersection over union by a Venn diagram. This diagram shows three sets, S, T, and R, which divide the plane into eight regions, numbered 1 through 8. These regions correspond to the eight possible relationships (in or out) that an element can have with the three sets.

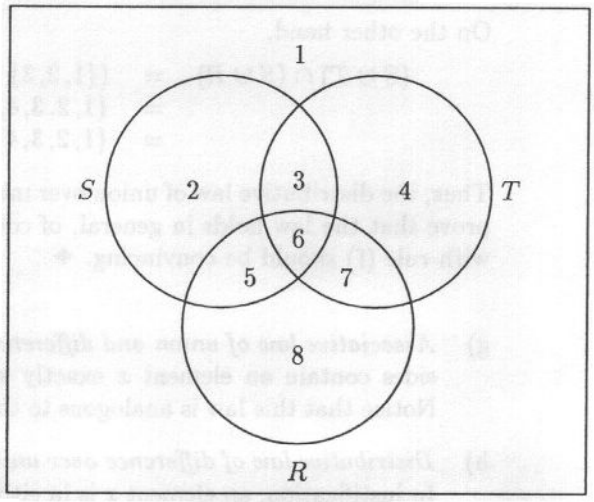


Fig. 7.2. Venn diagram showing the distributive law of intersection over union: $S \cap (T \cup R)$ consists of regions 3, 5, and 6, as does $(S \cap T) \cup (S \cap R)$.

We can use the diagram to help us keep track of the values of various subexpressions. For instance, $T \cup R$ is regions 3, 4, 5, 6, 7, and 8. Since S is regions 2, 3, 5, and 6, it follows that $S \cap (T \cup R)$ is regions 3, 5, and 6. Similarly, $S \cap T$ is regions 3 and 6, while $S \cap R$ is regions 5 and 6. It follows that $(S \cap T) \cup (S \cap R)$ is the same regions 3, 5, and 6, proving that

$$(S \cap (T \cup R)) \equiv ((S \cap T) \cup (S \cap R))$$

In general, we can prove an equivalence by considering one representative element from each region and checking that it is either in the set described by both sides of the equivalence or in neither of those sets. This method is very close to the truth-table method by which we prove algebraic laws for propositional logic in Chapter 12.

Proving Equivalences by Applying Transformations

Another way to prove two expressions equivalent is by turning one into the other using one or more of the algebraic laws we have already seen. We shall give a more formal treatment of how expressions are manipulated in Chapter 12, but for the present, let us observe that we can

1. Substitute any expression for any variable in an equivalence, provided that we substitute for all occurrences of that variable. The equivalence remains true.
2. Substitute, for a subexpression E in some equivalence, an expression F that is known to be equivalent to E. The equivalence remains true.

In addition, we can write any of the equivalences that were stated as laws and assume that equivalence is true.

✦ **Example 7.7.** We shall prove the equivalence $(S - (S \cup R)) \equiv \emptyset$. Let us start with law (g), the associative law for union and difference, which is

$$(S - (T \cup R)) \equiv ((S - T) - R)$$

We substitute S for each of the two occurrences of T to get a new equivalence:

$$(S - (S \cup R)) \equiv ((S - S) - R)$$

By law (l), $(S - S) \equiv \emptyset$. Thus, we may substitute $\emptyset$ for $(S - S)$ above to get:

$$(S - (S \cup R)) \equiv (\emptyset - R)$$

Law (m), with R substituted for S says that $\emptyset - R \equiv \emptyset$. We may thus substitute $\emptyset$ for $\emptyset - R$ and conclude that $(S - (S \cup R)) \equiv \emptyset$. ✦

The Subset Relationship

There is a family of comparison operators among sets that is analogous to the comparisons among numbers. If S and T are sets, we say that $S \subseteq T$ if every member of S is also a member of T. We can express this in words several ways: "S is a subset of T," "T is a superset of S," "S is contained in T," or "T contains S."

Containment of sets

We say that $S \subset T$, if $S \subseteq T$, and there is at least one element of T that is not also a member of S. The $S \subset T$ relationship can be read "S is a proper subset of T," "T is a proper superset of S," "S is properly contained in T," or "T properly contains S."

Proper subset

As with "less than," we can reverse the direction of the comparison; $S \supset T$ is synonymous with $T \subset S$, and $S \supseteq T$ is synonymous with $T \subseteq S$.

✦ **Example 7.8.** The following comparisons are all true:

1. $\{1,2\} \subseteq \{1,2,3\}$
2. $\{1,2\} \subset \{1,2,3\}$
3. $\{1,2\} \subseteq \{1,2\}$

Note that a set is always a subset of itself but a set is never a proper subset of itself, so that $\{1,2\} \subset \{1,2\}$ is false. ✦

There are a number of algebraic laws involving the subset operator and the other operators that we have already seen. We list some of them here.

o) $\emptyset \subseteq S$ for any set S.

p) If $S \subseteq T$, then

i) $(S \cup T) \equiv T$,

ii) $(S \cap T) \equiv S$, and

iii) $(S - T) \equiv \emptyset$.

Proving Equivalences by Showing Containments

Two sets S and T are equal if and only if $S \subseteq T$ and $T \subseteq S$; that is, each is a subset of the other. For if every element in S is an element of T and vice versa, then S and T have exactly the same members and thus are equal. Conversely, if S and T have exactly the same members, then surely $S \subseteq T$ and $T \subseteq S$ are true. This rule is analogous to the arithmetic rule that $a = b$ if and only if both $a \leq b$ and $b \leq a$ are true.

We can show the equivalence of two expressions E and F by showing that the set denoted by each is contained in the set denoted by the other. That is, we

1. Consider an arbitrary element x in E and prove that it is also in F, and then
2. Consider an arbitrary element x in F and prove that it is also in E.

Note that both proofs are necessary in order to prove that $E \equiv F$.

	STEP	REASON
1)	x is in $S - (T \cup R)$	Given
2)	x is in S	Definition of $-$ and (1)
3)	x is not in $T \cup R$	Definition of $-$ and (1)
4)	x is not in T	Definition of $\cup$ and (3)
5)	x is not in R	Definition of $\cup$ and (3)
6)	x is in $S - T$	Definition of $-$ with (2) and (4)
7)	x is in $(S - T) - R$	Definition of $-$ with (6) and (5)

Fig. 7.3. Proof of one half of the associative law for union and difference.

✦ **Example 7.9.** Let us prove the associative law for union and difference,

$$(S - (T \cup R)) \equiv ((S - T) - R)$$

We start by assuming that x is in the expression on the left. The sequence of steps is shown in Fig. 7.3. Note that in steps (4) and (5), we use the definition of union backwards. That is, (3) tells us that x is not in $T \cup R$. If x were in T, (3) would be wrong, and so we can conclude that x is not in T. Similarly, x is not in R.

	STEP	REASON
1)	x is in $(S - T) - R$	Given
2)	x is in $S - T$	Definition of $-$ and (1)
3)	x is not in R	Definition of $-$ and (1)
4)	x is in S	Definition of $-$ and (2)
5)	x is not in T	Definition of $-$ and (2)
6)	x is not in $T \cup R$	Definition of $\cup$ with (3) and (5)
7)	x is in $S - (T \cup R)$	Definition of $-$ with (4) and (6)

Fig. 7.4. Second half of the proof of the associative law for union and difference.

We are not done; we must now start by assuming that x is in $(S - T) - R$ and show that it is in $S - (T \cup R)$. The steps are shown in Fig. 7.4. ✦

✦ **Example 7.10.** As another example, let us prove part of (p), the rule that if $S \subseteq T$, then $S \cup T \equiv T$. We begin by assuming that x is in $S \cup T$. We know by the definition of union that either

1. x is in S or
2. x is in T.

In case (1), since $S \subseteq T$ is assumed, we know that x is in T. In case (2), we immediately see that x is in T. Thus, in either case x is in T, and we have completed the first half of the proof, the statement that $(S \cup T) \subseteq T$.

Now let us assume that x is in T. Then x is in $S \cup T$ by the definition of union. Thus, $T \subseteq (S \cup T)$, which is the second half of the proof. We conclude that if $S \subseteq T$ then $(S \cup T) \equiv T$. ✦

The Power Set of a Set

If S is any set, the *power set* of S is the set of subsets of S. We shall use $\mathbf{P}(S)$ to denote the power set of S, although the notation 2^S is also used.

✦ **Example 7.11.** Let $S = \{1, 2, 3\}$. Then

$$\mathbf{P}(S) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

Singleton set

That is, $\mathbf{P}(S)$ is a set with eight members; each member is itself a set. The empty set is in $\mathbf{P}(S)$, since surely $\emptyset \subseteq S$. The *singletons* — sets with one member of S, namely, $\{1\}$, $\{2\}$, and $\{3\}$ — are in $\mathbf{P}(S)$. Likewise, the three sets with two of the three members of S are in $\mathbf{P}(S)$, and S itself is a member of $\mathbf{P}(S)$.

As another example, $\mathbf{P}(\emptyset) = \{\emptyset\}$ since $\emptyset \subseteq S$, but for no set S besides the empty set is $S \subseteq \emptyset$. Note that $\{\emptyset\}$, the set containing the empty set, is not the same as the empty set. In particular, the former has a member, namely $\emptyset$, while the empty set has no members. ✦

The Size of Power Sets

If S has n members, then $\mathbf{P}(S)$ has 2^n members. In Example 7.11 we saw that a set of three members has a power set of $2^3 = 8$ members. Also, $2^0 = 1$, and we saw that the empty set, which contains zero elements, has a power set of one element.

Let $S = \{a_1, a_2, \ldots, a_n\}$, where $a_1, a_2, \ldots, a_n$ are any n elements. We shall now prove by induction on n that $\mathbf{P}(S)$ has 2^n members.

BASIS. If $n = 0$, then S is $\emptyset$. We have already observed that $\mathbf{P}(\emptyset)$ has one member. Since $2^0 = 1$, we have proved the basis.

INDUCTION. Suppose that when $S = \{a_1, a_2, \ldots, a_n\}$, $\mathbf{P}(S)$ has 2^n members. Let a_{n+1} be a new element, and let $T = S \cup \{a_{n+1}\}$, a set of $n + 1$ members. Now a subset of T either does not have or does have a_{n+1} as a member. Let us consider these two cases in turn.

1. The subsets of T that do not include a_{n+1} are also subsets of S, and therefore in $\mathbf{P}(S)$. By the inductive hypothesis, there are exactly 2^n such sets.
2. If R is a subset of T that includes a_{n+1}, let $Q = R - \{a_{n+1}\}$; that is, Q is R with a_{n+1} removed. Then Q is a subset of S. By the inductive hypothesis, there are exactly 2^n possible sets Q, and each one corresponds to a unique set R, which is $Q \cup \{a_{n+1}\}$.

We conclude that there are exactly 2×2^n, or 2^{n+1}, subsets of T, half that are subsets of S, and half that are formed from a subset of S by including a_{n+1}. Thus, the inductive step is proved; given that any set S of n elements has 2^n subsets, we have shown that any set T of $n + 1$ elements has 2^{n+1} subsets.

EXERCISES

7.3.1: In Fig. 7.2, we showed two expressions for the set of regions $\{3, 5, 6\}$. However, each of the regions can be represented by expressions involving S, T, and R and the operators union, intersection, and difference. Write two different expressions for each of the following:

a) Region 6 alone
b) Regions 2 and 4 together
c) Regions 2, 4, and 8 together

7.3.2: Use Venn diagrams to show the following algebraic laws. For each subexpression involved in the equivalence, indicate the set of regions it represents.

a) $(S \cup (T \cap R)) \equiv ((S \cup T) \cap (S \cup R))$
b) $((S \cup T) - R) \equiv ((S - R) \cup (T - R))$
c) $(S - (T \cup R)) \equiv ((S - T) - R)$

7.3.3: Show each of the equivalences from Exercise 7.3.2 by showing containment of each side in the other.

7.3.4: Assuming $S \subseteq T$, prove the following by showing that each side of the equivalence is a subset of the other:

a) $(S \cap T) \equiv S$
b) $(S - T) \equiv \emptyset$

7.3.5*: Into how many regions does a Venn diagram with n sets divide the plane, assuming that no set is a subset of any other? Suppose that of the n sets there is one that is a subset of one other, but there are no other containments. Then some regions would be empty. For example, in Fig. 7.1, if $S \subseteq T$, then region 2 would be empty, because there is no element that is in S but not in T. In general, how many nonempty regions would there be?

7.3.6: Prove that if $S \subseteq T$, then $\mathbf{P}(S) \subseteq \mathbf{P}(T)$.

7.3.7*: In C we can represent a set S whose members are sets by a linked list whose elements are the headers for lists; each such list represents a set that is one of the members of S. Write a C program that takes a list of elements representing a set (i.e., a list in which all the elements are distinct) and returns the power set of the given set. What is the running time of your program? *Hint*: Use the inductive proof that there are 2^n members in the power set of a set of n elements to devise a recursive algorithm that creates the power set. If you are clever, you can use the same list as part of several sets, to avoid copying the lists that represent members of the power set, thus saving both time and space.

7.3.8: Show that

a) $\mathbf{P}(S) \cup \mathbf{P}(T) \subseteq \mathbf{P}(S \cup T)$
b) $\mathbf{P}(S \cap T) \subseteq \mathbf{P}(S) \cap \mathbf{P}(T)$

Are either (a) or (b) true if containment is replaced by equivalence?

7.3.9: What is $\mathbf{P}(\mathbf{P}(\mathbf{P}(\emptyset)))$?

7.3.10*: If we apply the power-set operator n times, starting with $\emptyset$, how many members does the resulting set have? For an example, Exercise 7.3.9 is the case $n = 3$.

7.4 List Implementation of Sets

We have already seen, in Section 6.4, how to implement the dictionary operations *insert*, *delete*, and *lookup* using a linked-list data structure. We also observed there that the expected running time of these operations is $O(n)$ if the set has n elements. This running time is not as good as the $O(\log n)$ average time taken for the dictionary operations using a balanced binary search tree data structure, as in Section 5.8. On the other hand, as we shall see in Section 7.6, a linked-list

representation of dictionaries plays an essential role in the hash-table data structure for dictionaries, which is generally faster than the binary search tree.

Union, Intersection, and Difference

The basic set operations such as union can profit from the use of linked lists as a data structure, although the proper techniques are somewhat different from what we use for the dictionary operations. In particular, sorting the lists significantly improvess the running time for union, intersection, and difference. As we saw in Section 6.4, sorting makes only a small improvement in the running time of dictionary operations.

To begin, let us see what problems arise when we represent sets by unsorted lists. In this case, we must compare each element of each set with each element of the other. Thus, to take the union, intersection, or difference of sets of size n and m requires $O(mn)$ time. For example, to create a list U that represents the union of two sets S and T, we may start by copying the list for S onto the initially empty list U. Then we examine each element of T and see whether it is in S. If not, we add the element to U. The idea is sketched in Fig. 7.5.

```
(1)  copy S to U;
(2)  for (each x in T)
(3)      if (!lookup(x, S))
(4)          insert(x, U);
```

Fig. 7.5. Pseudocode sketch of the algorithm for taking the union of sets represented by unsorted lists.

Suppose S has n members and T has m members. The operation in line (1), copying S to U, can easily be accomplished in $O(n)$ time. The lookup of line (3) takes $O(n)$ time. We only execute the insertion of line (4) if we know from line (3) that x is not in S. Since x can only appear once on the list for T, we know that x is not yet in U. Therefore, it is safe to place x at the front of U's list, and line (4) can be accomplished in $O(1)$ time. The for-loop of lines (2) through (4) is iterated m times, and its body takes time $O(n)$. Thus, the time for lines (2) to (4) is $O(mn)$, which dominates the $O(n)$ for line (1).

There are similar algorithms for intersection and difference, each taking $O(mn)$ time. We leave these algorithms for the reader to design.

Union, Intersection, and Difference Using Sorted Lists

We can perform unions, intersections, and set differences much faster when the lists representing the sets are sorted. In fact, we shall see that it pays to sort the lists before performing these operations, even if the lists are not initially sorted. For example, consider the computation of $S \cup T$, where S and T are represented by sorted lists. The process is similar to the merge algorithm of Section 2.8. One difference is that when there is a tie for smallest between the elements currently at the fronts of the two lists, we make only one copy of the element, rather than two copies as we must for merge. The other difference is that we cannot simply remove elements from the lists for S and T for the union, since we should not destroy S or

T while creating their union. Instead, we must make copies of all elements to form the union.

We assume that the types **LIST** and **CELL** are defined as before, by the macro

```
DefCell(int, CELL, LIST);
```

The function **setUnion** is shown in Fig. 7.6. It makes use of an auxiliary function *assemble*(x, L, M) that creates a new cell at line (1), places element x in that cell at line (2), and calls **setUnion** at line (3) to take the union of the lists L and M. Then **assemble** returns a cell for x followed by the list that results from applying **setUnion** to L and M. Note that the functions **assemble** and **setUnion** are mutually recursive; each calls the other.

Function **setUnion** selects the least element from its two given sorted lists and passes to **assemble** the chosen element and the remainders of the two lists. There are six cases for **setUnion**, depending on whether or not one of its lists is **NULL**, and if not, which of the two elements at the heads of the lists precedes the other.

1. If both lists are **NULL**, **setUnion** simply returns **NULL**, ending the recursion. This case is lines (5) and (6) of Fig. 7.6.
2. If L is **NULL** and M is not, then at lines (7) and (8) we assemble the union by taking the first element from M, followed by the "union" of the **NULL** list with the tail of M. Note that, in this case, successive calls to **setUnion** result in M being copied.
3. If M is **NULL** but L is not, then at lines (9) and (10) we do the opposite, assembling the answer from the first element of L and the tail of L.
4. If the first elements of L and M are the same, then at lines (11) and (12) we assemble the answer from one copy of this element, referred to as **L->element**, and the tails of L and M.
5. If the first element of L precedes that of M, then at lines (13) and (14) we assemble the answer from this smallest element, the tail of L, and the entire list M.
6. Symmetrically, at lines (15) and (16), if M has the smallest element, then we assemble the answer from that element, the entire list L, and the tail of M.

✦ **Example 7.12.** Suppose S is $\{1,3,6\}$ and T is $\{5,3\}$. The sorted lists representing these sets are $L = (1,3,6)$ and $M = (3,5)$. We call *setUnion*(L, M) to take the union. Since the first element of L, which is 1, precedes the first element of M, which is 3, case (5) applies, and we assemble the answer from 1, the tail of L, which we shall call $L_1 = (3,6)$, and M. Function *assemble*$(1, L_1, M)$ calls *setUnion*(L_1, M) at line (3), and the result is the list with first element 1 and tail equal to whatever the union is.

This call to **setUnion** is case (4), where the two leading elements are equal; both are 3 here. Thus, we assemble the union from one copy of element 3 and the tails of the lists L_1 and M. These tails are L_2, consisting of only the element 6, and M_1, consisting of only the element 5. The next call is *setUnion*(L_2, M_1), which is an instance of case (6). We thus add 5 to the union and call *setUnion*$(L_2,$ **NULL**$)$. That is case (3), generating 6 for the union and calling *setUnion*(**NULL**, **NULL**). Here, we

```
LIST setUnion(LIST L, LIST M);
LIST assemble(int x, LIST L, LIST M);

/* assemble produces a list whose head element is x and
   whose tail is the union of lists L and M */

LIST assemble(int x, LIST L, LIST M)
{
    LIST first;

(1)     first = (LIST) malloc(sizeof(struct CELL));
(2)     first->element = x;
(3)     first->next = setUnion(L, M);
(4)     return first;
}

/* setUnion returns a list that is the union of L and M */

LIST setUnion(LIST L, LIST M)
{
(5)     if (L == NULL && M == NULL)
(6)         return NULL;
(7)     else if (L == NULL) /* M cannot be NULL here */
(8)         return assemble(M->element, NULL, M->next);
(9)     else if (M == NULL) /* L cannot be NULL here */
(10)        return assemble(L->element, L->next, NULL);
        /* if we reach here, neither L nor M can be NULL */
(11)    else if (L->element == M->element)
(12)        return assemble(L->element, L->next, M->next);
(13)    else if (L->element < M->element)
(14)        return assemble(L->element, L->next, M);
(15)    else /* here, M->element < L->element */
(16)        return assemble(M->element, L, M->next);
}
```

Fig. 7.6. Computing the union of sets represented by sorted lists.

have case (1), and the recursion ends. The result of the initial call to `setUnion` is the list $(1, 3, 5, 6)$. Figure 7.7 shows in detail the sequence of calls and returns made on this example data. ✦

Notice that the list generated by `setUnion` always comes out in sorted order. We can see why the algorithm works, by observing that whichever case applies, each element in lists L or M is either copied to the output, by becoming the first parameter in a call to `assemble`, or remains on the lists that are passed as parameters in the recursive call to `setUnion`.

Running Time of Union

If we call `setUnion` on sets with n and m elements, respectively, then the time taken

```
call setUnion((1,3,6),(3,5))
    call assemble(1,(3,6),(3,5))
        call setUnion((3,6),(3,5))
            call assemble(3,(6),(5))
                call setUnion((6),(5))
                    call assemble(5,(6),NULL)
                        call setUnion((6),NULL)
                            call assemble(6,NULL,NULL)
                                call setUnion(NULL,NULL)
                                return NULL
                            return (6)
                        return (6)
                    return (5,6)
                return (5,6)
            return (3,5,6)
        return (3,5,6)
    return (1,3,5,6)
return (1,3,5,6)
```

Fig. 7.7. Sequence of calls and returns for Example 7.12.

Big-Oh for Functions of More Than One Variable

As we pointed out in Section 6.9, the notion of big-oh, which we defined only for functions of one variable, can be applied naturally to functions of more than one variable. We say that $f(x_1, \ldots, x_k)$ is $O(g(x_1, \ldots, x_k))$ if there are constants c and $a_1, \ldots, a_k$ such that whenever $x_i \geq a_i$ for all $i = 1, \ldots, k$, it is the case that $f(x_1, \ldots, x_k) \leq cg(x_1, \ldots, x_k)$. In particular, note that even though $m+n$ is greater than mn when one of m and n is 0 and the other is greater than 0, we can still say that $m + n$ is $O(mn)$, by choosing constants c, a_1, and a_2 all equal to 1.

by **setUnion** is $O(m+n)$. To see why, note that calls to **assemble** spend $O(1)$ time creating a cell for the output list and then calling **setUnion** on the remaining lists. Thus, the calls to **assemble** in Fig. 7.6 can be thought of as costing $O(1)$ time plus the time for a call to **setUnion** on lists the sum of whose lengths is either one less than that of L and M, or in case (4), two less. Further, all the work in **setUnion**, exclusive of the call to **assemble**, takes $O(1)$ time.

It follows that when **setUnion** is called on lists of total length $m+n$, it will result in at most $m + n$ recursive calls to **setUnion** and an equal number to **assemble**. Each takes $O(1)$ time, exclusive of the time taken by the recursive call. Thus, the time to take the union is $O(m + n)$, that is, proportional to the sum of the sizes of the sets.

This time is less than that of the $O(mn)$ time needed to take the union of sets represented by unsorted lists. In fact, if the lists for our sets are not sorted, we can sort them in $O(n \log n + m \log m)$ time, and then take the union of the sorted lists. Since $n \log n$ dominates n and $m \log m$ dominates m, we can express the total

cost of sorting and taking the union as $O(n \log n + m \log m)$. That expression can be greater than $O(mn)$, but is less whenever n and m are close in value — that is, whenever the sets are approximately the same size. Thus, it usually makes sense to sort before taking the union.

Intersection and Difference

The idea in the algorithm for union outlined in Fig. 7.6 works for intersections and differences of sets as well: when the sets are represented by sorted lists, intersection and difference are also performed in linear time. For intersection, we want to copy an element to the output only if it appears on both lists, as in case (4). If either list is `NULL`, we cannot have any elements in the intersection, and so cases (1), (2), and (3) can be replaced by a step that returns `NULL`. In case (4), we copy the element at the heads of the lists to the intersection. In cases (5) and (6), where the heads of the lists are different, the smaller cannot appear on both lists, and so we do not add anything to the intersection but pop the smaller off its list and take the intersection of the remainders.

To see why that makes sense, suppose, for example, that a is at the head of list L, that b is at the head of list M, and that $a < b$. Then a cannot appear on the sorted list M, and so we can rule out the possibility that a is on both lists. However, b can appear on list L somewhere after a, so that we may still be able to use b from M. Thus, we need to take the intersection of the tail of L with the entire list M. Conversely, if b were less than a, we would take the intersection of L with the tail of M. C code to compute the intersection is shown in Fig. 7.8. It is also necessary to modify `assemble` to call `intersection` instead of `setUnion`. We leave this change as well as a program to compute the difference of sorted lists as exercises.

```
LIST intersection(LIST L, LIST M)
{
    if (L == NULL || M == NULL)
        return NULL;
    else if (L->element == M->element)
        return assemble(L->element, L->next, M->next);
    else if (L->element < M->element)
        return intersection(L->next, M);
    else /* here, M->element < L->element */
        return intersection(L, M->next);
}
```

Fig. 7.8. Computing the intersection of sets represented by sorted lists. A new version of `assemble` is required.

EXERCISES

7.4.1: Write C programs for taking the (a) union, (b) intersection, and (c) difference of sets represented by unsorted lists.

7.4.2: Modify the program of Fig. 7.6 so that it takes the (a) intersection and (b) difference of sets represented by sorted lists.

7.4.3: The functions **assemble** and **setUnion** from Fig. 7.6 leave the lists whose union they take intact; that is, they make copies of elements rather than use the cells of the given lists themselves. Can you simplify the program by allowing it to destroy the given lists as it takes their union?

7.4.4*: Prove by induction on the sum of the lengths of the lists given as parameters that **setUnion** from Fig. 7.6 returns the union of the given lists.

Symmetric difference

7.4.5*: The *symmetric difference* of two sets S and T is $(S - T) \cup (T - S)$, that is, the elements that are in exactly one of S and T. Write a program to take the symmetric difference of two sets that are represented by sorted lists. Your program should make one pass through the lists, like Fig. 7.6, rather than call routines for union and difference.

7.4.6*: We analyzed the program of Fig. 7.6 informally by arguing that if the total of the lengths of the lists was n, there were $O(n)$ calls to **setUnion** and **assemble** and each call took $O(1)$ time plus whatever time the recursive call took. We can formalize this argument by letting $T_U(n)$ be the running time for **setUnion** and $T_A(n)$ be the running time of **assemble** on lists of total length n. Write recursive rules defining T_U and T_A in terms of each other. Substitute to eliminate T_A, and set up a conventional recurrence for T_U. Solve that recurrence. Does it show that **setUnion** takes $O(n)$ time?

7.5 Characteristic-Vector Implementation of Sets

Frequently, the sets we encounter are each subsets of some small set U, which we shall refer to as the "universal set."[1] For example, a hand of cards is a subset of the set of all 52 cards. When the sets with which we are concerned are each subsets of some small set U, there is a representation of sets that is much more efficient than the list representation discussed in the previous section. We order the elements of U in some way so that each element of U can be associated with a unique "position," which is an integer from 0 up to $n - 1$, where n is the number of elements in U.

Then, given a set S that is contained in U, we can represent S by a *characteristic vector* of 0's and 1's, such that for each element x of U, if x is in S, the position corresponding to x has a 1, and if x is not in S, then that position has a 0.

✦ **Example 7.13.** Let U be the set of cards. We may order the cards any way we choose, but one reasonable scheme is to order them by suits: clubs, then diamonds, then hearts, then spades. Then, within a suit, we order the cards ace, 2, 3, ..., 10, jack, queen, king. For instance, the position of the ace of clubs is 0, the king of clubs is 12, the ace of diamonds is 13, and the jack of spades is 49. A royal flush in hearts is represented by the characteristic vector

0000000000000000000000000001000000001111000000000000

[1] Of course U cannot be a true universal set, or set of all sets, which we argued does not exist because of Russell's paradox.

The first 1, in position 26, represents the ace of hearts; and the other four 1's, in positions 35 through 38, represent the 10, jack, queen, and king of hearts.

The set of all clubs is represented by

11111111111110000000000000000000000000000000000000000

and the set of all picture cards is represented by

00000000001110000000001110000000001110000000000111

✦

Array Implementation of Sets

We can represent characteristic vectors for subsets of a universal set of n elements using Boolean arrays of the following type:

```
typedef BOOLEAN USET[n];
```

The type **BOOLEAN** is as described in Section 1.6. To insert the element corresponding to position i into a set S declared to be of type **USET**, we have only to execute

```
S[i] = TRUE;
```

Similarly, to delete the element corresponding to position i from S, we do

```
S[i] = FALSE;
```

If we want to look up this element, we have only to return the value $S[i]$, which tells us whether the ith element is present in S or not.

Note that each of the dictionary operations *insert*, *delete*, and *lookup* thus takes $O(1)$ time, when sets are represented by characteristic vectors. The only disadvantage to this technique is that all sets must be subsets of some universal set U. Moreover, the universal set must be small; otherwise, the length of the arrays becomes so large that we cannot store them conveniently. In fact, since we shall normally have to initialize all elements of the array for a set S to **TRUE** or **FALSE**, the initialization of any subset of U (even $\emptyset$) must take time proportional to the size of U. If U had a large number of elements, the time to initialize a set could dominate the cost of all other operations.

To form the union of two sets that are subsets of a common universal set of n elements, and that are represented by characteristic vectors S and T, we define another characteristic vector R to be the bitwise **OR** of the characteristic vectors S and T:

```
R[i] = S[i] || T[i], for 0 ≤ i < n
```

Similarly, we can make R represent the intersection of S and T by taking the bitwise **AND** of S and T:

```
R[i] = S[i] && T[i], for 0 ≤ i < n
```

Finally, we can make R represent the set difference $S - T$ as follows:

```
R[i] = S[i] && !T[i], for 0 ≤ i < n
```

The arrays representing characteristic vectors and the Boolean operations on them can be implemented using the bitwise operators of C if we define the type **BOOLEAN** appropriately. However, the code is machine specific, and so we shall not present any details here. A portable (but more space consuming) implementation of characteristic vectors can be accomplished with arrays of **int**'s of the appropriate size, and this is the definition of **BOOLEAN** that we have assumed.

✦ **Example 7.14.** Let us consider sets of apple varieties. Our universal set will consist of the six varieties listed in Fig. 7.9; the order of their listing indicates their position in characteristic vectors.

	VARIETY	COLOR	RIPENS
0)	Delicious	red	late
1)	Granny Smith	green	early
2)	Gravenstein	red	early
3)	Jonathan	red	early
4)	McIntosh	red	late
5)	Pippin	green	late

Fig. 7.9. Characteristics of some apple varieties.

The set of red apples is represented by the characteristic vector

$$Red = 101110$$

and the set of early apples is represented by

$$Early = 011100$$

Thus, the set of apples that are either red or early, that is, $Red \cup Early$, is represented by the characteristic vector 111110. Note that this vector has a 1 in those positions where either the vector for *Red*, that is, 101110, or the vector for *Early*, that is, 011100, or both, have a 1.

We can find the characteristic vector for $Red \cap Early$, the set of apples that are both red and early, by placing a 1 in those positions where both 101110 and 011100 have 1. The resulting vector is 001100, representing the set of apples {Gravenstein, Jonathan}. The set of apples that are red but not early, that is,

$$Red - Early$$

is represented by the vector 100010. The set is {Delicious, McIntosh}. ✦

Notice that the time to perform union, intersection, and difference using characteristic vectors is proportional to the length of the vectors. That length is not directly related to the size of the sets, but is equal to the size of the universal set. If the sets have a reasonable fraction of the elements in the universal set, then the time for union, intersection, and difference is proportional to the sizes of the sets involved. That is better than the $O(n \log n)$ time for sorted lists, and much better than the $O(n^2)$ time for unsorted lists. However, the drawback of characteristic

vectors is that, should the sets be much smaller than the universal set, the running time of these operations can be far greater than the sizes of the sets involved.

EXERCISES

7.5.1: Give the characteristic vectors of the following sets of cards. For convenience, you can use 0^k to represent k consecutive 0's and 1^k for k consecutive 1's.

a) The cards found in a pinochle deck

b) The red cards

c) The one-eyed jacks and the suicide king

7.5.2: Using bitwise operators, write C programs to compute the (a) union and (b) difference of two sets of cards, the first represented by words $a1$ and $a2$, the second represented by $b1$ and $b2$.

7.5.3*: Suppose we wanted to represent a bag (multiset) whose elements were contained in some small universal set U. How could we generalize the characteristic-vector method of representation to bags? Show how to perform (a) *insert*, (b) *delete*, and (c) *lookup* on bags represented this way. Note that bag *lookup*(x) returns the number of times x appears in the bag.

7.6 Hashing

The characteristic-vector representation of dictionaries, when it can be used, allows us to access directly the place where an element is represented, that is, to access the position in the array that is indexed by the value of the element. However, as we mentioned, we cannot allow our universal set to be too large, or the array will be too long to fit in the available memory of the computer. Even if it did fit, the time to initialize the array would be prohibitive. For example, suppose we wanted to store a real dictionary of words in the English language, and also suppose we were willing to ignore words longer than 10 letters. We would still have $26^{10} + 26^9 + \cdots + 26$ possible words, or over 10^{14} words. Each of these possible words would require a position in the array.

At any time, however, there are only about a million words in the English language, so that only one out of 100 million of the array entries would be **TRUE**. Perhaps we could collapse the array, so that many possible words could share an entry. For example, suppose we assigned the first 100 million possible words to the first cell of the array, the next 100 million possibilities to the second cell, and so on, up to the millionth cell. There are two problems with this arrangement:

1. It is no longer enough just to put **TRUE** in a cell, because we won't know which of the 100 million possible words are actually present in the dictionary, or if in fact more than one word in any one group is present.

2. If, for example, the first 100 million possible words include all the short words, then we would expect many more than the average number of words from the dictionary to fall into this group of possible words. Note that our arrangement has as many cells of the array as there are words in the dictionary, and so we expect the average cell to represent one word; but surely there are in English many thousands of words in the first group, which would include all the words of up to five letters, and some of the six-letter words.

To solve problem (1), we need to list, in each cell of the array, all the words in its group that are present in the dictionary. That is, the array cell becomes the header of a linked list with these words. To solve problem (2), we need to be careful how we assign potential words to groups. We must distribute elements among groups so that it is unlikely (although never impossible) that there will be many elements in a single group. Note that if there are a large number of elements in a group, and we represent groups by linked lists, then lookup will be very slow for members of a large group.

The Hash Table Data Structure

We have now evolved from the characteristic vector — a valuable data structure that is of limited use — to a structure called a *hash table* that is useful for any dictionary whatsoever, and for many other purposes as well.[2] The speed of the hash table for the dictionary operations can be made $O(1)$ on the average, independent of the size of the dictionary, and independent of the size of the universal set from which the dictionary is drawn. A picture of a hash table appears in Fig. 7.10. However, we show the list for only one group, that to which x belongs.

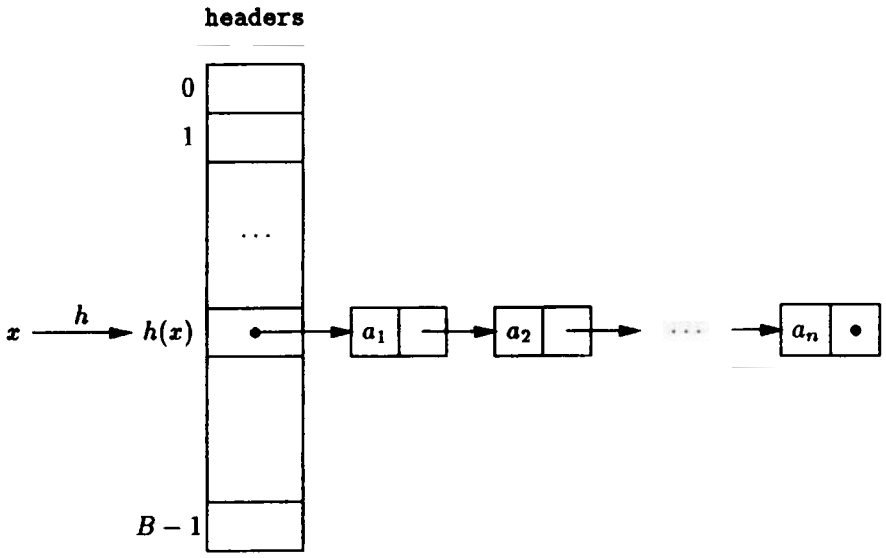


Fig. 7.10. A hash table.

[2] Although in situations where a characteristic vector is feasible, we would normally prefer that representation over any other.

Bucket

There is a *hash function* that takes an element x as argument and produces an integer value between 0 and $B-1$, where B is the number of *buckets* in the hash table. The value $h(x)$ is the bucket in which we place the element x. Thus, the buckets correspond to the "groups" of words that we talked about in the preceding informal discussion, and the hash function is used to decide to which bucket a given element belongs.

Hash function

The appropriate hash function to use depends on the type of the elements. For example,

1. If elements are integers, we could let $h(x)$ be x % B, that is, the remainder when x is divided by B. That number is always in the required range, 0 to $B-1$.

2. If the elements are character strings, we can take an element $x = a_1a_2\cdots a_k$, where each a_i is a character, and compute $y = a_1 + a_2 + \cdots + a_k$, since a `char` in C is a small integer. We then have an integer y that is the sum of the integer equivalents of all the characters in the string x. If we divide y by B and take the remainder, we have a bucket number in the range 0 to $B-1$.

What is important is that the hash function "hashes" the element. That is, h wildly mixes up the buckets into which elements fall, so they tend to fall in approximately equal numbers into all the buckets. This equitable distribution must occur even for a fairly regular pattern of elements, such as consecutive integers or character strings that differ in only one position.

Each bucket consists of a linked list wherein are stored all the elements of the set that are sent by the hash function to that bucket. To find an element x, we compute $h(x)$, which gives us a bucket number. If x is anywhere, it is in that bucket, so that we may search for x by running down the list for that bucket. In effect, the hash table allows us to use the (slow) list representation for sets, but, by dividing the set into B buckets, allows us to search lists that are only $1/B$ as long as the size of the set, on the average. If we make B roughly as large as the set, then buckets will average only one element, and we can find elements in an average of $O(1)$ time, just as for the characteristic-vector representation of sets.

✦ **Example 7.15.** Suppose we wish to store a set of character strings of up to 32 characters, where each string is terminated by the null character. We shall use the hash function outlined in (2) above, with $B = 5$, that is, a five-bucket hash table. To compute the hash value of each element, we sum the integer values of the characters in each string, up to but not including the null character. The following declarations give us the desired types.

```
(1)    #define B 5
(2)    typedef char ETYPE[32];
(3)    DefCell(ETYPE, CELL, LIST);
(4)    typedef LIST HASHTABLE[B];
```

Line (1) defines the constant B to be the number of buckets, 5. Line (2) defines the type `ETYPE` to be arrays of 32 characters. Line (3) is our usual definition of cells and linked lists, but here the element type is `ETYPE`, that is, 32-character arrays. Line (4) defines a hashtable to be an array of B lists. If we then declare

HASHTABLE headers

the array **headers** is of the appropriate type to contain the bucket headers for our hash table.

```
int h(ETYPE x)
{
    int i, sum;

    sum = 0;
    for (i = 0; x[i]    '\0'; i++)
        sum += x[i];
    return sum % B;
}
```

Fig. 7.11. A hash function that sums the integer equivalents of characters, assuming ETYPE is an array of characters.

Now, we must define the hash function h. The code for this function is shown in Fig. 7.11. The integer equivalent of each of the characters of the string x is summed in the variable **sum**. The last step computes and returns as the value of the hash function h the remainder of this sum when it is divided by the number of buckets B.

Let us consider some examples of words and the buckets into which the function h puts them. We shall enter into the hash table the seven words[3]

anyone lived in a pretty how town

In order to compute h(**anyone**), we need to understand the integer values of characters. In the usual ASCII code for characters, the lower-case letters have integer values starting at 97 for **a** (that's 1100001 in binary), 98 for **b**, and so on, up to 122 for **z**. The upper-case letters correspond to integers that are 32 less than their lower-case equivalents — that is, from 65 for **A** (1000001 in binary) to 90 for **Z**.

Thus, the integer equivalents for the characters in **anyone** are 97, 110, 121, 111, 110, 101. The sum of these is 650. When we divide by B, which is 5, we get the remainder 0. Thus, **anyone** belongs in bucket 0. The seven words of our example are assigned, by the hash function of Fig. 7.11, to the buckets indicated in Fig. 7.12.

We see that three of the seven words have been assigned to one bucket, number 0. Two words are assigned to bucket 2, and one each to buckets 1 and 4. That is somewhat less even a distribution than would be typical, but with a small number of words and buckets, we should expect anomalies. As the number of words becomes large, they will tend to distribute themselves among the five buckets approximately evenly. The hash table, after insertion of these seven words, is shown in Fig. 7.13. ✦

Implementing the Dictionary Operations by a Hash Table

To insert, delete, or look up an element x in a dictionary that is represented by a hash table, there is a simple three-step process:

---

[3] The words are from a poem of the same name by e. e. cummings. The poem doesn't get any easier to decode. The next line is "with up so floating many bells down."

WORD	SUM	BUCKET
anyone	650	0
lived	532	2
in	215	0
a	97	2
pretty	680	0
how	334	4
town	456	1

Fig. 7.12. Words, their values, and their buckets.

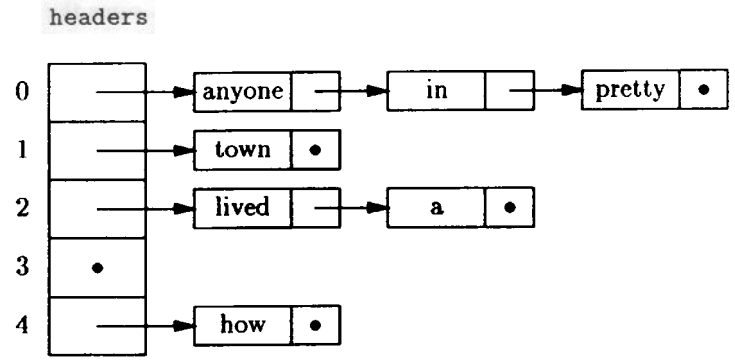


Fig. 7.13. Hash table holding seven elements

1. Compute the proper bucket, which is $h(x)$.
2. Use the array of header pointers to find the list of elements for the bucket numbered $h(x)$.
3. Perform the operation on this list, just as if the list represented the entire set.

The algorithms in Section 6.4 can be used for the list operations after suitable modifications for the fact that elements here are character strings while in Section 6.4 elements were integers. As an example, we show the complete function for inserting an element into a hash table in Fig. 7.14. You can develop similar functions for **delete** and **lookup** as an exercise.

To understand Fig. 7.14, it helps to notice that the function **bucketInsert** is similar to the function **insert** from Fig. 6.5. At line (1) we test to see whether we have reached the end of the list. If we have, then we create a new cell at line (2). However, at line (3), instead of storing an integer into the newly created cell, we use the function **strcpy** from the standard header file **string.h** to copy the string x into the element field of the cell.

Also, at line (5), to test if x has not yet been found on the list, we use function **strcmp** from **string.h**. That function returns 0 if and only if x and the element in the current cell are equal. Thus, we continue down the list as long as the value of the comparison is nonzero, that is, as long as the current element is not x.

The function **insert** here consists of a single line, in which we call **buck-**

```
    #include <string.h>

    void bucketInsert(ETYPE x, LIST *pL)
    {
(1)     if ((*pL) == NULL) {
(2)         (*pL) = (LIST) malloc(sizeof(struct CELL));
(3)         strcpy((*pL)->element, x);
(4)         (*pL)->next = NULL;
        }
(5)     else if (strcmp((*pL)->element, x)) /* x and element
                    are different */
(6)         bucketInsert(x, &((*pL)->next));
    }

    void insert(ETYPE x, HASHTABLE H)
    {
(7)     bucketInsert(x, &(H[h(x)]));
    }
```

Fig. 7.14. Inserting an element into a hash table.

`etInsert` after first finding the element of the array that is the header for the appropriate bucket, $h(x)$. We assume that the hash function h is defined elsewhere. Also recall that the type `HASHTABLE` means that `H` is an array of pointers to cells (i.e., an array of lists).

✦ **Example 7.16.** Suppose we wish to delete the element `in` from the hash table of Fig. 7.13, assuming the hash function described in Example 7.15. The delete operation is carried out essentially like the function `insert` of Fig. 7.14. We compute h(`in`), which is 0. We thus go to the header for bucket number 0. The second cell on the list for this bucket holds `in`, and we delete that cell. The detailed C program is left as an exercise. ✦

Running Time of Hash Table Operations

As we can see by examining Fig. 7.14, the time taken by the function *insert* to find the header of the appropriate bucket is $O(1)$, assuming that the time to compute $h(x)$ is a constant independent of the number of elements stored in the hash table.[4] To this constant we must add on the average an additional $O(n/B)$ time, if n is the number of elements in the hash table and B is the number of buckets. The reason is that *bucketInsert* will take time proportional to the length of the list, and that length, on the average, must be the total number of elements divided by the number of buckets, or n/B.

An interesting consequence is that if we make B approximately equal to the number of elements in the set — that is, n and B are close — then n/B is about 1

[4] That would be the case for the hash function of Fig. 7.11, or most other hash functions encountered in practice. The time for computing the bucket number may depend on the type of the element — longer strings may require the summation of more integers, for example — but the time is not dependent on the number of elements stored.

and the dictionary operations on a hash table take $O(1)$ time each, on the average, just as when we use a characteristic-vector representation. If we try to do better by making B much larger than n, so that most buckets are empty, it still takes us $O(1)$ time to find the bucket header, and so the running time does not improve significantly once B becomes larger than n.

Restructuring hash tables

We must also consider that in some circumstances it may not be possible to keep B close to n all the time. If the set is growing rapidly, then n increases while B remains fixed, so that ultimately n/B becomes large. It is possible to restructure the hash table by picking a larger value for B and then inserting each of the elements into the new table. It takes $O(n)$ time to do so, but that time is no greater than the $O(n)$ time that must be spent inserting the n elements into the hash table in the first place. (Note that n insertions, at $O(1)$ average time per insertion, require $O(n)$ time in all.)

EXERCISES

7.6.1: Continue filling the hash table of Fig. 7.13 with the words `with up so floating many bells down`.

7.6.2*: Comment on how effective the following hash functions would be at dividing typical sets of English words into buckets of roughly equal size:

a) Use $B = 10$, and let $h(x)$ be the remainder when the length of the word x is divided by 10.

b) Use $B = 128$, and let $h(x)$ be the integer value of the last character of x.

c) Use $B = 10$. Take the sum of the values of the characters in x. Square the result, and take the remainder when divided by 10.

7.6.3: Write C programs for performing (a) *delete* and (b) *lookup* in a hash table, using the same assumptions as for the code in Fig. 7.14.

7.7 Relations and Functions

While we have generally assumed that elements of sets are atomic, in practice it is often useful to give elements some structure. For example, in the previous section we talked about elements that were character strings of length 32. Another important structure for elements is fixed-length lists, which are similar to C structures. Lists used as set elements will be called *tuples*, and each list element is called a *component* of the tuple.

Tuple, component

The number of components a tuple has is called its *arity*. For example, (a, b) is a tuple of arity 2; its first component is a and its second component is b. A tuple of arity k is also called a *k-tuple*.

Arity: unary, binary

A set of elements, each of which is a tuple of the same arity, — say, k — is called a *relation*. The arity of this relation is k. A tuple or relation of arity 1 is *unary*. If the arity is 2, it is *binary*, and in general, if the arity is k, then the tuple or relation is *k-ary*.

✦ **Example 7.17.** The relation $R = \{(1,2), (1,3), (2,2)\}$ is a relation of arity 2, or a binary relation. Its members are $(1,2)$, $(1,3)$, and $(2,2)$, each of which is a tuple of arity 2. ✦

In this section, we shall consider primarily binary relations. There are also many important applications of nonbinary relations, especially in representing and manipulating tabular data (as in relational databases). We shall discuss this topic extensively in Chapter 8.

Cartesian Products

Before studying binary relations formally, we need to define another operator on sets. Let A and B be two sets. Then the *product* of A and B, denoted $A \times B$, is defined as the set of pairs in which the first component is chosen from A and the second component from B. That is,

$$A \times B = \{(a,b) \mid a \in A \text{ and } b \in B\}$$

The product is sometimes called the *Cartesian* product, after the French mathematician René Descartes.

✦ **Example 7.18.** Recall that $\mathbf{Z}$ is the conventional symbol for the set of all integers. Thus, $\mathbf{Z} \times \mathbf{Z}$ stands for the set of pairs of integers.

As another example, if A is the two-element set $\{1,2\}$ and B is the three-element set $\{a,b,c\}$, then $A \times B$ is the six-element set

$$\{(1,a), (1,b), (1,c), (2,a), (2,b), (2,c)\}$$

Note that the product of sets is aptly named, because if A and B are finite sets, then the number of elements in $A \times B$ is the product of the number of elements in A and the number of elements in B. ✦

Cartesian Product of More Than Two Sets

Unlike the arithmetic product, the Cartesian product does not have the common properties of commutativity or associativity. It is easy to find examples where

$$A \times B \neq B \times A$$

disproving commutativity. The associative law does not even make sense, because $(A \times B) \times C$ would have as members pairs like $\big((a,b),c\big)$, while members of $A \times (B \times C)$ would be pairs of the form $\big(a,(b,c)\big)$.

k-way product of sets

Since we shall need on several occasions to talk about sets of tuples with more than two components, we need to extend the product notation to a k-way product. We let $A_1 \times A_2 \times \cdots \times A_k$ stand for the *product* of sets $A_1, A_2, \ldots, A_k$, that is, the set of k-tuples $(a_1, a_2, \ldots, a_k)$ such that $a_1 \in A_1$, $a_2 \in A_2, \ldots$, and $a_k \in A_k$.

✦ **Example 7.19.** $\mathbf{Z} \times \mathbf{Z} \times \mathbf{Z}$ represents the set of triples of integers (i, j, k) — it contains, for example, the triple $(1, 2, 3)$. This three-way product should not be confused with $(\mathbf{Z} \times \mathbf{Z}) \times \mathbf{Z}$, which represents pairs like $((1, 2), 3)$, or $\mathbf{Z} \times (\mathbf{Z} \times \mathbf{Z})$, which represents pairs like $(1, (2, 3))$.

On the other hand, note that all three product expressions can be represented by structures consisting of three integer fields. The distinction is in how one interprets the structures of this type. Thus, we often feel free to "confuse" parenthesized and unparenthesized product expressions. Similarly, the three C type declarations

```
struct {int f1; int f2; int f3;};
struct {struct {int f1; int f2;}; int f3;};
struct {int f1; struct {int f2; int f3;};};
```

would all be stored in a similar way — only the notation for accessing fields would differ. ✦

Binary Relations

A binary relation R is a set of pairs that is a subset of the product of two sets A and B. If a relation R is a subset of $A \times B$, we say that R is *from* A *to* B. We call A the *domain* and B the *range* of the relation. If B is the same as A, we say that R is a relation *on* A or "on the domain" A.

Domain, range

✦ **Example 7.20.** The arithmetic relation $<$ on integers is a subset of $\mathbf{Z} \times \mathbf{Z}$, consisting of those pairs (a, b) such that a is less than b. Thus, the symbol $<$ may be regarded as the name of the set

$$\{(a, b) \mid (a, b) \in \mathbf{Z} \times \mathbf{Z}, \text{ and } a \text{ is less than } b\}$$

We then use $a < b$ as a shorthand for "$(a, b) \in <$," or "(a, b) is a member of the relation $<$." The other arithmetic relations on integers, such as $>$ or $\leq$, can be defined similarly, as can the arithmetic comparisons on real numbers.

For another example, consider the relation R from Example 7.17. Its domain and range are uncertain. We know that 1 and 2 must be in the domain, because these integers appear as first components of tuples in R. Similarly, we know that the range of R must include 2 and 3. However, we could regard R as a relation from $\{1, 2\}$ to $\{2, 3\}$, or as a relation from $\mathbf{Z}$ to $\mathbf{Z}$, as two examples among an infinity of choices. ✦

Infix Notation for Relations

As we suggested in Example 7.20, it is common to use an infix notation for binary relations, so that a relation like $<$, which is really a set of pairs, can be written between the components of pairs in the relation. That is why we commonly find expressions like $1 < 2$ and $4 \geq 4$, rather than the more pedantic "$(1, 2) \in <$" or "$(4, 4) \in \geq$."

✦ **Example 7.21.** The same notation can be used for arbitrary binary relations. For instance, the relation R from Example 7.17 can be written as the three "facts" $1R2$, $1R3$, and $2R2$. ✦

Declared and Current Domains and Ranges

The second part of Example 7.20 underscores the point that we cannot tell the domain or range of a relation just by looking at it. Surely the set of elements appearing in first components must be a subset of the domain, and the set of elements that occur in second components must be a subset of the range. However, there could be other elements in the domain or range.

The difference is not important when a relation does not change. However, we shall see in Sections 7.8 and 7.9, and also in Chapter 8, that relations whose values change are very important. For example, we might speak of a relation whose domain is the students in a class, and whose range is integers, representing total scores on homework. Before the class starts, there are no pairs in this relation. After the first assignment is graded, there is one pair for each student. As time goes on, students drop the class or are added to the class, and scores increase.

We could define the domain of this relation to be the set of all students registered at the university and the range to be the set of integers. Surely, at any time, the value of the relation is a subset of the Cartesian product of these two sets. On the other hand, at any time, the relation has a *current domain* and a *current range*, which are the sets of elements appearing in first and second components, respectively, of the pairs in the relation. When we need to make a distinction, we can call the domain and range the *declared* domain and range. The current domain and range will always be a subset of the declared domain and range, respectively.

Current domain, range

Declared domain, range

Graphs for Binary Relations

We can represent a relation R whose domain is A and whose range is B by a graph. We draw a node for each element that is in A and/or B. If aRb, then we draw an arrow ("arc") from a to b. (General graphs are discussed in more detail in Chapter 9.)

✦ **Example 7.22.** The graph for the relation R from Example 7.17 is shown in Fig. 7.15. It has nodes for the elements 1, 2, and 3. Since $1R2$, there is an arc from node 1 to node 2. Since $1R3$, there is an arc from 1 to 3, and since $2R2$, there is an arc from node 2 to itself. There are no other arcs, because there are no other pairs in R. ✦

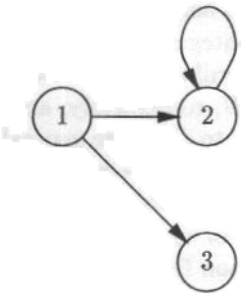


Fig. 7.15. Graph for the relation $\{(1,2), (1,3), (2,2)\}$.

Functions

Suppose a relation R, from domain A to range B, has the property that for every member a of A there is at most one element b in B such that aRb. Then R is said to be a *partial function from domain A to range B.*

Partial function

If for every member a of A there is exactly one element b in B such that aRb, then R is said to be a *total function* from A to B. The difference between a partial function and a total function is that a partial function can be undefined on some elements of its domain; for example, for some a in A, there may be no b in B such that aRb. We shall use the term "function" to refer to the more general notion of a partial function, but whenever the distinction between a partial function and a total function is important, we shall use the word "partial."

Total function

There a common notation used to describe functions. We often write $R(a) = b$ if b is the unique element such that aRb.

✦ **Example 7.23.** Let S be the total function from $\mathbf{Z}$ to $\mathbf{Z}$ given by

$$\{(a, b) \mid b = a^2\}$$

that is, the set of pairs of integers whose second component is the square of the first component. Then S has such members as $(3, 9)$, $(-4, 16)$, and $(0, 0)$. We can express the fact that S is the squaring function by writing $S(3) = 9$, $S(-4) = 16$, and $S(0) = 0$. ✦

Notice that the set-theoretic notion of a function is not much different from the notion of a function that we encountered in C. That is, suppose `s` is a C function declared as

```
int s(int a)
{
    return a*a;
}
```

that takes an integer and returns its square. We usually think of `s(a)` as being the same function as $S(a)$, although the former is a way to compute squares and the latter only defines the operation of squaring abstractly. Also note that in practice `s(a)` is always a partial function, since there are many values of `a` for which `s(a)` will not return an integer because of the finiteness of computer arithmetic.

C has functions that take more than one parameter. A C function `f` that takes two integer parameters `a` and `b`, returning an integer, is a function from $\mathbf{Z} \times \mathbf{Z}$ to $\mathbf{Z}$. Similarly, if the two parameters are of types that make them belong to sets A and B, respectively, and `f` returns a member of type C, then `f` is a function from $A \times B$ to C. More generally, if `f` takes k parameters — say, from sets $A_1, A_2, \ldots, A_k$, respectively — and returns a member of set B, then we say that `f` is a function from $A_1 \times A_2 \times \cdots \times A_k$ to B.

For example, we can regard the function `lookup(x,L)` from Section 6.4 as a function from $\mathbf{Z} \times L$ to {`TRUE`, `FALSE`}. Here, L is the set of linked lists of integers.

Formally, a function from domain $A_1 \times \cdots \times A_k$ to range B is a set of pairs of the form $\big((a_1, \ldots, a_k), b\big)$, where each a_i is in set A_i and b is in B. Notice that the first element of the pair is itself a k-tuple. For example, the function `lookup(x,L)` discussed above can be thought of as the set of pairs $\big((x, L), t\big)$, where x is an

The Many Notations for Functions

A function F from, say, $A \times B$ to C is technically a subset of $(A \times B) \times C$. A typical pair in the function F would thus have the form $((a, b), c)$, where a, b, and c are members of A, B, and C, respectively. Using the special notation for functions, we can write $F(a, b) = c$.

We can also view F as a relation from $A \times B$ to C, since every function is a relation. Using the infix notation for relations, the fact that $((a, b), c)$ is in F could also be written $(a, b)Fc$.

When we extend the Cartesian product to more than two sets, we may wish to remove parentheses from product expressions. Thus, we might identify $(A \times B) \times C$ with the technically inequivalent expression $A \times B \times C$. In that case, a typical member of F could be written (a, b, c). If we stored F as a set of such triples, we would have to remember that the first two components together make up the domain element and the third component is the range element.

integer, L is a list of integers, and t is either **TRUE** or **FALSE**, depending on whether x is or is not on the list L. We can think of a function, whether written in C or as formally defined in set theory, as a box that takes a value from the domain set and produces a value from the range set, as suggested in Fig. 7.16 for the function **lookup**.

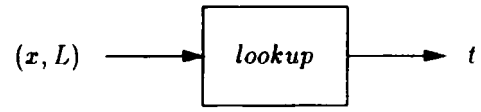


Fig. 7.16. A function associates elements from the domain with unique elements from the range.

One-to-One Correspondences

Let F be a partial function from domain A to range B with the following properties:

1. For every element a in A, there is an element b in B such that $F(a) = b$.
2. For every b in B, there is some a in A such that $F(a) = b$.
3. For no b in B are there two elements a_1 and a_2 in A such that $F(a_1)$ and $F(a_2)$ are both b.

Then F is said to be a *one-to-one correspondence* from A to B. The term *bijection* is also used for a one-to-one correspondence.

Property (1) says that F is a total function from A to B. Property (2) is the condition of being *onto*: F is a total function from A onto B. Some mathematicians use the term *surjection* for a total function that is onto.

Surjection

Properties (2) and (3) together say that F behaves like a total function from B to A. A total function with property (3) is sometimes called an *injection*.

Injection

A one-to-one correspondence is basically a total function in both directions, but it is important to observe that whether F is a one-to-one correspondence depends not only on the pairs in F, but on the declared domain and range. For example, we could take any one-to-one correspondence from A to B and change the domain by

adding to A some new element e not mentioned in F. F would not be a one-to-one correspondence from $A \cup \{e\}$ to B.

✦ **Example 7.24.** The squaring function S from $\mathbf{Z}$ to $\mathbf{Z}$ of Example 7.23 is not a one-to-one correspondence. It does satisfy property (1), since for every integer i there is some integer, namely, i^2, such that $S(i) = i^2$. However, it fails to satisfy (2), since there are some b's in $\mathbf{Z}$ — in particular all the negative integers — that are not $S(a)$ for any a. S also fails to satisfy (3), since there are many examples of two distinct a's for which $S(a)$ equals the same b. For instance, $S(3) = 9$ and $S(-3) = 9$.

For an example of a one-to-one correspondence, consider the total function P from $\mathbf{Z}$ to $\mathbf{Z}$ defined by $P(a) = a+1$. That is, P adds 1 to any integer. For instance, $P(5) = 6$, and $P(-5) = -4$. An alternative way to look at the situation is that P consists of the tuples

$$\{\ldots, (-2,-1), (-1,0), (0,1), (1,2), \ldots\}$$

or that it has the graph of Fig. 7.17.

We claim that P is a one-to-one correspondence from integers to integers. First, it is a partial function, since when we add 1 to an integer a we get the unique integer $a+1$. It satisfies property (1), since for every integer a, there is some integer $a+1$, which is $P(a)$. Property (2) is also satisfied, since for every integer b there is some integer, namely, $b-1$, such that $P(b-1) = b$. Finally, property (3) is satisfied, because for an integer b there cannot be two distinct integers such that when you add 1 to either, the result is b. ✦

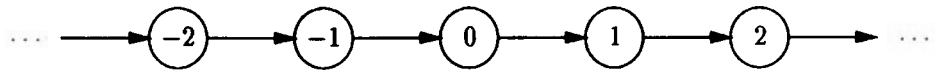


Fig. 7.17. Graph for the relation that is the function $P(a) = a + 1$.

A one-to-one correspondence from A to B is a way of establishing a unique association between the elements of A and B. For example, if we clap our hands together, the left and right thumbs touch, the left and right index fingers touch, and so on. We can think of this association between the set of fingers on the left hand and the fingers on the right hand as a one-to-one correspondence F, defined by F("left thumb") = "right thumb", F("left index finger") = "right index finger", and so on. We could similarly think of the association as the inverse function, from the right hand to the left. In general, a one-to-one correspondence from A to B can be inverted by switching the order of components in its pairs, to become a one-to-one correspondence from B to A.

A consequence of the existence of this one-to-one correspondence between hands is that the number of fingers on each hand is the same. That seems a natural and intuitive notion; two sets have the same number of elements exactly when there is a one-to-one correspondence from one set to the other. However, we shall see in Section 7.11 that when sets are infinite, there are some surprising conclusions we are forced to draw from this definition of "same number of elements."

EXERCISES

7.7.1: Give an example of sets A and B for which $A \times B$ is not the same as $B \times A$.

7.7.2: Let R be the relation defined by aRb, bRc, cRd, aRc, and bRd.

a) Draw the graph of R.
b) Is R a function?
c) Name two possible domains for R; name two possible ranges.
d) What is the smallest set S such that R is a relation on S (i.e., the domain and the range can both be S)?

7.7.3: Let T be a tree and let S be the set of nodes of T. Let R be the "child-parent" relation; that is, cRp if and only if c is a child of p. Answer the following, and justify your answers:

a) Is R a partial function, no matter what tree T is?
b) Is R a total function from S to S no matter what T is?
c) Can R ever be a one-to-one correspondence (i.e., for some tree T)?
d) What does the graph for R look like?

7.7.4: Let R be the relation on the set of integers $\{1, 2, \ldots, 10\}$ defined by aRb if a and b are distinct and have a common divisor other than 1. For example, $2R4$ and $6R9$, but not $2R3$.

a) Draw the graph for R.
b) Is R a function? Why or why not?

7.7.5*: Although we observed that $S = (A \times B) \times C$ and $T = A \times (B \times C)$ are not the same set, we can show that they are "essentially the same" by exhibiting a natural one-to-one correspondence between them. For each $\big((a, b), c\big)$ in S, let

$$F\Big(\big((a, b), c\big)\Big) = \big(a, (b, c)\big)$$

Show that F is a one-to-one correspondence from S to T.

7.7.6: What do the three statements $F(10) = 20$, $10F20$, and $(10, 20) \in F$ have in common?

Inverse relation

7.7.7*: The *inverse* of a relation R is the set of pairs (b, a) such that (a, b) is in R.

a) Explain how to get the graph of the inverse of R from the graph for R.
b) If R is a total function, is the inverse of R necessarily a function? What if R is a one-to-one correspondence?

7.7.8: Show that a relation is a one-to-one correspondence if and only if it is a total function and its inverse is also a total function.

7.8 Implementing Functions as Data

In a programming language, functions are usually implemented by code, but when their domain is small, they can be implemented using techniques quite similar to the ones we used for sets. In this section we shall discuss the use of linked lists, characteristic vectors, and hash tables to implement finite functions.

Functions as Programs and Functions as Data

analogy

Operations on Functions

The operations we most commonly perform on functions are similar to those for dictionaries. Suppose F is a function from domain set A to range set B. Then we may

1. *Insert* a new pair (a, b), such that $F(a) = b$. The only nuance is that, since F must be a function, should there already be a pair (a, c) for any c, this pair must be replaced by (a, b).
2. *Delete* the value associated with $F(a)$. Here, we need to give only the domain value a. If there is any b such that $F(a) = b$, the pair (a, b) is removed from the set. If there is no such pair, then no change is made.
3. *Lookup* the value associated with $F(a)$; that is, given domain value a, we return the value b such that $F(a) = b$. If there is no such pair (a, b) in the set, then we return some special value warning that $F(a)$ is undefined.

✦ **Example 7.25.** Suppose F consists of the pairs $\{(3, 9), (-4, 16), (0, 0)\}$; that is, $F(3) = 9$; $F(-4) = 16$, and $F(0) = 0$. Then ***lookup***(3) returns 9, and ***lookup***(2) returns a value indicating that no value is defined for $F(2)$. If F is the "squaring" function, the value -1 might be used to indicate a missing value, since -1 is not the true square of any integer.

The operation ***delete***(3) removes the pair $(3, 9)$, while ***delete***(2) has no effect. If we execute ***insert***$(5, 25)$, the pair $(5, 25)$ is added to the set F, or equivalently, we now have $F(5) = 25$. If we execute ***insert***$(3, 10)$, the old pair $(3, 9)$ is removed from F, and the new pair $(3, 10)$ is added to F, so that now $F(3) = 10$. ✦

Linked-List Representation of Functions

A function, being a set of pairs, can be stored in a linked list just like any other set. It is useful to define cells with three fields, one for the domain value, one for the range value, and one for a next-cell pointer. For example, we could define cells as

```
typedef struct CELL *LIST;
struct CELL {
    DTYPE domain;
    RTYPE range;
    LIST next;
};
```

where `DTYPE` is the type for domain elements and `RTYPE` is the type for range elements. Then a function is represented by a pointer to (the first cell of) a linked list.

The function in Fig. 7.18 performs the operation *insert*(a, b, L), assuming that `DTYPE` and `RTYPE` are both arrays of 32 characters. We search for a cell containing a in the `domain` field. If found, we set its `range` field to b. If we reach the end of the list, we create a new cell and store (a, b) therein. Otherwise, we test whether the cell has domain element a. If so, we change the range value to b, and we are done. If the domain has a value other than a, then we recursively insert into the tail of the list.

```
typedef char DTYPE[32], RTYPE[32];

void insert(DTYPE a, RTYPE b, LIST *pL)
{
    if ((*pL) == NULL) {/* at end of list */
        (*pL) = (LIST) malloc(sizeof(struct CELL));
        strcpy((*pL)->domain, a);
        strcpy((*pL)->range, b);
        (*pL)->next = NULL;

    else if (!strcmp(a, (*pL)->domain)) /* a = domain element;
            change F(a) */
        strcpy((*pL)->range, b);
    else /* domain element is not a */
        insert(a, b, &((*pL)->next));
};
```

Fig. 7.18. Inserting a new fact into a function represented as a linked list.

If the function F has n pairs, then *insert* takes $O(n)$ time on the average. Likewise, the analogous *delete* and *lookup* functions for a function represented as a linked list require $O(n)$ time on the average.

Vector Representation of Functions

Suppose the declared domain is the integers 0 through $DNUM - 1$, or it can be regarded as such, perhaps by being an enumeration type. Then we can use a generalization of a characteristic vector to represent functions. Define a type `FUNCT` for the characteristic vector as

```
typedef RTYPE FUNCT[DNUM];
```

Here it is essential that either the function be total or that `RTYPE` contain a value that we can interpret as "no value."

✦ **Example 7.26.** Suppose we want to store information about apples, like the harvest information of Fig. 7.9, but we now want to give the actual month of harvest, rather than the binary choice early/late. We can associate an integer constant with each element in the domain and range by defining the enumeration types

```
enum APPLES {Delicious, GrannySmith, Jonathan, McIntosh,
             Gravenstein, Pippin};
enum MONTHS {Unknown, Jan, Feb, Mar, Apr, May, Jun, Jul, Aug,
             Sep, Oct, Nov, Dec};
```

This declaration associates 0 with the identifier `Delicious`, 1 with `GrannySmith`, and so on. It also associates 0 with `Unknown`, 1 with `Jan`, and so on. The identifier `Unknown` indicates that the harvest month is not known. We can now declare an array

```
int Harvest[6];
```

with the intention that the array `Harvest` represents the set of pairs in Fig. 7.19. Then the array `Harvest` appears as in Fig. 7.20, where, for example, the entry `Harvest[Delicious] = Oct` means `Harvest[0] = 10`. ✦

APPLE	HARVEST MONTH
Delicious	Oct
Granny Smith	Aug
Jonathan	Sep
McIntosh	Oct
Gravenstein	Sep
Pippin	Nov

Fig. 7.19. Harvest months of apples.

Hash-Table Representation of Functions

We can store the pairs belonging to a function in a hash table. The crucial point is that we apply the hash function only to the domain element to determine the bucket of the pair. The cells in the linked lists forming the buckets have one field for the domain element, another for the corresponding range element, and a third to link one cell to the next on the list. An example should make the technique clear.

Delicious	Oct
GrannySmith	Aug
Jonathan	Sep
McIntosh	Oct
Gravenstein	Sep
Pippin	Nov

Fig. 7.20. The array `Harvest`.

✦ **Example 7.27.** Let us use the same data about apples that appeared in Example 7.26, except now we shall use the actual names rather than integers to represent the domain. To represent the function `Harvest`, we shall use a hash table with five buckets. We shall define `APPLES` to be 32-character arrays, while `MONTHS` is an enumeration as in Example 7.26. The buckets are linked lists with field `variety` for a domain element of type `APPLES`, field `harvested` for a range element of type `int` (a month), and a link field `next` to the next element of the list.

We shall use a hash function h similar to that shown in Fig. 7.11 of Section 7.6. Of course, h is applied to domain elements only — that is, to character strings of length 32, consisting of the name of an apple variety.

Now, we can define the type `HASHTABLE` as an array of `B LIST`'s. B is the number of buckets, which we have taken to be 5. All these declarations appear in the beginning of Fig. 7.22. We may then declare a hash table `Harvest` to represent the desired function.

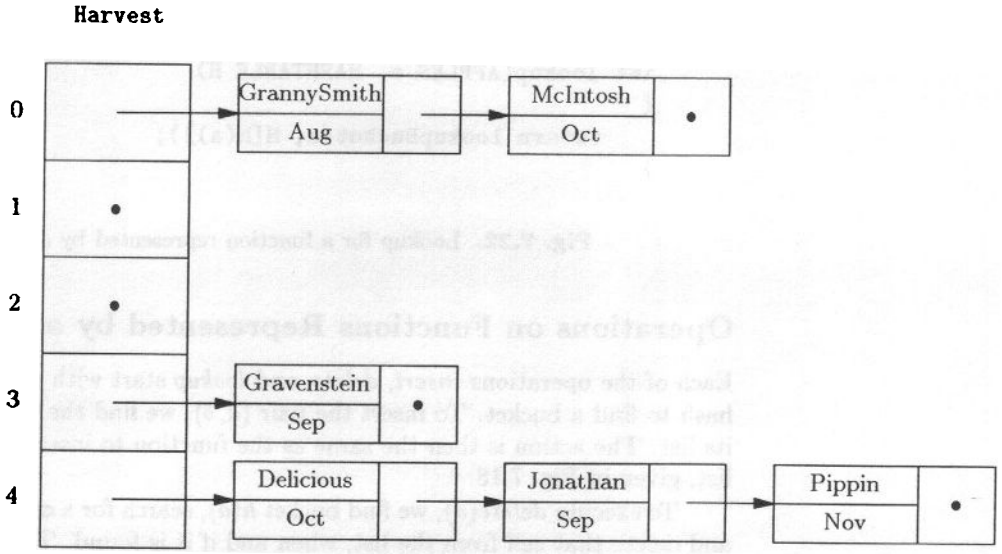


Fig. 7.21. Apples and their harvest months stored in a hash table.

After inserting the six apple varieties listed in Fig. 7.19, the arrangement of cells within buckets is shown in Fig. 7.21. For example, the word `Delicious` yields the sum 929 if we add up the integer values of the nine characters. Since the remainder

when 929 is divided by 5 is 4, the Delicious apple belongs in bucket 4. The cell for Delicious has that string in the **variety** field, the month **Oct** in the **harvested** field, and a pointer to the next cell of the bucket. ✦

```
#include <string.h>
#define B 5

typedef char APPLES[32];
enum MONTHS {Unknown, Jan, Feb, Mar, Apr, May, Jun, Jul, Aug,
            Sep, Oct, Nov, Dec};
typedef struct CELL *LIST;
struct CELL {
    APPLES variety;
    int harvested;
    LIST next;
};
typedef LIST HASHTABLE[B];

int lookupBucket(APPLES a, LIST L)
{
    if (L == NULL)
        return Unknown;
    if (!strcmp(a, L->variety)) /* found */
        return L->harvested;
    else /* a not found; examine tail */
        return lookupBucket(a, L->next);
}

int lookup(APPLES a, HASHTABLE H)
{
    return lookupBucket(a, H[h(a)]);
}
```

Fig. 7.22. Lookup for a function represented by a hash table.

Operations on Functions Represented by a Hash Table

Each of the operations *insert*, *delete*, and *lookup* start with a domain value that we hash to find a bucket. To insert the pair (a, b), we find the bucket $h(a)$ and search its list. The action is then the same as the function to insert a function pair into a list, given in Fig. 7.18.

To execute *delete*(a), we find bucket $h(a)$, search for a cell with domain value a, and delete that cell from the list, when and if it is found. The *lookup*(a) operation is executed by again hashing a and searching the bucket $h(a)$ for a cell with domain value a. If such a cell is found, the associated range value is returned.

For example, the function **lookup(a, H)** is shown in Fig. 7.22. The function **lookupBucket(a, L)** runs down the list L for a bucket and returns the value

harvested(a)

Vectors versus Hash Tables

There is a fundamental difference in the way we viewed the information about apples in Examples 7.26 and 7.27. In the characteristic-vector approach, apple varieties were a fixed set, which became an enumerated type. There is no way, while a C program is running, to change the set of apple names, and it is meaningless to perform a lookup with a name that is not part of our enumerated set.

On the other hand, when we set the same function up as a hash table, we treated the apple names as character strings, rather than members of an enumerated type. As a consequence, it is possible to modify the set of names while the program is running — say, in response to some input data about new apple varieties. It makes sense for a lookup to be performed for a variety not in the hash table, and we had to make provisions, by the addition of a "month" `Unknown`, for the possibility that we would look up a variety that was not mentioned in our table. Thus, the hash table offers increased flexibility over the characteristic vector, at some cost in speed.

that is, the month in which apple variety a is harvested. If the month is undefined, it returns the value `Unknown`.

Efficiency of Operations on Functions

The times required for the operations on functions for the three representations we have discussed here are the same as for the operations of the same names on dictionaries. That is, if the function consists of n pairs, then the linked-list representation requires $O(n)$ time per operation on the average. The characteristic-vector approach requires only $O(1)$ time per operation, but, as for dictionaries, it can be used only if the domain type is of limited size. The hash table with B buckets offers average time per operation of $O(n/B)$. If it is possible to make B close to n, then $O(1)$ time per operation, on the average, can be achieved.

EXERCISES

7.8.1: Write functions that perform (a) *delete* and (b) *lookup* on functions represented by linked lists, analogous to the *insert* function of Fig. 7.18.

7.8.2: Write functions that perform (a) *insert*, (b) *delete*, and (c) *lookup* on a function represented by a vector, that is, an array of `RTYPE`'s indexed by integers representing `DTYPE`'s.

7.8.3: Write functions that perform (a) *insert* and (b) *delete* on functions represented by hash tables, analogous to the *lookup* function of Fig. 7.22.

7.8.4: A binary search tree can also be used to represent functions as data. Define appropriate data structures for a binary search tree to hold the apple information in Fig. 7.19, and implement (a) *insert*, (b) *delete*, and (c) *lookup* using these structures.

7.8.5: Design an information retrieval system to keep track of information about at bats and hits for baseball players. Your system should accept triples of the form

```
Ruth 5 2
```

to indicate that Ruth in 5 at bats got 2 hits. The entry for Ruth should be updated appropriately. You should also be able to query the number of at bats and hits for any player. Implement your system so that the functions *insert* and *lookup* will work on any data structure as long as they use the proper subroutines and types.

7.9 Implementing Binary Relations

The implementation of binary relations differs in some details from the implementation of functions. Recall that both binary relations and functions are sets of pairs, but a function has for each domain element a at most one pair of the form (a, b) for any b. In contrast, a binary relation can have any number of range elements associated with a given domain element a.

In this section, we shall first consider the meaning of insertion, deletion, and lookup for binary relations. Then we see how the three implementations we have been using — linked lists, characteristic vectors, and hash tables — generalize to binary relations. In Chapter 8, we shall discuss implementation of relations with more than two components. Frequently, data structures for such relations are built from the structures for functions and binary relations.

Operations on Binary Relations

When we insert a pair (a, b) into a binary relation R, we do not have to concern ourselves with whether or not there already is a pair (a, c) in R, for some $c \neq b$, as we do when we insert (a, b) into a function. The reason, of course, is that there is no limit on the number of pairs in R that can have the domain value a. Thus, we shall simply insert the pair (a, b) into R as we would insert an element into any set.

Likewise, deletion of a pair (a, b) from a relation is similar to deletion of an element from a set: we look for the pair and remove it if it is present.

The *lookup* operation can be defined in several ways. For example, we could take a pair (a, b) and ask whether this pair is in R. However, if we interpret *lookup* thus, along with the *insert* and *delete* operations we just defined, a relation behaves like any dictionary. The fact that the elements being operated upon are pairs, rather than atomic, is a minor detail; it just affects the type of elements in the dictionary.

However, it is often useful to define *lookup* to take a domain element a and return all the range elements b such that (a, b) is in the binary relation R. This interpretation of *lookup* gives us an abstract data type that is somewhat different from the dictionary, and it has certain uses that are distinct from those of the dictionary ADT.

✦ **Example 7.28.** Most varieties of plums require one of several other specific varieties for pollination; without the appropriate "pollinizer," the tree cannot bear fruit. A few varieties are "self-fertile": they can serve as their own pollinizer. Figure 7.23 shows a binary relation on the set of plum varieties. A pair (a, b) in this relation means that variety b is a pollinizer for variety a.

Inserting a pair into this table corresponds to asserting that one variety is a pollinizer for another. For example, if a new variety is developed, we might enter into the relation facts about which varieties pollinize the new variety, and which it

VARIETY	POLLINIZER
Beauty	Santa Rosa
Santa Rosa	Santa Rosa
Burbank	Beauty
Burbank	Santa Rosa
Eldorado	Santa Rosa
Eldorado	Wickson
Wickson	Santa Rosa
Wickson	Beauty

Fig. 7.23. Pollinizers for certain plum varieties.

More General Operations on Relations

We may want more information than the three operations *insert*, *delete*, and *lookup* can provide when applied to the plum varieties of Example 7.28. For example, we may want to ask "What varieties does Santa Rosa pollinate?" or "Does Eldorado pollinate Beauty?" Some data structures, such as a linked list, allow us to answer questions like these as fast as we can perform the three basic dictionary operations, if for no other reason than that linked lists are not very efficient for any of these operations.

A hash table based on domain elements does not help answer questions in which we are given a range element and must find all the associated domain elements — for instance, "What varieties does Santa Rosa pollinate?" We could, of course, base the hash function on range elements, but then we could not answer easily questions like "What pollinates Burbank?" We could base the hash function on a combination of the domain and range values, but then we couldn't answer either type of query efficiently; we could only answer easily questions like "Does Eldorado pollinate Beauty?"

There are ways to get questions of all these types answered efficiently. We shall have to wait, however, until the next chapter, on the relational model, to learn the techniques.

can pollinize. Deletion of a pair corresponds to a retraction of the assertion that one variety can pollinize another.

The lookup operation we defined takes a variety a as argument, looks in the first column for all pairs having the value a, and returns the set of associated range values. That is, we ask, "What varieties can pollinize variety a?" This question seems to be the one we are most likely to ask about the information in this table, because when we plant a plum tree, we must make sure that, if it is not self-fertile, then there is a pollinizer nearby. For instance, if we invoke *lookup*(Burbank), we expect the answer {Beauty, Santa Rosa}. ✦

Linked-List Implementation of Binary Relations

We can link the pairs of a relation in a list if we like. The cells of this list consist

of a domain element, a range element, and a pointer to the next cell, just like the cells for functions. Insertion and deletion are carried out as for ordinary sets, as discussed in Section 6.4. The only nuance is that equality of set members is determined by comparing both the field holding the domain element and the field holding the range element.

Lookup is a somewhat different operation from the operations of the same name we have encountered previously. We must go down the list, looking for cells with a particular domain value a, and we must assemble a list of the associated range values. An example will show the mechanics of the *lookup* operation on linked lists.

✦ **Example 7.29.** Suppose we want to implement the plum relation of Example 7.28 as a linked list. We could define the type **PVARIETY** as a character string of length 32; and cells, whose type we shall call **RCELL** (relation cell), can be defined by a structure:

```
typedef char PVARIETY[32];
typedef struct RCELL *RLIST;
struct RCELL {
    PVARIETY variety;
    PVARIETY pollinizer;
    RLIST next;
};
```

We also need a cell containing one plum variety and a pointer to the next cell, in order to build a list of the pollinizers of a given variety, and thus to answer a *lookup* query. This type we shall call **PCELL**, and we define

```
typedef struct PCELL *PLIST;
struct PCELL {
    PVARIETY pollinizer;
    PLIST next;
};
```

We can then define *lookup* by the function in Fig. 7.24.

The function *lookup* takes a domain element a and a pointer to the first cell of a linked list of pairs as arguments. We perform the *lookup*(a) operation on a relation R by calling `lookup(a,L)`, where L is a pointer to the first cell on the linked list representing relation R. Lines (1) and (2) are simple. If the list is empty, we return **NULL**, since surely there are no pairs with first component a in an empty list.

The hard case occurs when a is found in the domain field, called **variety**, in the first cell of the list. This case is detected at line (3) and handled by lines (4) through (7). We create at line (4) a new cell of type **PCELL**, which becomes the first cell on the list of **PCELL**'s that we shall return. Line (5) copies the associated range value into this new cell. Then at line (6) we call **lookup** recursively on the tail of the list L. The return value from this call, which is a pointer to the first cell on the resulting list (**NULL** if the list is empty), becomes the **next** field of the cell we created at line (4). Then at line (7) we return a pointer to the newly created cell, which holds one range value and is linked to cells holding other range values for domain value a, if any exist.

The last case occurs when the desired domain value a is not found in the first cell of the list L. Then we just call *lookup* on the tail of the list, at line (8), and

```
    PLIST lookup(PVARIETY a, RLIST L)
    {
        PLIST P;

(1)     if (L == NULL)
(2)         return NULL;
(3)     else if (!strcmp(L->variety, a)) /* L->variety == a */ {
(4)         P = (PLIST) malloc(sizeof(struct PCELL));
(5)         strcpy(P->pollinizer, L->pollinizer);
(6)         P->next = lookup(a, L->next);
(7)         return P;
        }
        else /* a not the domain value of current pair */
(8)         return lookup(a, L->next);
    }
```

Fig. 7.24. Lookup in a binary relation represented by a linked list.

return whatever that call returns. ✦

A Characteristic-Vector Approach

For sets and for functions, we saw that we could create an array indexed by elements of a "universal" set and place appropriate values in the array. For sets, the appropriate array values are **TRUE** and **FALSE**, and for functions they are those values that can appear in the range, plus (usually) a special value that means "none."

For binary relations, we can index an array by members of a small declared domain, just as we did for functions. However, we cannot use a single value as an array element, because a relation can have any number of range values for a given domain value. The best we can do is to use as an array element the header of a linked list that contains all the range values associated with a given domain value.

✦ **Example 7.30.** Let us redo the plum example using this organization. As was pointed out in the last section, when we use a characteristic-vector style, we must fix the set of values, in the domain at least; there is no such constraint for linked-list or hash-table representations. Thus, we must redeclare the **PVARIETY** type to be an enumerated type:

```
enum PVARIETY {Beauty, SantaRosa, Burbank, Eldorado, Wickson};
```

We can continue to use the **PCELL** type for lists of varieties, as defined in Example 7.29. Then we may define the array

```
PLIST Pollinizers[5];
```

That is, the array representing the relation of Fig. 7.23 is indexed by the varieties mentioned in that figure, and the value associated with each variety is a pointer to the first cell on its list of pollinizers. Figure 7.25 shows the pairs of Fig. 7.23 represented in this way. ✦

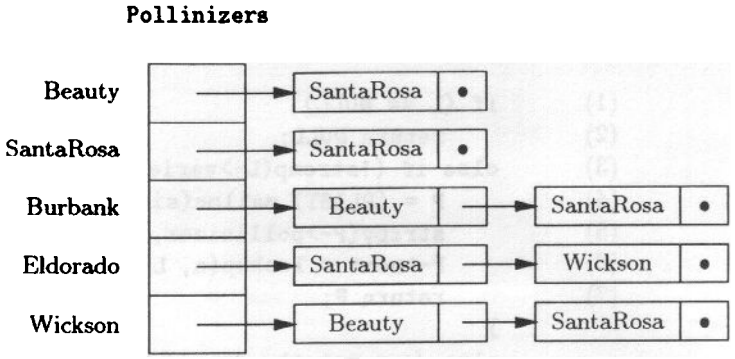


Fig. 7.25. Characteristic-vector representation of the pollinizers relation.

Insertion and deletion of pairs is performed by going to the appropriate array element and thence to the linked list. At that point, insertion in or deletion from the list is performed normally. For example, if we determined that Wickson cannot adequately pollinate Eldorado, we could execute the operation

delete(Eldorado, Wickson)

The header of the list for Eldorado is found in `Pollinizers[Eldorado]`, and from there we go down the list until we find a cell holding Wickson and delete it.

Lookup is trivial; we have only to return the pointer found in the appropriate array entry. For example, to answer the query `lookup(Burbank, Pollinizers)`, we simply return the list `Pollinizers[Burbank]`.

Hash-Table Implementation of Binary Relations

We may store a given binary relation R in a hash table, using a hash function that depends only on the first component of a pair. That is, the pair (a, b) will be placed in bucket $h(a)$, where h is the hash function. Note that this arrangement is exactly the same as that for a function; the only difference is that for a binary relation a bucket may contain more than one pair with a given value a as the first component, whereas for a function, it could never contain more than one such pair.

To insert the pair (a, b), we compute $h(a)$ and examine the bucket with that number to be sure that (a, b) is not already there. If it is not, we append (a, b) to the end of the list for that bucket. To delete (a, b), we go to the bucket $h(a)$, search for this pair, and remove it from the list if it is there.

To execute *lookup*(a), we again find the bucket $h(a)$ and go down the list for this bucket, collecting all the b's that appear in cells with first component a. The *lookup* function of Fig. 7.24, which we wrote for a linked list, applies equally well to the list that forms one bucket of a hash table.

Running Time of Operations on a Binary Relation

The performance of the three representations for binary relations is not much different from the performance of the same structures on functions or dictionaries. Consider first the list representation. While we have not written the functions for *insert* and *delete*, we should be able to visualize that these functions will run down the entire list, searching for the target pair, and stop upon finding it. On a list of

length n, this search takes $O(n)$ average time, since we must scan the entire list if the pair is not found and, on the average, half the list if it is found.

For *lookup*, an examination of Fig. 7.24 should convince us that this function takes $O(1)$ time plus a recursive call on the tail of a list. We thus make n calls if the list is of length n, for a total time of $O(n)$.

Now consider the generalized characteristic vector. The operation *lookup*(a) is easiest. We go to the array element indexed by a, and there we find our answer, a list of all the b's such that (a, b) is in the relation. We don't even have to examine the elements or copy them. Thus, *lookup* takes $O(1)$ time when characteristic vectors are used.

On the other hand, *insert* and *delete* are less simple. To insert (a, b), we can go to the array element indexed by a easily enough, but we must search the entire list to make sure that (a, b) is not already there.[5] That requires an amount of time proportional to the average length of a list, that is, to the average number of range values associated with a given domain value. We shall call this parameter m. Another way to look at m is that it is n, the total number of pairs in the relation, divided by the number of different domain values. If we assume that any list is as likely to be searched as any other, then we require $O(m)$ time on the average to perform an *insert* or a *delete*.

Finally, let us consider the hash table. If there are n pairs in our relation and B buckets, we expect there to be an average of n/B pairs per bucket. However, the parameter m must be figured in as well. If there are n/m different domain values, then at most n/m buckets can be nonempty, since the bucket for a pair is determined only by the domain value. Thus, m is a lower bound on the average size of a bucket, regardless of B. Since n/B is also a lower bound, the time to perform one of the three operations is $O(\max(m, n/B))$.

✦ **Example 7.31.** Suppose there is a relation of 1000 pairs, distributed among 100 domain values. Then the typical domain value has 10 associated range values; that is, $m = 10$. If we use 1000 buckets — that is, $B = 1000$ — then m is greater than n/B, which is 1, and we expect the average bucket that we might actually search (because its number is $h(a)$ for some domain value a that appears in the relation) to have about 10 pairs. In fact, it will have on the average slightly more, because by coincidence, the same bucket could be $h(a_1)$ and $h(a_2)$ for different domain values a_1 and a_2. If we choose $B = 100$, then $m = n/B = 10$, and we would again expect each bucket we might search to have about 10 elements. As just mentioned, the actual number is slightly more because of coincidences, where two or more domain values hash to the same bucket. ✦

EXERCISES

7.9.1: Using the data types from Example 7.29, write a function that takes a pollinizer value b and a list of variety-pollinizer pairs, and returns a list of the varieties that are pollinized by b.

[5] We could insert the pair without regard for whether it is already present, but that would have both the advantages and disadvantages of the list representation discussed in Section 6.4, where we allowed duplicates.

"Dictionary Operations" on Functions and Relations

A set of pairs might be thought of as a set, as a function, or as a relation. For each of these cases, we have defined operations *insert*, *delete*, and *lookup* suitably. These operations differ in form. Most of the time, the operation takes both the domain and range element of the pair. However, sometimes only the domain element is used as an argument. The table below summarizes the differences in the use of these three operations.

	Set of Pairs	Function	Relation
Insert	Domain and Range	Domain and Range	Domain and Range
Delete	Domain and Range	Domain only	Domain and Range
Lookup	Domain and Range	Domain only	Domain only

7.9.2: Write (a) *insert* and (b) *delete* routines for variety-pollinizer pairs using the assumptions of Example 7.29.

7.9.3: Write (a) *insert*, (b) *delete*, and (c) *lookup* functions for a relation represented by the vector data structure of Example 7.30. When inserting, do not forget to check for an identical pair already in the relation.

7.9.4: Design a hash-table data structure to represent the pollinizer relation that forms the primary example of this section. Write functions for the operations *insert*, *delete*, and *lookup*.

7.9.5*: Prove that the function **lookup** of Fig. 7.24 works correctly, by showing by induction on the length of list L that **lookup** returns a list of all the elements b such that the pair (a, b) is on the list L.

7.9.6*: Design a data structure that allows $O(1)$ average time to perform each of the operations *insert*, *delete*, *lookup*, and *inverseLookup*. The latter operation takes a range element and finds the associated domain elements.

7.9.7: In this section and the previous, we defined some new abstract data types that had operations we called *insert*, *delete*, and *lookup*. However, these operations were defined slightly differently from the operations of the same name on dictionaries. Make a table for the ADT's DICTIONARY, FUNCTION (as discussed in Section 7.8), and RELATION (as discussed in this section) and indicate the possible abstract implementations and the data structures that support them. For each, indicate the running time of each operation.

7.10 Some Special Properties of Binary Relations

In this section we shall consider some of the special properties that certain useful binary relations have. We begin by defining some basic properties: transitivity, reflexivity, symmetry, and antisymmetry. These are combined to form common types of binary relations: partial orders, total orders, and equivalence relations.

Transitivity

Let R be a binary relation on the domain D. We say that the relation R is *transitive* if whenever aRb and bRc are true, aRc is also true. Figure 7.26 illustrates the transitivity property as it appears in the graph of a relation. Whenever the dotted arrows from a to b and from b to c appear in the diagram, for some particular a, b, and c, then the solid arrow from a to c must also be in the diagram. It is important to remember that transitivity, like the other properties to be defined in this section, pertains to the relation as a whole. It is not enough that the property be satisfied for three particular domain elements; it must be satisfied for all triples a, b, and c in the declared domain D.

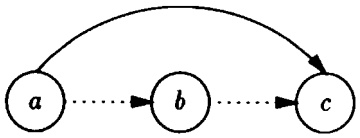


Fig. 7.26. Transitivity condition requires that if both the arcs aRb and bRc are present in the graph of a relation, then so is the arc aRc.

✦ **Example 7.32.** Consider the relation $<$ on $\mathbf{Z}$, the set of integers. That is, $<$ is the set of pairs of integers (a, b) such that a is less than b. The relation $<$ is transitive, because if $a < b$ and $b < c$, we know that $a < c$. Similarly, the relations $\leq$, $>$, and $\geq$ on integers are transitive. These four comparison relations are likewise transitive on the set of real numbers.

However, consider the relation $\neq$ on the integers (or the reals for that matter). This relation is not transitive. For instance, let a and c both be 3, and let b be 5. Then $a \neq b$ and $b \neq c$ are both true. If the relation were transitive, we would have $a \neq c$. But that says $3 \neq 3$, which is wrong. We conclude that $\neq$ is not transitive.

Transitivity of subset

For another example of a transitive relation, consider $\subseteq$, the subset relation. We might like to consider the relation as being the set of all pairs of sets (S, T) such that $S \subseteq T$, but to imagine that there is such a set would lead us to Russell's paradox again. However, suppose we have a "universal" set U. We can let $\subseteq_U$ be the set of pairs of sets

$$\{(S,T) \mid S \subseteq T \text{ and } T \subseteq U\}$$

Then $\subseteq_U$ is a relation on $\mathbf{P}(U)$, the power set of U, and we can think of $\subseteq_U$ as the subset relation.

For instance, let $U = \{1, 2\}$. Then $\subseteq_{\{1,2\}}$ consists of the nine (S, T)-pairs shown in Fig. 7.27. Thus, $\subseteq_U$ contains exactly those pairs such that the first component is a subset (not necessarily proper) of the second component and both are subsets of $\{1, 2\}$.

It is easy to check that $\subseteq_U$ is transitive, no matter what the universal set U is. If $A \subseteq B$ and $B \subseteq C$, then it must be that $A \subseteq C$. The reason is that for every x in A, we know that x is in B, because $A \subseteq B$. Since x is in B, we know that x is in C, because $B \subseteq C$. Thus, every element of A is an element of C. Therefore, $A \subseteq C$. ✦

S	T
$\emptyset$	$\emptyset$
$\emptyset$	$\{1\}$
$\emptyset$	$\{2\}$
$\emptyset$	$\{1,2\}$
$\{1\}$	$\{1\}$
$\{1\}$	$\{1,2\}$
$\{2\}$	$\{2\}$
$\{2\}$	$\{1,2\}$
$\{1,2\}$	$\{1,2\}$

Fig. 7.27. The pairs in the relation $\subseteq_{\{1,2\}}$.

Reflexivity

Some binary relations R have the property that for every element a in the declared domain, R has the pair (a, a); that is, aRa. If so, we say that R is *reflexive*. Figure 7.28 suggests that the graph of a reflexive relation has a loop on every element of its declared domain. The graph may have other arrows in addition to the loops. However, it is not sufficient that there be loops for the elements of the current domain; there must be one for each element of the declared domain.

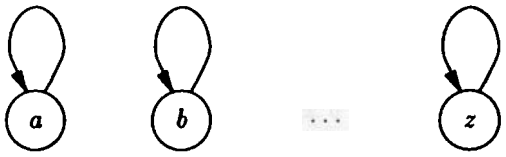


Fig. 7.28. A reflexive relation R has xRx for every x in its declared domain.

✦ **Example 7.33.** The relation $\geq$ on the reals is reflexive. For each real number a, we have $a \geq a$. Similarly, $\leq$ is reflexive, and both these relations are also reflexive on the integers. However, $<$ and $>$ are not reflexive, since $a < a$ and $a > a$ are each false for at least one value of a; in fact, they are both false for all a.

The subset relations $\subseteq_U$ defined in Example 7.32 are also reflexive, since $A \subseteq A$ for any set A. However, the similarly defined relations $\subset_U$ that contain the pair (S, T) if $T \subseteq U$ and $S \subset T$ — that is, S is a proper subset of T — are not reflexive. The reason is that $A \subset A$ is false for some A (in fact, for all A). ✦

Symmetry and Antisymmetry

Inverse relation

Let R be a binary relation. As defined in Exercise 7.7.7, the *inverse* of R is the set of pairs of R with the components reversed. That is, the inverse of R, denoted R^{-1}, is

$$\{(b, a) \mid (a, b) \in R\}$$

For example, $>$ is the inverse of $<$, since $a > b$ exactly when $b < a$. Likewise, $\geq$ is the inverse of $\leq$.

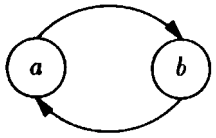


Fig. 7.29. Symmetry requires that if aRb, then bRa as well.

We say that R is *symmetric* if it is its own inverse. That is, R is symmetric if, whenever aRb, we also have bRa. Figure 7.29 suggests what symmetry looks like in the graph of a relation. Whenever the forward arc is present, the backward arc must also be present.

We say that R is *antisymmetric* if aRb and bRa are both true only when $a = b$. Note that it is not necessary that aRa be true for any particular a in an antisymmetric relation. However, an antisymmetric relation can be reflexive. Figure 7.30 shows how the antisymmetry condition relates to graphs of relations.

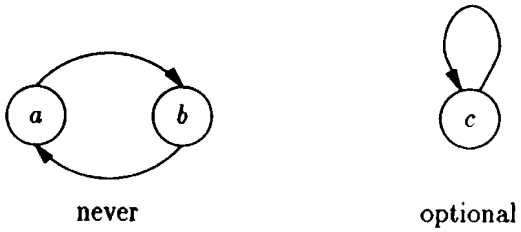


Fig. 7.30. An antisymmetric relation cannot have a cycle involving two elements, but loops on a single element are permitted.

✦ **Example 7.34.** The relation $\leq$ on integers or reals is antisymmetric, because if $a \leq b$ and $b \leq a$, it must be that $a = b$. The relation $<$ is also antisymmetric, because under no circumstances are $a < b$ and $b < a$ both true. Similarly, $\geq$ and $>$ are antisymmetric, as are the subset relations $\subseteq_U$ that we discussed in Example 7.32.

However, note that $\leq$ is not symmetric. For example, $3 \leq 5$, but $5 \leq 3$ is false. Likewise, none of the other relations mentioned in the previous paragraph is symmetric.

An example of a symmetric relation is $\neq$ on the integers. That is, if $a \neq b$, then surely $b \neq a$. ✦

Pitfalls in Property Definitions

As we have pointed out, the definition of a property is a general condition, one that applies to all elements of the domain. For example, in order for a relation R on declared domain D to be reflexive, we need to have aRa for every $a \in D$. It is not sufficient for aRa to be true for one a, nor does it make sense to say that a relation is reflexive for some elements and not others. If there is even one a in D for which aRa is false, then R is not reflexive. (Thus, reflexivity may depend on the domain, as well as on the relation R.)

Also, a condition like transitivity — "if aRb and bRc then aRc" — is of the form "if A then B." Remember that we can satisfy such a statement either by making B true or by making A false. Thus, for a given triple a, b, and c, the transitivity condition is satisfied whenever aRb is false, or whenever bRc is false, or whenever aRc is true. As an extreme example, the empty relation is transitive, symmetric, and antisymmetric, because the "if" condition is never satisfied. However, the empty relation is not reflexive, unless the declared domain is $\emptyset$.

Partial Orders and Total Orders

A *partial order* is a transitive and antisymmetric binary relation. A relation is said to be a *total order* if in addition to being transitive and antisymmetric, it makes every pair of elements in the domain *comparable*. That is to say, if R is a total order, and if a and b are any two elements in its domain, then either aRb or bRa is true. Note that every total order is reflexive, because we may let a and b be the same element, whereupon the comparability requirement tells us that aRa.

Comparable elements

✦ **Example 7.35.** The arithmetic comparisons $\leq$ and $\geq$ on integers or reals are total orders and therefore are also partial orders. Notice that for any a and b, either $a \leq b$ or $b \leq a$, but both are true exactly when $a = b$.

The comparisons $<$ and $>$ are partial orders but not total orders. While they are antisymmetric, they are not reflexive; that is, neither $a < a$ nor $a > a$ is true.

The subset relations $\subseteq_U$ and $\subset_U$ on 2^U for some universal set U are partial orders. We already observed that they are transitive and antisymmetric. These relations are not total orders, however, as long as U has at least two members, since then there are incomparable elements. For example, let $U = \{1, 2\}$. Then $\{1\}$ and $\{2\}$ are subsets of U, but neither is a subset of the other. ✦

One can view a total order R as a linear sequence of elements, as suggested in Fig. 7.31, where whenever aRb for distinct elements a and b, a appears to the left of b along the line. For example, if R is $\leq$ on the integers, then the elements along the line would be $\ldots, -2, -1, 0, 1, 2, \ldots$. If R is $\leq$ on the reals, then the points correspond to the points along the real line, as if the line were an infinite ruler; the real number x is found x units to the right of the 0 mark if x is nonnegative, and $-x$ units to the left of the zero mark if x is negative.

If R is a partial order but not a total order, we can also draw the elements of the domain in such a way that if aRb, then a is to the left of b. However, because there may be some incomparable elements, we cannot necessarily draw the elements

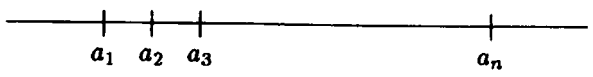


Fig. 7.31. Picture of a total order on $a_1, a_2, a_3, \ldots, a_n$.

in one line so that the relation R means "to the left."

✦ **Example 7.36.** Figure 7.32 represents the partial order $\subseteq_{\{1,2,3\}}$. We have drawn the relation as a *reduced graph*, in which we have omitted arcs that can be inferred by transitivity. That is, $S \subseteq_{\{1,2,3\}} T$ if either

Reduced graph

1. $S = T$,
2. There is an arc from S to T, or
3. There is a path of two or more arcs leading from S to T.

For example, we know that $\emptyset \subseteq_{\{1,2,3\}} \{1,3\}$, because of the path from $\emptyset$ to $\{1\}$ to $\{1,3\}$. ✦

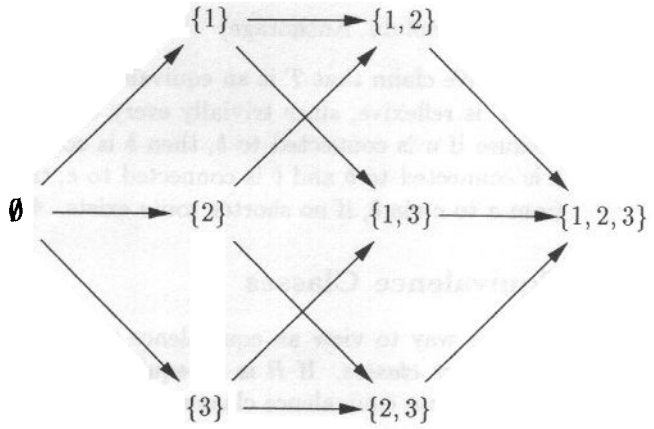


Fig. 7.32. Reduced graph for the partial order $\subseteq_{\{1,2,3\}}$.

Equivalence Relations

An *equivalence relation* is a binary relation that is reflexive, symmetric, and transitive. This kind of relation is quite different from the partial orders and total orders we have met in our previous examples. In fact, a partial order can never be an equivalence relation, except in the trivial cases that the declared domain is empty, or there is only one element a in the declared domain and the relation is $\{(a, a)\}$.

✦ **Example 7.37.** A relation like $\leq$ on integers is not an equivalence relation. Although it is transitive and reflexive, it is not symmetric. If $a \leq b$, we do not have $b \leq a$, except if $a = b$.

For an example that is an equivalence relation, let R consist of those pairs of integers (a, b) such that $a - b$ is an integer multiple of 3. For example $3R9$, since

$3 - 9 = -6 = 3 \times (-2)$. Also, $5R(-4)$, since $5 - (-4) = 9 = 3 \times 3$. However, $(1, 2)$ is not in R (or we can say "$1R2$ is false"), since $1 - 2 = -1$, which is not an integer multiple of 3. We can demonstrate that R is an equivalence relation, as follows:

1. R is reflexive, since aRa for any integer a, because $a - a$ is zero, which is a multiple of 3.
2. R is symmetric. If $a - b$ is a multiple of 3 — say, $3c$ for some integer c — then $b - a$ is $-3c$ and is therefore also an integer multiple of 3.
3. R is transitive. Suppose aRb and bRc. That is, $a - b$ is a multiple of 3, say, $3d$; and $b - c$ is a multiple of 3, say, $3e$. Then

$$a - c = (a - b) + (b - c) = 3d + 3e = 3(d + e)$$

and so $a - c$ is also a multiple of 3. Thus, aRb and bRc imply aRc, which means that R is transitive.

For another example, let S be the set of cities of the world, and let T be the relation defined by aTb if a and b are connected by roads, that is, if it is possible to drive by car from a to b. Thus, the pair (Toronto, New York) is in T, but

(Honolulu, Anchorage)

is not. We claim that T is an equivalence relation.

T is reflexive, since trivially every city is connected to itself. T is symmetric because if a is connected to b, then b is connected to a. T is transitive because if a is connected to b and b is connected to c, then a is connected to c; we can travel from a to c via b, if no shorter route exists. ✦

Equivalence Classes

Another way to view an equivalence relation is that it partitions its domain into *equivalence classes*. If R is an equivalence relation on a domain D, then we can divide D into equivalence classes so that

1. Each domain element is in exactly one equivalence class.
2. If aRb, then a and b are in the same equivalence class.
3. If aRb is false, then a and b are in different equivalence classes.

✦ **Example 7.38.** Consider the relation R of Example 7.37, where aRb when $a - b$ is a multiple of 3. One equivalence class is the set of integers that are exactly divisible by 3, that is, those that leave a remainder of 0 when divided by 3. This class is $\{\ldots, -3, 0, 3, 6, \ldots\}$. A second is the set of integers that leave a remainder of 1 when divided by 3, that is, $\{\ldots, -2, 1, 4, 7, \ldots\}$. The last class is the set of integers that leave a remainder of 2 when divided by 3. This class is $\{\ldots, -1, 2, 5, 8, \ldots\}$. The classes partition the set of integers into three disjoint sets, as suggested by Fig. 7.33.

Notice that when two integers leave the same remainder when divided by 3, then their difference is evenly divided by 3. For instance, $14 = 3 \times 4 + 2$ and $5 = 3 \times 1 + 2$. Thus, $14 - 5 = 3 \times 4 - 3 \times 1 + 2 - 2 = 3 \times 3$. We therefore know that $14R5$. On the other hand, if two integers leave different remainders when divided by

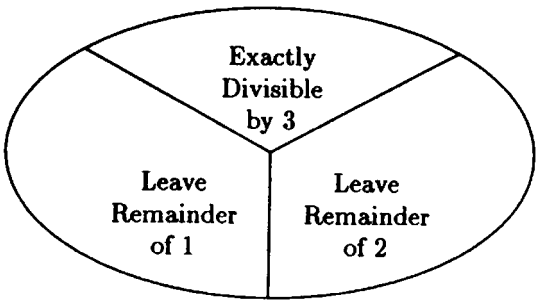


Fig. 7.33. Equivalence classes for the relation on the integers: "Difference is divisible by 3."

3, their difference surely is not evenly divisible by 3. Thus, integers from different classes, like 5 and 7, are not related by R. ✦

To construct the equivalence classes for an equivalence relation R, let $class(a)$ be the set of elements b such that aRb. For example, if our equivalence relation is the one we called R in Example 7.37, then $class(4)$ is the set of integers that leave a remainder of 1 when divided by 3; that is $class(4) = \{\ldots, -2, 1, 4, 7, \ldots\}$.

Notice that if we let a vary over each of the elements of the domain, we typically get the same class many times. In fact, when aRb, then $class(a) = class(b)$. To see why, suppose that c is in $class(a)$. Then aRc, by definition of *class*. Since we are given that aRb, by symmetry it follows that bRa. By transitivity, bRa and aRc imply bRc. But bRc says that c is in $class(b)$. Thus, every element in $class(a)$ is in $class(b)$. Since the same argument tells us that, as long as aRb, every element in $class(b)$ is also in $class(a)$, we conclude that $class(a)$ and $class(b)$ are identical.

However, if $class(a)$ is not the same as $class(b)$, then these classes can have no element in common. Suppose otherwise. Then there must be some c in both $class(a)$ and $class(b)$. By our previous assumption, we know that aRc and bRc. By symmetry, cRb. By transitivity, aRc and cRb imply aRb. But we just showed that whenever aRb is true, $class(a)$ and $class(b)$ are the same. Since we assumed these classes were not the same, we have a contradiction. Therefore, the assumed c in the intersection of $class(a)$ and $class(b)$ cannot exist.

There is one more observation we need to make: every domain element is in some equivalence class. In particular, a is always in $class(a)$, because reflexivity tells us aRa.

We can now conclude that an equivalence relation divides its domain into equivalence classes that are disjoint and that place each element into exactly one class. Example 7.38 illustrated this phenomenon.

Closures of Relations

A common operation on relations is to take a relation that does not have some property and add as few pairs as possible to create a relation that does have that property. The resulting relation is called the *closure* (for that property) of the original relation.

Transitive closure

✦ **Example 7.39.** We discussed reduced graphs in connection with Fig. 7.32. Although we were representing a transitive relation, $\subseteq_{\{1,2,3\}}$, we drew arcs corresponding to only a subset of the pairs in the relation. We can reconstruct the entire relation by applying the transitive law to infer new pairs, until no new pairs can be inferred. For example, we see that there are arcs corresponding to the pairs $(\{1\},\{1,3\})$ and $(\{1,3\},\{1,2,3\})$, and so the transitive law tells us that the pair $(\{1\},\{1,2,3\})$ must also be in the relation. Then this pair, together with the pair $(\emptyset,\{1\})$ tells us that $(\emptyset,\{1,2,3\})$ is in the relation. To these we must add the "reflexive" pairs (S,S), for each set S that is a subset of $\{1,2,3\}$. In this manner, we can reconstruct all the pairs in the relation $\subseteq_{\{1,2,3\}}$. ✦

Topological sorting

Another useful closure operation is *topological sorting*, where we take a partial order and add tuples until it becomes a total order. While the transitive closure of a binary relation is unique, there are frequently several total orders that contain a given partial order. We shall learn in Chapter 9 of a surprisingly efficient algorithm for topological sorting. For the moment, let us consider an example where topological sorting is useful.

✦ **Example 7.40.** It is common to represent a sequence of tasks that must be performed in a manufacturing process by a set of "precedences" that must be obeyed. For a simple example, you must put on your left sock before your left shoe, and your right sock before your right shoe. However, there are no other precedences that must be obeyed. We can represent these precedences by a set consisting of the two pairs $(leftsock, leftshoe)$ and $(rightsock, rightshoe)$. This set is a partial order.

We can extend this relation to six different total orders. One is the total order in which we dress the left foot first; this relation is a set that contains the ten pairs

$(leftsock, leftsock)$ $(leftsock, leftshoe)$ $(leftsock, rightsock)$ $(leftsock, rightshoe)$
$(leftshoe, leftshoe)$ $(leftshoe, rightsock)$ $(leftshoe, rightshoe)$
$(rightsock, rightsock)$ $(rightsock, rightshoe)$
$(rightshoe, rightshoe)$

We can think of this total order as the linear arrangement

$$leftsock \rightarrow leftshoe \rightarrow rightsock \rightarrow rightshoe$$

There is the analogous procedure where we dress the right foot first.

There are four other possible total orders consistent with the original partial order, where we first put on the socks and then the shoes. These are represented by the linear arrangements

$$leftsock \rightarrow rightsock \rightarrow leftshoe \rightarrow rightshoe$$
$$leftsock \rightarrow rightsock \rightarrow rightshoe \rightarrow leftshoe$$
$$rightsock \rightarrow leftsock \rightarrow leftshoe \rightarrow rightshoe$$
$$rightsock \rightarrow leftsock \rightarrow rightshoe \rightarrow leftshoe$$

✦

A third form of closure is to find the smallest equivalence relation containing a given relation. For example, a road map represents a relation consisting of pairs of cities connected by road segments having no intermediate cities. To determine the

Connected components

road-connected cities, we can apply reflexivity, transitivity, and symmetry to infer those pairs of cities that are connected by some sequence of these elementary roads. This form of closure is called finding the "connected components" in a graph, and an efficient algorithm for the problem will be discussed in Chapter 9.

EXERCISES

7.10.1: Give an example of a relation that is reflexive for one declared domain but not reflexive for another declared domain. Remember that for D to be a possible domain for a relation R, D must include every element that appears in a pair of R but it may also include more elements.

7.10.2**: How many pairs are there in the relation $\subseteq_{\{1,2,3\}}$? In general, how many pairs are there in $\subseteq_U$, if U has n elements? *Hint*: Try to guess the function from a few cases like the two-element case (Fig. 7.27) where there are 9 pairs. Then prove your guess correct by induction.

7.10.3: Consider the binary relation R on the domain of four-letter strings defined by sRt if t is formed from the string s by cycling its characters one position left. That is, $abcdRbcda$, where a, b, c, and d are individual letters. Determine whether R is (a) reflexive, (b) symmetric, (c) transitive, (d) a partial order, and/or (e) an equivalence relation. Give a brief argument why, or a counterexample, in each case.

7.10.4: Consider the domain of four-letter strings in Exercise 7.10.3. Let S be the binary relation consisting of R applied 0 or more times. Thus, $abcdSabcd$, $abcdSbcda$, $abcdScdab$, and $abcdSdabc$. Put another way, a string is related by S to any of its rotations. Answer the five questions from Exercise 7.10.3 for the relation S. Again, give justification in each case.

7.10.5*: What is wrong with the following "proof"?

(Non)Theorem: If binary relation R is symmetric and transitive, then R is reflexive.

(Non)Proof: Let x be some member of the domain of R. Pick y such that xRy. By symmetry, yRx. By transitivity, xRy and yRx imply xRx. Since x is an arbitrary member of R's domain, we have shown that xRx for every element in the domain of R, which "proves" that R is reflexive.

7.10.6: Give examples of relations with declared domain $\{1, 2, 3\}$ that are

a) Reflexive and transitive, but not symmetric
b) Reflexive and symmetric, but not transitive
c) Symmetric and transitive, but not reflexive
d) Symmetric and antisymmetric
e) Reflexive, transitive, and a total function
f) Antisymmetric and a one-to-one correspondence

7.10.7*: How many arcs are saved if we use the reduced graph for the relation $\subseteq_U$, where U has n elements, rather than the full graph?

7.10.8: Are (a) $\subseteq_U$ and (b) $\subset_U$ either partial orders or total orders when U has one element? What if U has zero elements?

7.10.9*: Show by induction on n, starting at $n = 1$, that if there is a sequence of n pairs $a_0Ra_1, a_1Ra_2, \ldots, a_{n-1}Ra_n$, and if R is a transitive relation, then a_0Ra_n. That is, show that if there is any path in the graph of a transitive relation, then there is an arc from the beginning of the path to the end.

7.10.10: Find the smallest equivalence relation containing the pairs (a, b), (a, c), (d, e), and (b, f).

7.10.11: Let R be the relation on the set of integers such that aRb if a and b are distinct and have a common divisor other than 1. Determine whether R is (a) reflexive, (b) symmetric, (c) transitive, (d) a partial order, and/or (e) an equivalence relation.

7.10.12: Repeat Exercise 7.10.11 for the relation R_T on the nodes of a particular tree T defined by aR_Tb if and only if a is an ancestor of b in tree T. However, unlike Exercise 7.10.11, your possible answers are "yes," "no," or "it depends on what tree T is."

7.10.13: Repeat Exercise 7.10.12 for relation S_T on the nodes of a particular tree T defined by aS_Tb if and only if a is to the left of b in tree T.

7.11 Infinite Sets

All of the sets that one would implement in a computer program are finite, or limited, in extent; one could not store them in a computer's memory if they were not. Many sets in mathematics, such as the integers or reals, are infinite in extent. These remarks seem intuitively clear, but what distinguishes a finite set from an infinite one?

The distinction between finite and infinite is rather surprising. A finite set is one that does not have the same number of elements as any of its proper subsets. Recall from Section 7.7 that we said we could use the existence of a one-to-one correspondence between two sets to establish that that they are *equipotent*, that is, they have the same number of members.

Equipotent sets

If we take a finite set such as $S = \{1, 2, 3, 4\}$ and any proper subset of it, such as $T = \{1, 2, 3\}$, there is no way to find a one-to-one correspondence between the two sets. For example, we could map 4 of S to 3 of T, 3 of S to 2 of T, and 2 of S to 1 of T, but then we would have no member of T to associate with 1 of S. Any other attempt to build a one-to-one correspondence from S to T must likewise fail.

Your intuition might suggest that the same should hold for any set whatsoever: how could a set have the same number of elements as a set formed by throwing away one or more of its elements? Consider the natural numbers (nonnegative integers) $\mathbf{N}$ and the proper subset of $\mathbf{N}$ formed by throwing away 0; call it $\mathbf{N} - \{0\}$, or $\{1, 2, 3, \ldots\}$. Then consider the one-to-one correspondence F from $\mathbf{N}$ to $\mathbf{N} - \{0\}$ defined by $F(0) = 1$, $F(1) = 2$, and, in general, $F(i) = i + 1$.

Surprisingly, F is a one-to-one correspondence from $\mathbf{N}$ to $\mathbf{N} - \{0\}$. For each i in $\mathbf{N}$, there is at most one j such that $F(i) = j$, so F is a function. In fact, there is exactly one such j, namely $i + 1$, so that condition (1) in the definition of one-to-one correspondence (see Section 7.7) is satisfied. For every j in $\mathbf{N} - \{0\}$ there is some i such that $F(i) = j$, namely, $i = j - 1$. Thus condition (2) in the definition of one-to-one correspondence is satisfied. Finally, there cannot be two

Infinite Hotels

To help you appreciate that there are as many numbers from 0 up as from 1 up, imagine a hotel with an infinite number of rooms, numbered 0, 1, 2, and so on; for any integer, there is a room with that integer as room number. At a certain time, there is a guest in each room. A kangaroo comes to the front desk and asks for a room. The desk clerk says, "We don't see many kangaroos around here." Wait — that's another story. Actually, the desk clerk makes room for the kangaroo as follows. He moves the guest in room 0 to room 1, the guest in room 1 to room 2, and so on. All the old guests still have a room, and now room 0 is vacant, and the kangaroo goes there. The reason this "trick" works is that there are truly the same number of rooms numbered from 1 up as are numbered from 0 up.

distinct numbers i_1 and i_2 in $\mathbf{N}$ such that $F(i_1)$ and $F(i_2)$ are both j, because then $i_1 + 1$ and $i_2 + 1$ would both be j, from which we would conclude that $i_1 = i_2$. We are forced to conclude that F is a one-to-one correspondence between $\mathbf{N}$ and its proper subset $\mathbf{N} - \{0\}$.

Formal Definition of Infinite Sets

The definition accepted by mathematicians of an *infinite set* is one that has a one-to-one correspondence between itself and at least one of its proper subsets. There are more extreme examples of how an infinite set and a proper subset can have a one-to-one correspondence between them.

✦ **Example 7.41.** The set of natural numbers and the set of even natural numbers are equipotent. Let $F(i) = 2i$. Then F is a one-to-one correspondence that maps 0 to 0, 1 to 2, 2 to 4, 3 to 6, and in general, every natural number to a unique natural number, its double.

Similarly, $\mathbf{Z}$ and $\mathbf{N}$ are the same size; that is, there are as many nonnegative and negative integers as nonnegative integers. Let $F(i) = 2i$ for all $i \geq 0$, and let $F(i) = -2i - 1$ for $i < 0$. Then 0 goes to 0, 1 to 2, -1 to 1, 2 to 4, -2 to 3, and so on. Every integer is sent to a unique nonnegative integer, with the negative integers going to odd numbers and the nonnegative integers to even numbers.

Even more surprising, the set of pairs of natural numbers is equinumerous with $\mathbf{N}$ itself. To see how the one-to-one correspondence is constructed, consider Fig. 7.34, which shows the pairs in $\mathbf{N} \times \mathbf{N}$ arranged in an infinite square. We order the pairs according to their sum, and among pairs of equal sum, by order of their first components. This order begins $(0,0)$, $(0,1)$, $(1,0)$, $(0,2)$, $(1,1)$, $(2,0)$, $(0,3)$, $(1,2)$, and so on, as suggested by Fig. 7.34.

Now, every pair has a place in the order. The reason is that for any pair (i,j), there are only a finite number of pairs with a smaller sum, and a finite number with the same sum and a smaller value of i. In fact, we can calculate the position of the pair (i,j) in the order; it is $(i+j)(i+j+1)/2 + i$. That is, our one-to-one correspondence associates the pair (i,j) with the unique natural number $(i+j)(i+j+1)/2 + i$.

Notice that we have to be careful how we order pairs. Had we ordered them by rows in Fig. 7.34, we would never get to the pairs on the second or higher rows,

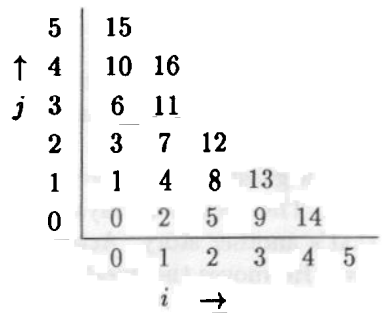


Fig. 7.34. Ordering pairs of natural numbers.

Every Set Is Either Finite or Infinite

At first glance, it might appear that there are things that are not quite finite and not quite infinite. For example, when we talked about linked lists, we put no limit on the length of a linked list. Yet whenever a linked list is created during the execution of a program, it has a finite length. Thus, we can make the following distinctions:

1. Every linked list is finite in length; that is, it has a finite number of cells.
2. The length of a linked list may be any nonnegative integer, and the set of possible lengths of linked lists is infinite.

because there are an infinite number of pairs on each row. Similarly, ordering by columns would not work. ✦

The formal definition of infinite sets is interesting, but that definition may not meet our intuition of what infinite sets are. For example, one might expect that an infinite set was one that, for every integer n, contained at least n elements. Fortunately, this property can be proved for every set that the formal definition tells us is infinite. The proof is an example of induction.

STATEMENT $S(n)$: If I is an infinite set, then I has a subset with n elements.

BASIS. Let $n = 0$. Surely $\emptyset \subseteq I$.

INDUCTION. Assume $S(n)$ for some $n \geq 0$. We shall prove that I has a subset with $n+1$ elements. By the inductive hypothesis, I has a subset T with n elements. By the formal definition of an infinite set, there is a proper subset $J \subset I$ and a 1-1 correspondence f from I to J. Let a be an element in $I - J$; surely a exists because J is a proper subset.

Consider R, the *image* of T under f, that is, if $T = \{b_1, \ldots, b_n\}$, then $R = \{f(b_1), \ldots, f(b_n)\}$. Since f is 1-1, each of $f(b_1), \ldots, f(b_n)$ are different, so R is of size n. Since f is from I to J, each of the $f(b_k)$'s is in J; that is, $R \subseteq J$. Thus, a

Cardinality of Sets

We defined two sets S and T to be equipotent (equal in size) if there is a one-to-one correspondence from S to T. Equipotence is an equivalence relation on any set of sets, and we leave this point as an exercise. The equivalence class to which a set S belongs is said to be the *cardinality* of S. For example, the empty set belongs to an equivalence class by itself; we can identify this class with cardinality 0. The class containing the set $\{a\}$, where a is any element, is cardinality 1, the class containing the set $\{a, b\}$ is cardinality 2, and so on.

Countable set, aleph-zero

The class containing **N** is "the cardinality of the integers," usually given the name *aleph-zero*, and a set in this class is said to be *countable*. The set of real numbers belongs to another equivalence class, often called *the continuum*. There are, in fact, an infinite number of different infinite cardinalities.

cannot be in R. It follows that $R \cup \{a\}$ is a subset of I with $n+1$ elements, proving $S(n+1)$.

Countable and Uncountable Sets

From Example 7.41, we might think that all infinite sets are equipotent. We've seen that **Z**, the set of integers, and **N**, the set of nonnegative integers, are the same size, as are some infinite subsets of these that intuitively "seem" smaller than **N**. Since we saw in Example 7.41 that the pairs of natural numbers are equinumerous with **N**, it follows that the nonnegative rational numbers are equinumerous with the natural numbers, since a rational is just a pair of natural numbers, its numerator and denominator. Likewise, the (nonnegative and negative) rationals can be shown to be just as numerous as the integers, and therefore as the natural numbers.

Any set S for which there is a one-to-one correspondence from S to **N** is said to be *countable*. The use of the term "countable" makes sense, because S must have an element corresponding to 0, an element corresponding to 1, and so on, so that we can "count" the members of S. From what we just said, the integers, the rationals, the even numbers, and the set of pairs of natural numbers are all countable sets. There are many other countable sets, and we leave the discovery of the appropriate one-to-one correspondences as exercises.

However, there are infinite sets that are not countable. In particular, the real numbers are not countable. In fact, we shall show that there are more real numbers between 0 and 1 than there are natural numbers. The crux of the argument is that the real numbers between 0 and 1 can each be represented by a decimal fraction of infinite length. We shall number the positions to the right of the decimal point 0, 1, and so on. If the reals between 0 and 1 are countable, then we can number them, r_0, r_1, and so on. We can then arrange the reals in an infinite square table, as suggested by Fig. 7.35. In our hypothetical listing of the real numbers between 0 and 1, $\pi/10$ is assigned to row zero, 5/9 is assigned to row one, 5/8 is assigned to row two, 4/33 is assigned to row three, and so on.

Diagonalization

However, we can prove that Fig. 7.35 does not really represent a listing of all the reals in the range 0 to 1. Our proof is of a type known as a *diagonalization*, where we use the diagonal of the table to create a value that cannot be in the list of reals. We create a new real number r with decimal representation $.a_0a_1a_2\cdots$.

		POSITIONS							
		0	1	2	3	4	5	6	···
	0	3	1	4	1	5	9	2	
REAL	1	5	5	5	5	5	5	5	
NUMBERS	2	6	2	5	0	0	0	0	
↓	3	1	2	1	2	1	2	1	
	4								

Fig. 7.35. Hypothetical table of real numbers, assuming that the reals are countable.

The value of the ith digit, a_i, depends on that of the ith diagonal digit, that is, on the value found at the ith position of the ith real. If this value is 0 through 4, we let $a_i = 8$. If the value at the ith diagonal position is 5 through 9, then $a_i = 1$.

✦ **Example 7.42.** Given the part of the table suggested by Fig. 7.35, our real number r begins $.8118\cdots$. To see why, note that the value at position 0 of real 0 is 3, and so $a_0 = 8$. The value at position 1 of real 1 is 5, and so $a_1 = 1$. Continuing, the value at position 2 of real 2 is 5 and the value at position 3 of real 3 is 2, and so the next two digits are 18. ✦

We claim that r does not appear anywhere in the hypothetical list of reals, even though we supposed that all real numbers from 0 to 1 were in the list. Suppose r were r_j, the real number associated with row j. Consider the difference d between r and r_j. We know that a_j, the digit in position j of the decimal expansion of r, was specifically chosen to differ by at least 4 and at most 8 from the digit in the jth position of r_j. Thus, the contribution to d from the jth position is between $4/10^{j+1}$ and $8/10^{j+1}$.

The contribution to d from all positions after the jth is no more than $1/10^{j+1}$, since that would be the difference if one of r and r_j had all 0's there and the other had all 9's. Hence, the contribution to d from all positions j and greater is between $3/10^{j+1}$ and $9/10^{j+1}$.

Finally, in positions before the jth, r and r_j are either the same, in which case the contribution to d from the first $j-1$ positions is 0, or r and r_j differ by at least $1/10^j$. In either case, we see that d cannot be 0. Thus, r and r_j cannot be the same real number.

We conclude that r does not appear in the list of real numbers. Thus, our hypothetical one-to-one correspondence from the nonnegative integers to the reals between 0 and 1 is not one to one. We have shown there is at least one real number in that range, namely r, that is not associated with any integer.

EXERCISES

7.11.1: Show that equipotence is an equivalence relation. *Hint*: The hard part is transitivity, showing that if there is a one-to-one correspondence f from S to T, and a one-to-one correspondence g from T to R, then there is a one-to-one correspondence from S to R. This function is the *composition* of f and g, that is, the function that sends each element x in S to $g(f(x))$ in R.

Composition of functions

7.11.2: In the ordering of pairs in Fig. 7.34, what pair is assigned number 100?

7.11.3*: Show that the following sets are countable (have a one-to-one correspondence between them and the natural numbers):

a) The set of perfect squares
b) The set of triples (i, j, k) of natural numbers
c) The set of powers of 2
d) The set of finite sets of natural numbers

7.11.4**: Show that $\mathbf{P(N)}$, the power set of the natural numbers, has the same cardinality as the reals — that is, there is a one-to-one correspondence from $\mathbf{P(N)}$ to the reals between 0 and 1. Note that this conclusion does not contradict Exercise 7.11.3(d), because here we are talking about finite and infinite sets of integers, while there we counted only finite sets. *Hint*: The following construction almost works, but needs to be fixed. Consider the characteristic vector for any set of natural numbers. This vector is an infinite sequence of 0's and 1's. For example, $\{0, 1\}$ has the characteristic vector $1100\cdots$, and the set of odd numbers has the characteristic vector $010101\cdots$. If we put a decimal point in front of a characteristic vector, we have a binary fraction between 0 and 1, which represents a real number. Thus, every set is sent to a real in the range 0 to 1, and every real number in that range can be associated with a set, by turning its binary representation into a characteristic vector. The reason this association is not a one-to-one correspondence is that certain reals have two binary representations. For example, $.11000\cdots$ and $.10111\cdots$ both represent the real number 3/4. However, these sequences as characteristic vectors represent different sets; the first is $\{0, 1\}$ and the second is the set of all integers except 1. You can modify this construction to define a one-to-one correspondence.

7.11.5**: Show that there is a one-to-one correspondence from pairs of reals in the range 0 to 1 to reals in that range. *Hint*: It is not possible to imitate the table of Fig. 7.34 directly. However, we may take a pair of reals, say, (r, s), and combine the infinite decimal fractions for r and s to make a unique new real number t. This number will not be related to r and s by any simple arithmetic expression, but from t, we can recover r and s uniquely. The reader must discover a way to construct the decimal expansion of t from the expansions of r and s.

7.11.6**: Show that whenever a set S contains subsets of all integer sizes $0, 1, \ldots$, then it is an infinite set according to the formal definition of "infinite"; that is, S has a one-to-one correspondence with one of its proper subsets.

7.12 Summary of Chapter 7

You should take away the following points from Chapter 7:

- The concept of a set is fundamental to both mathematics and computer science.
- The common operations on sets such as union, intersection, and difference can be visualized in terms of Venn diagrams.
- Algebraic laws can be used to manipulate and simplify expressions involving sets and operations on sets.

- Linked lists, characteristic vectors, and hash tables provide three basic ways to represent sets. Linked lists offer the greatest flexibility for most set operations but are not always the most efficient. Characteristic vectors provide the greatest speed for certain set operations but can be used only when the universal set is small. Hash tables are often the method of choice, providing both economy of representation and speed of access.
- (Binary) relations are sets of pairs. A function is a relation in which there is at most one tuple with a given first component.
- A one-to-one correspondence between two sets is a function that associates a unique element of the second set with each element of the first, and vice versa.
- There are a number of significant properties of binary relations: reflexivity, transitivity, symmetry, and asymmetry are among the most important.
- Partial orders, total orders, and equivalence relations are important special cases of binary relations.
- Infinite sets are those sets that have a one-to-one correspondence with one of their proper subsets.
- Some infinite sets are "countable," that is, they have a one-to-one correspondence with the integers. Other infinite sets, such as the reals, are not countable.
- The data structures and operations defined on sets and relations in this chapter will be used in many different ways in the remainder of this book.

7.13 Bibliographic Notes for Chapter 7

Halmos [1974] provides a good introduction to set theory. Hashing techniques were first developed in the 1950's, and Peterson [1957] covers the early techniques. Knuth [1973] and Morris [1968] contain additional material on hashing techniques. Reingold [1972] discusses the computational complexity of basic set operations. The theory of infinite sets was developed by Cantor [1915].

Cantor, G. [1915]. "Contributions to the founding of the theory of transfinite numbers," reprinted by Dover Press, New York.

Halmos, P. R. [1974]. *Naive Set Theory*, Springer-Verlag, New York.

Knuth, D. E. [1973]. *The Art of Computer Programming*, Vol. III, *Sorting and Searching*, Addison-Wesley, Reading, Mass.

Morris, R. [1968]. "Scatter storage techniques," *Comm. ACM* **11**:1, pp. 35–44.

Peterson, W. W. [1957]. "Addressing for random access storage," *IBM J. Research and Development* **1**:7, pp. 130–146.

Reingold, E. M. [1972]. "On the optimality of some set algorithms," *J. ACM* **19**:4, pp. 649–659.

CHAPTER 5

The Tree Data Model

There are many situations in which information has a hierarchical or nested structure like that found in family trees or organization charts. The abstraction that models hierarchical structure is called a *tree* and this data model is among the most fundamental in computer science. It is the model that underlies several programming languages, including Lisp.

Trees of various types appear in many of the chapters of this book. For instance, in Section 1.3 we saw how directories and files in some computer systems are organized into a tree structure. In Section 2.8 we used trees to show how lists are split recursively and then recombined in the merge sort algorithm. In Section 3.7 we used trees to illustrate how simple statements in a program can be combined to form progressively more complex statements.

5.1 What This Chapter Is About

The following themes form the major topics of this chapter:

- The terms and concepts related to trees (Section 5.2).
- The basic data structures used to represent trees in programs (Section 5.3).
- Recursive algorithms that operate on the nodes of a tree (Section 5.4).
- A method for making inductive proofs about trees, called structural induction, where we proceed from small trees to progressively larger ones (Section 5.5).
- The binary tree, which is a variant of a tree in which nodes have two "slots" for children (Section 5.6).
- The binary search tree, a data structure for maintaining a set of elements from which insertions and deletions are made (Sections 5.7 and 5.8).

- The priority queue, which is a set to which elements can be added, but from which only the maximum element can be deleted at any one time. An efficient data structure, called a partially ordered tree, is introduced for implementing priority queues, and an $O(n \log n)$ algorithm, called heapsort, for sorting n elements is derived using a balanced partially ordered tree data structure, called a heap (Sections 5.9 and 5.10).

5.2 Basic Terminology

Nodes and edges

Trees are sets of points, called *nodes*, and lines, called *edges*. An edge connects two distinct nodes. To be a tree, a collection of nodes and edges must satisfy certain properties; Fig. 5.1 is an example of a tree.

Root

1. In a tree, one node is distinguished and called the *root*. The root of a tree is generally drawn at the top. In Fig. 5.1, the root is n_1.

Parent and child

2. Every node c other than the root is connected by an edge to some one other node p called the *parent* of c. We also call c a *child* of p. We draw the parent of a node above that node. For example, in Fig. 5.1, n_1 is the parent of n_2, n_3, and n_4, while n_2 is the parent of n_5 and n_6. Said another way, n_2, n_3, and n_4 are children of n_1, while n_5 and n_6 are children of n_2.

All nodes are connected to the root

3. A tree is *connected* in the sense that if we start at any node n other than the root, move to the parent of n, to the parent of the parent of n, and so on, we eventually reach the root of the tree. For instance, starting at n_7, we move to its parent, n_4, and from there to n_4's parent, which is the root, n_1.

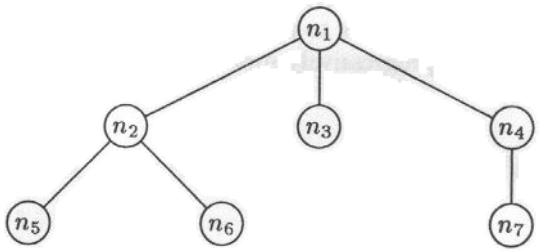


Fig. 5.1. Tree with seven nodes.

An Equivalent Recursive Definition of Trees

It is also possible to define trees recursively with an inductive definition that constructs larger trees out of smaller ones.

BASIS. A single node n is a tree. We say that n is the root of this one-node tree.

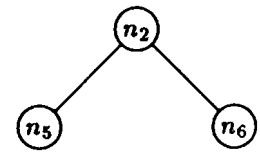
INDUCTION. Let r be a new node and let $T_1, T_2, \ldots, T_k$ be one or more trees with roots $c_1, c_2, \ldots, c_k$, respectively. We require that no node appear more than once in the T_i's; and of course r, being a "new" node, cannot appear in any of these trees. We form a new tree T from r and $T_1, T_2, \ldots, T_k$ as follows:

Pa

a) Make r the root of tree T.

b) Add an edge from r to each of $c_1, c_2, \ldots, c_k$, thereby making each of these nodes a child of the root r. Another way to view this step is that we have made r the parent of each of the roots of the trees $T_1, T_2, \ldots, T_k$.

✦ **Example 5.1.** We can use this recursive definition to construct the tree in Fig. 5.1. This construction also verifies that the structure in Fig. 5.1 is a tree. The nodes n_5 and n_6 are each trees themselves by the basis rule, which says that a single node can be considered a tree. Then we can apply the inductive rule to create a new tree with n_2 as the root r, and the tree T_1, consisting of n_5 alone, and the tree T_2, consisting of n_6 alone, as children of this new root. The nodes c_1 and c_2 are n_5 and n_6, respectively, since these are the roots of the trees T_1 and T_2. As a result, we can conclude that the structure



is a tree; its root is n_2.

Similarly, n_7 alone is a tree by the basis, and by the inductive rule, the structure

is a tree; its root is n_4.

Node n_3 by itself is a tree. Finally, if we take the node n_1 as r, and n_2, n_3, and n_4 as the roots of the three trees just mentioned, we create the structure in Fig. 5.1, verifying that it indeed is a tree. ✦

Paths, Ancestors, and Descendants

The parent-child relationship can be extended naturally to ancestors and descendants. Informally, the ancestors of a node are found by following the unique path from the node to its parent, to its parent's parent, and so on. Strictly speaking, a node is also its own ancestor. The descendant relationship is the inverse of the ancestor relationship, just as the parent and child relationships are inverses of each other. That is, node d is a descendant of node a if and only if a is an ancestor of d.

More formally, suppose $m_1, m_2, \ldots, m_k$ is a sequence of nodes in a tree such that m_1 is the parent of m_2, which is the parent of m_3, and so on, down to m_{k-1}, which is the parent of m_k. Then $m_1, m_2, \ldots, m_k$ is called a *path* from m_1 to m_k in the tree. The *length* of the path is $k-1$, one less than the number of nodes on the path. Note that a path may consist of a single node (if $k = 1$), in which case the length of the path is 0.

Path length

✦ **Example 5.2.** In Fig. 5.1, n_1, n_2, n_6 is a path of length 2 from the root n_1 to the node n_6; n_1 is a path of length zero from n_1 to itself. ✦

Proper ancestor and descendant

If $m_1, m_2, \ldots, m_k$ is a path in a tree, node m_1 is called an *ancestor* of m_k and node m_k a *descendant* of m_1. If the path is of length 1 or more, then m_1 is called a *proper* ancestor of m_k and m_k a *proper* descendant of m_1. Again, remember that the case of a path of length 0 is possible, in which case the path lets us conclude that m_1 is an ancestor of itself and a descendant of itself, although not a proper ancestor or descendant. The root is an ancestor of every node in a tree and every node is a descendant of the root.

✦ **Example 5.3.** In Fig. 5.1, all seven nodes are descendants of n_1, and n_1 is an ancestor of all nodes. Also, all nodes but n_1 itself are proper descendants of n_1, and n_1 is a proper ancestor of all nodes in the tree but itself. The ancestors of n_5 are n_5, n_2, and n_1. The descendants of n_4 are n_4 and n_7. ✦

Sibling

Nodes that have the same parent are sometimes called *siblings*. For example, in Fig. 5.1, nodes n_2, n_3, and n_4 are siblings, and n_5 and n_6 are siblings.

Subtrees

In a tree T, a node n, together with all of its proper descendants, if any, is called a *subtree* of T. Node n is the root of this subtree. Notice that a subtree satisfies the three conditions for being a tree: it has a root, all other nodes in the subtree have a unique parent in the subtree, and by following parents from any node in the subtree, we eventually reach the root of the subtree.

✦ **Example 5.4.** Referring again to Fig. 5.1, node n_3 by itself is a subtree, since n_3 has no descendants other than itself. As another example, nodes n_2, n_5, and n_6 form a subtree, with root n_2, since these nodes are exactly the descendants of n_2. However, the two nodes n_2 and n_6 by themselves do not form a subtree without node n_5. Finally, the entire tree of Fig. 5.1 is a subtree of itself, with root n_1. ✦

Leaves and Interior Nodes

A *leaf* is a node of a tree that has no children. An *interior node* is a node that has one or more children. Thus, every node of a tree is either a leaf or an interior node, but not both. The root of a tree is normally an interior node, but if the tree consists of only one node, then that node is both the root and a leaf.

✦ **Example 5.5.** In Fig. 5.1, the leaves are n_5, n_6, n_3, and n_7. The nodes n_1, n_2, and n_4 are interior. ✦

Height and Depth

Level

In a tree, the *height* of a node n is the length of a longest path from n to a leaf. The *height of the tree* is the height of the root. The *depth*, or *level*, of a node n is the length of the path from the root to n.

✦ **Example 5.6.** In Fig. 5.1, node n_1 has height 2, n_2 has height 1, and leaf n_3 has height 0. In fact, any leaf has height 0. The tree in Fig. 5.1 has height 2. The depth of n_1 is 0, the depth of n_2 is 1, and the depth of n_5 is 2. ✦

Ordered Trees

Optionally, we can assign a left-to-right order to the children of any node. For example, the order of the children of n_1 in Fig. 5.1 is n_2 leftmost, then n_3, then n_4. This left-to-right ordering can be extended to order all the nodes in a tree. If m and n are siblings and m is to the left of n, then all of m's descendants are to the left of all of n's descendants.

✦ **Example 5.7.** In Fig. 5.1, the nodes of the subtree rooted at n_2 – that is, n_2, n_5, and n_6 — are all to the left of the nodes of the subtrees rooted at n_3 and n_4. Thus, n_2, n_5, and n_6 are all to the left of n_3, n_4, and n_7. ✦

In a tree, take any two nodes x and y neither of which is an ancestor of the other. As a consequence of the definition of "to the left," one of x and y will be to the left of the other. To tell which, follow the paths from x and y toward the root. At some point, perhaps at the root, perhaps lower, the paths will meet at some node z as suggested by Fig. 5.2. The paths from x and y reach z from two different nodes m and n, respectively; it is possible that $m = x$ and/or $n = y$, but it must be that $m \neq n$, or else the paths would have converged somewhere below z.

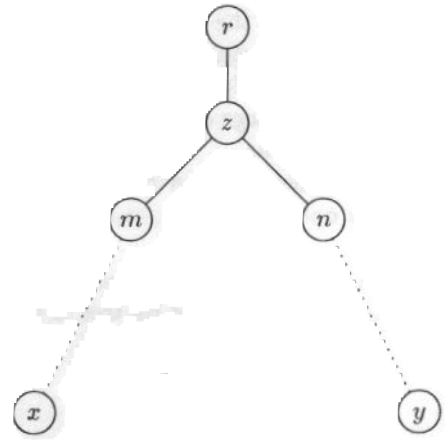


Fig. 5.2. Node x is to the left of node y.

Suppose m is to the left of n. Then since x is in the subtree rooted at m and y is in the subtree rooted at n, it follows that x is to the left of y. Similarly, if m were to the right of n, then x would be to the right of y.

✦ **Example 5.8.** Since no leaf can be an ancestor of another leaf, it follows that all leaves can be ordered "from the left." For instance, the order of the leaves in Fig. 5.1 is n_5, n_6, n_3, n_7. ✦

Labeled Trees

A *labeled* tree is a tree in which a label or value is associated with each node of the tree. We can think of the label as the information associated with a given node. The label can be something as simple, such as a single integer, or complex, such as the text of an entire document. We can change the label of a node, but we cannot change the name of a node.

If the name of a node is not important, we can represent a node by its label. However, the label does not always provide a unique name for a node, since several nodes may have the same label. Thus, many times we shall draw a node with both its label and its name. The following paragraphs illustrate the concept of a labeled tree and offer some samples.

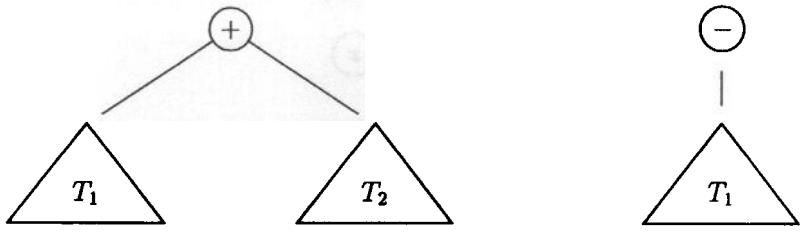
Expression Trees — An Important Class of Trees

Arithmetic expressions are representable by labeled trees, and it is often quite helpful to visualize expressions as trees. In fact, *expression trees*, as they are sometimes called, specify the association of an expression's operands and its operators in a uniform way, regardless of whether the association is required by the placement of parentheses in the expression or by the precedence and associativity rules for the operators involved.

Let us recall the discussion of expressions in Section 2.6, especially Example 2.17, where we gave a recursive definition of expressions involving the usual arithmetic operators. By analogy with the recursive definition of expressions, we can recursively define the corresponding labeled tree. The general idea is that each time we form a larger expression by applying an operator to smaller expressions, we create a new node, labeled by that operator. The new node becomes the root of the tree for the large expression, and its children are the roots of the trees for the smaller expressions.

For instance, we can define the labeled trees for arithmetic expressions with the binary operators $+$, $-$, $\times$, and $/$, and the unary operator $-$, as follows.

BASIS. A single atomic operand (e.g., a variable, an integer, or a real, as in Section 2.6) is an expression, and its tree is a single node, labeled by that operand.



(a) $(E_1 + E_2)$ (b) $(-E_1)$

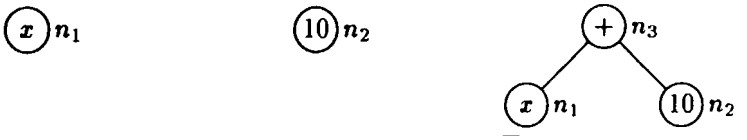
Fig. 5.3. Expression trees for $(E_1 + E_2)$ and $(-E_1)$.

INDUCTION. If E_1 and E_2 are expressions represented by trees T_1 and T_2, respectively, then the expression $(E_1 + E_2)$ is represented by the tree of Fig. 5.3(a), whose root is labeled +. This root has two children, which are the roots of T_1 and T_2, respectively, in that order. Similarly, the expressions $(E_1 - E_2)$, $(E_1 \times E_2)$, and (E_1/E_2) have expression trees with roots labeled $-$, $\times$, and /, respectively, and subtrees T_1 and T_2. Finally, we may apply the unary minus operator to one expression, E_1. We introduce a root labeled $-$, and its one child is the root of T_1; the tree for $(-E_1)$ is shown in Fig. 5.3(b).

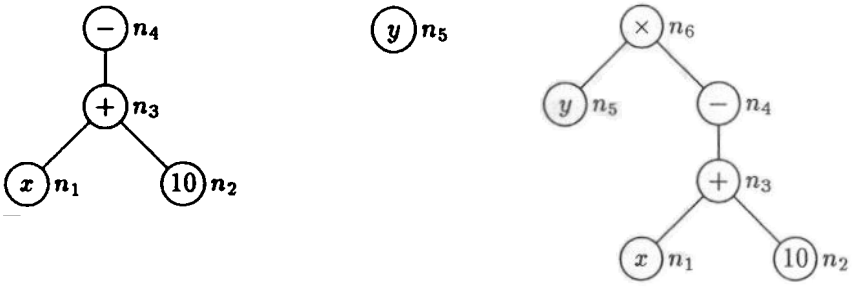
✦ **Example 5.9.** In Example 2.17 we discussed the recursive construction of a sequence of six expressions from the basis and inductive rules. These expressions, listed in Fig. 2.16, were

i)	x	*iv)*	$(-(x + 10))$
ii)	10	*v)*	y
iii)	$(x + 10)$	*vi)*	$\big(y \times (-(x + 10))\big)$

Expressions (*i*), (*ii*), and (*v*) are single operands, and so the basis rule tells us that the trees of Fig. 5.4(a), (b), and (e), respectively, represent these expressions. Note that each of these trees consists of a single node to which we have given a name — n_1, n_2, and n_5, respectively — and a label, which is the operand in the circle.



(a) For x. (b) For 10. (c) For $(x + 10)$.



(d) For $(-(x + 10))$. (e) For y. (f) For $(y \times (-(x + 10)))$.

Fig. 5.4. Construction of expression trees.

Expression (*iii*) is formed by applying the operator + to the operands x and 10, and so we see in Fig. 5.4(c) the tree for this expression, with root labeled +, and the roots of the trees in Fig. 5.4(a) and (b) as its children. Expression (*iv*) is

formed by applying unary $-$ to expression (*iii*), so that the tree for $\big(-(x+10)\big)$, shown in Fig. 5.4(d), has root labeled $-$ above the tree for $(x+10)$. Finally, the tree for the expression $\Big(y \times \big(-(x+10)\big)\Big)$, shown in Fig. 5.4(f), has a root labeled $\times$, whose children are the roots of the trees of Fig. 5.4(e) and (d), in that order. ✦

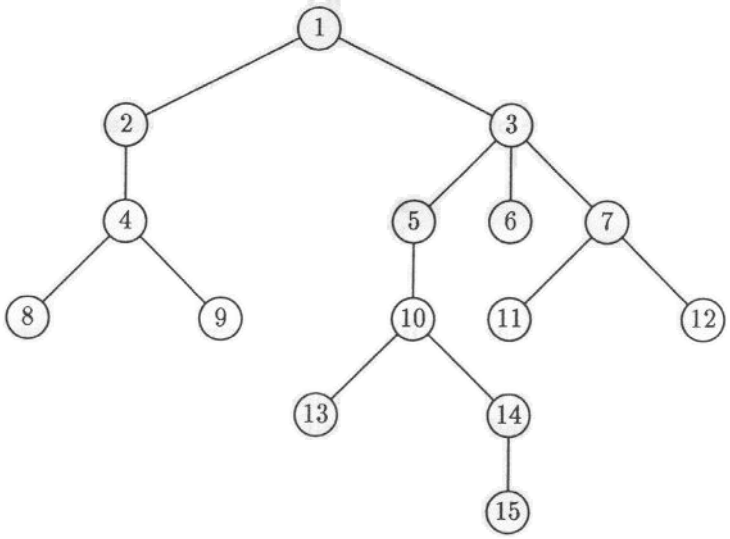


Fig. 5.5. Tree for Exercise 5.2.1.

EXERCISES

5.2.1: In Fig. 5.5 we see a tree. Tell what is described by each of the following phrases:

a) The root of the tree
b) The leaves of the tree
c) The interior nodes of the tree
d) The siblings of node 6
e) The subtree with root 5
f) The ancestors of node 10
g) The descendants of node 10
h) The nodes to the left of node 10
i) The nodes to the right of node 10
j) The longest path in the tree
k) The height of node 3
l) The depth of node 13
m) The height of the tree

5.2.2: Can a leaf in a tree ever have any (a) descendants? (b) proper descendants?

5.2.3: Prove that in a tree no leaf can be an ancestor of another leaf.

5.2.4*: Prove that the two definitions of trees in this section are equivalent. *Hint*: To show that a tree according the nonrecursive definition is a tree according the recursive definition, use induction on the number of nodes in the tree. In the opposite direction, use induction on the number of rounds used in the recursive definition.

5.2.5: Suppose we have a graph consisting of four nodes, r, a, b, and c. Node r is an isolated node and has no edges connecting it. The remaining three nodes form a cycle; that is, we have an edge connecting a and b, an edge connecting b and c, and an edge connecting c and a. Why is this graph not a tree?

5.2.6: In many kinds of trees, there is a significant distinction between the interior nodes and the leaves (or rather the labels of these two kinds of nodes). For example, in an expression tree, the interior nodes represent operators, and the leaves represent atomic operands. Give the distinction between interior nodes and leaves for each of the following kinds of trees:

a) Trees representing directory structures, as in Section 1.3
b) Trees representing the splitting and merging of lists for merge sort, as in Section 2.8
c) Trees representing the structure of a function, as in Section 3.7

5.2.7: Give expression trees for the following expressions. Note that, as is customary with expressions, we have omitted redundant parentheses. You must first restore the proper pairs of parentheses, using the customary rules for precedence and associativity of operators.

a) $(x + 1) \times (x - y + 4)$
b) $1 + 2 + 3 + 4 + 5 + 6$
c) $9 \times 8 + 7 \times 6 + 5$

5.2.8: Show that if x and y are two distinct nodes in an ordered tree, then exactly one of the following conditions must hold:

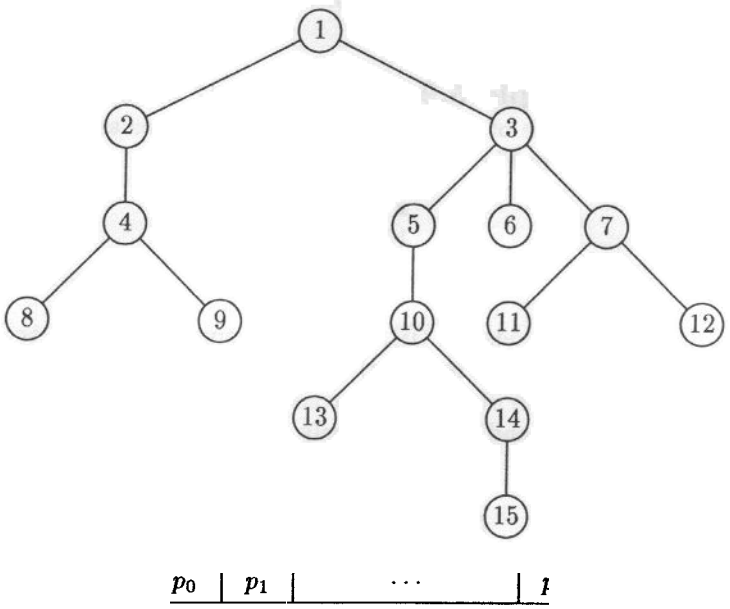
a) x is a proper ancestor of y
b) x is a proper descendant of y
c) x is to the left of y
d) x is to the right of y

5.3 Data Structures for Trees

Many data structures can be used to represent trees. Which one we should use depends on the particular operations we want to perform. As a simple example, if all we ever want to do is to locate the parents of nodes, then we can represent each node by a structure consisting of a label plus a pointer to the structure representing the parent of that node.

As a general rule, the nodes of a tree can be represented by structures in which the fields link the nodes together in a manner similar to the way in which the nodes are connected in the abstract tree; the tree itself can be represented by a pointer to the root's structure. Thus, when we talk about representing trees, we are primarily interested in how the nodes are represented.

One distinction in representations concerns where the structures for the nod "live" in the memory of the computer. In C, we can create the space for stru tures for nodes by using the function **malloc** from the standard library **stdlib.** in which case nodes "float" in memory and are accessible only through pointer Alternatively, we can create an array of structures and use elements of the array



In C this data structure can be represented by the type declaration

```
typedef struct NODE *pNODE;
struct NODE {
    int info;
    pNODE children[BF];
};
```

Here, the field **info** represents the information that constitutes the label of a no and **BF** is the constant defined to be the branching factor. We shall see ma variants of this declaration throughout this chapter.

In this and most other data structures for trees, we represent a tree by a point to the root node. Thus, **pNODE** also serves as the type of a tree. We could, in fac use the type **TREE** in place of **pNODE**, and we shall adopt that convention when talk about binary trees starting in Section 5.6. However, for the moment, we sha use the name **pNODE** for the type "pointer to node," since in some data structure pointers to nodes are used for other purposes besides representing trees.

The array-of-pointers representation allows us to access the ith child of a node in $O(1)$ time. This representation, however, is very wasteful of space whe only a few nodes in the tree have many children. In this case, most of the pointe in the arrays will be **NULL**.

Try to Remember Trie

The term "trie" comes from the middle of the word "retrieval." It was originally intended to be pronounced "tree." Fortunately, common parlance has switched to the distinguishing pronunciation "try."

Trie

✦ **Example 5.10.** A tree can be used to represent a collection of words in a way that makes it quite efficient to check whether a given sequence of characters is a valid word. In this type of tree, called a *trie*, each node except the root has an associated letter. The string of characters represented by a node n is the sequence of letters along the path from the root to n. Given a set of words, the trie consists of nodes for exactly those strings of characters that are prefixes of some word in the set. The label of a node consists of the letter represented by the node and also a Boolean telling whether or not the string from the root to that node forms a complete word; we shall use for the Boolean the integer 1 if so and 0 if not.[1]

For instance, suppose our "dictionary" consists of the four words `he`, `hers`, `his`, `she`. A trie for these words is shown in Fig. 5.7. To determine whether the word `he` is in the set, we start at the root n_1, move to the child n_2 labeled `h`, and then from that node move to its child n_4 labeled `e`. Since these nodes all exist in the tree, and n_4 has 1 as part of its label, we conclude that `he` is in the set.

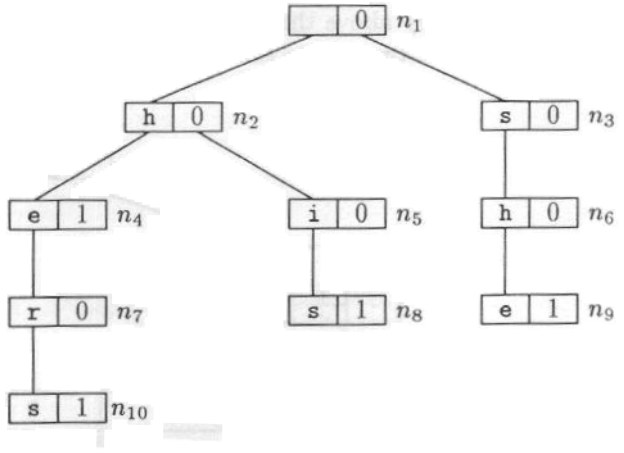


Fig. 5.7. Trie for words `he`, `hers`, `his`, and `she`.

As another example, suppose we want to determine whether `him` is in the set. We follow the path from the root to n_2 to n_5, which represents the prefix `hi`; but at n_5 we find no child corresponding to the letter `m`. We conclude that `him` is not in the set. Finally, if we search for the word `her`, we find our way from the root to node n_7. That node exists but does not have a 1. We therefore conclude that `her` is not in the set, although it is a proper prefix of a word, `hers`, in the set.

[1] In the previous section we acted as if the label was a single value. However, values can be of any type, and labels can be structures consisting of two or more fields. In this case, the label has one field that is a letter and a second that is an integer that is either 0 or 1.

Nodes in a trie have a branching factor equal to the number of different characters in the alphabet from which the words are formed. For example, if we do not distinguish between upper- and lower-case, and words contain no special characters such as apostrophes, then we can take the branching factor to be 26. The type of a node, including the two label fields, can be defined as in Fig. 5.8. In the array `children`, we assume that the letter `a` is represented by index 0, the letter `b` by index 1, and so on.

```
typedef struct NODE *pNODE;
struct NODE {
    char letter;
    int isWord;
    pNODE children[BF];
};
```

Fig. 5.8. Definition of an alphabetic trie.

The abstract trie of Fig. 5.7 can be represented by the data structure of Fig. 5.9. We represent nodes by showing the first two fields, `letter` and `isWord`, along with those elements of the array `children` that have non-`NULL` pointers. In the `children` array, for each non-`NULL` element, the letter indexing the array is shown in the entry above the pointer to the child, but that letter is not actually present in the structure. Note that the `letter` field of the root is irrelevant. ✦

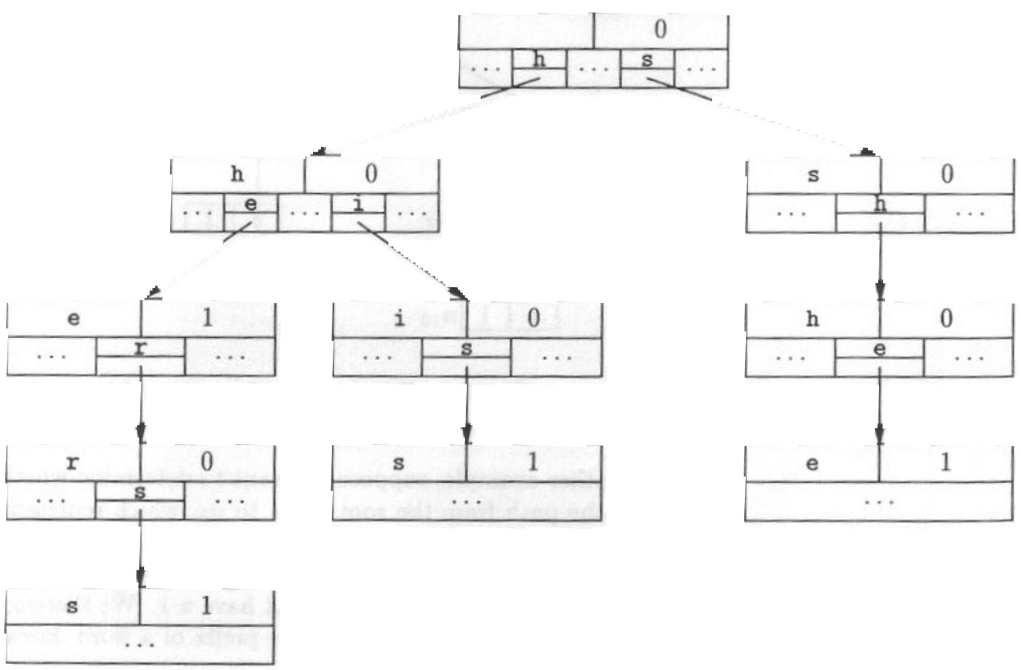


Fig. 5.9. Data structure for the trie of Fig. 5.7.

Leftmost-Child–Right-Sibling Representation of Trees

Using arrays of pointers for nodes is not necessarily space-efficient, because in typical cases, the great majority of pointers will be NULL. That is certainly the case in Fig. 5.9, where no node has more than two non-NULL pointers. In fact, if we think about it, we see that the number of pointers in any trie based on a 26-letter alphabet will have 26 times as many spaces for pointers as there are nodes. Since no node can have two parents and the root has no parent at all, it follows that among N nodes there are only $N - 1$ non-NULL pointers; that is, less than one out of 26 pointers is useful.

One way to overcome the space inefficiency of the array-of-pointers representation of a tree is to use linked lists to represent the children of nodes. The space occupied by a linked list for a node is proportional to the number of children of that node. There is, however, a time penalty with this representation; accessing the ith child takes $O(i)$ time, because we must traverse a list of length $i - 1$ to get to the ith node. In comparison, we can get to the ith child in $O(1)$ time, independent of i, using an array of pointers to the children.

In the representation of trees called *leftmost-child–right-sibling*, we put into each node a pointer only to its leftmost child; a node does not have pointers to any of its other children. To find the second and subsequent children of a node n, we create a linked list of those children, with each child c pointing to the child of n immediately to the right of c. That node is called the *right sibling* of c.

Right sibling

✦ **Example 5.11.** In Fig. 5.1, n_3 is the right sibling of n_2, n_4 is the right sibling of n_3, and n_4 has no right sibling. We would find the children of n_1 by following its leftmost-child pointer to n_2, then the right-sibling pointer to n_3, and then the right-sibling pointer of n_3 to n_4. There, we would find a NULL right-sibling pointer and know that n_1 has no more children.

Figure 5.10 contains a sketch of the leftmost-child–right-sibling representation for the tree in Fig. 5.1. The downward arrows are the leftmost-child links; the sideways arrows are the right-sibling links. ✦

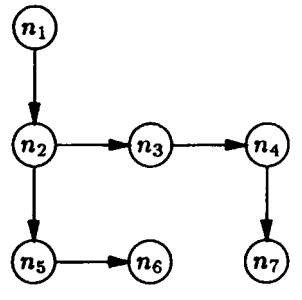


Fig. 5.10. Leftmost-child–right-sibling representation for the tree in Fig. 5.1.

In a leftmost-child–right-sibling representation of a tree, nodes can be defined as follows:

```
typedef struct NODE *pNODE;
struct NODE {
    int info;
    pNODE leftmostChild, rightSibling;
};
```

The field **info** holds the label associated with the node and it can have any type. The fields **leftmostChild** and **rightSibling** point to the leftmost child and right sibling of the node in question. Note that while **leftmostChild** gives information about the node itself, the field **rightSibling** at a node is really part of the linked list of children of that node's parent.

✦ **Example 5.12.** Let us represent the trie of Fig. 5.7 in the leftmost-child–right-sibling form. First, the type of nodes is

```
typedef struct NODE *pNODE;
struct NODE {
    char letter;
    int isWord;
    pNODE leftmostChild, rightSibling;
};
```

The first two fields represent information, according to the scheme described in Example 5.10. The trie of Fig. 5.7 is represented by the data structure shown in Fig. 5.11. Notice that each leaf has a **NULL** leftmost-child pointer, and each rightmost child has a **NULL** right-sibling pointer.

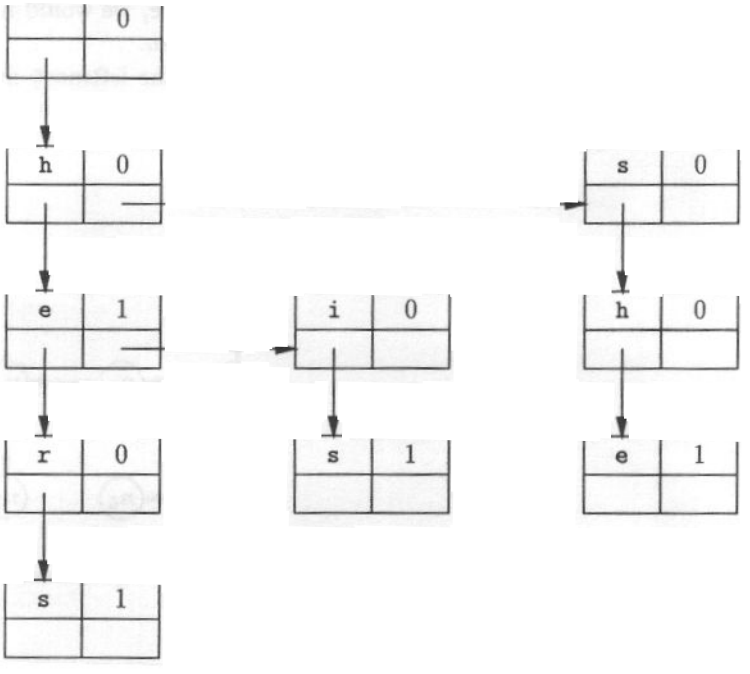


Fig. 5.11. Leftmost-child–right-sibling representation for the trie of Fig. 5.7.

As an example of how one uses the leftmost-child–right-sibling representation, we see in Fig. 5.12 a function **seek(let, n)** that takes a letter *let* and a pointer to a node n as arguments. It returns a pointer to the child of n that has *let* in its **letter** field, and it returns **NULL** if there is no such node. In the while loop of Fig. 5.12, each child of n is examined in turn. We reach line (6) if either *let* is found or we have examined all the children and thus have fallen out of the loop. In either case, c holds the correct value, a pointer to the child holding *let* if there is one, and **NULL** if not.

Notice that **seek** takes time proportional to the number of children that must be examined until we find the child we are looking for, and if we never find it, then the time is proportional to the number of children of node n. In comparison, using the array-of-pointers representation of trees, **seek** could simply return the value of the array element for letter *let*, taking $O(1)$ time. ✦

```
    pNODE seek(char let, pNODE n)
    {
(1)     c = n->leftmostChild;
(2)     while (c != NULL)
(3)         if (c->letter == let)
(4)             break;
            else
(5)             c = c->rightSibling;
(6)     return c;
    }
```

Fig. 5.12. Finding the child for a desired letter.

Parent Pointers

Sometimes, it is useful to include in the structure for each node a pointer to the parent. The root has a **NULL** parent pointer. For example, the structure of Example 5.12 could become

```
typdef struct NODE *pNODE
struct NODE {
    char letter;
    int isWord;
    pNODE leftmostChild, rightSibling, parent;
};
```

With this structure, it becomes possible to determine what word a given node represents. We repeatedly follow parent pointers until we come to the root, which we can identify because it alone has the value of **parent** equal to **NULL**. The **letter** fields encountered along the way spell the word, backward.

EXERCISES

5.3.1: For each node in the tree of Fig. 5.5, indicate the leftmost child and right sibling.

Comparison of Tree Representations

We summarize the relative merits of the array-of-pointers (trie) and the leftmost-child–right-sibling representations for trees:

- The array-of-pointers representation offers faster access to children, requiring $O(1)$ time to reach any child, no matter how many children there are.
- The leftmost-child–right-sibling representation uses less space. For instance, in our running example of the trie of Fig. 5.7, each node contains 26 pointers in the array representation and two pointers in the leftmost-child–right-sibling representation.
- The leftmost-child–right-sibling representation does not require that there be a limit on the branching factor of nodes. We can represent trees with any branching factor, without changing the data structure. However, if we use the array-of-pointers representation, once we choose the size of the array, we cannot represent a tree with a larger branching factor.

5.3.2: Represent the tree of Fig. 5.5

a) As a trie with branching factor 3

b) By leftmost-child and right-sibling pointers

How many bytes of memory are required by each representation?

5.3.3: Consider the following set of singular personal pronouns in English: I, my, mine, me, you, your, yours, he, his, him, she, her, hers. Augment the trie of Fig. 5.7 to include all thirteen of these words.

5.3.4: Suppose that a complete dictionary of English contains 2,000,000 words and that the number of prefixes of words — that is, strings of letters that can be extended at the end by zero or more additional letters to form a word — is 10,000,000.

a) How many nodes would a trie for this dictionary have?

b) Suppose that we use the structure in Example 5.10 to represent nodes. Let pointers require four bytes, and suppose that the information fields `letter` and `isWord` each take one byte. How many bytes would the trie require?

c) Of the space calculated in part (b), how much is taken up by `NULL` pointers?

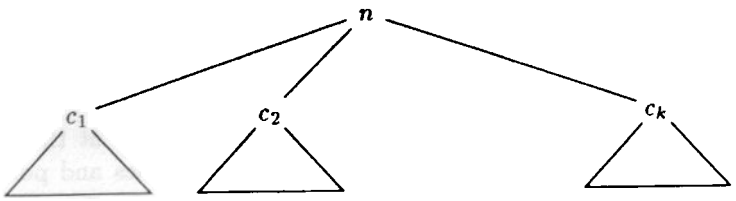
5.3.5: Suppose we represent the dictionary described in Exercise 5.3.4 by using the structure of Example 5.12 (a leftmost-child–right-sibling representation). Under the same assumptions about space required by pointers and information fields as in Exercise 5.3.4(b), how much space does the tree for the dictionary require? What portion of that space is `NULL` pointers?

Lowest common ancestor

5.3.6: In a tree, a node c is the *lowest common ancestor* of nodes x and y if c is an ancestor of both x and y, and no proper descendant of c is an ancestor of x and y. Write a program that will find the lowest common ancestor of any pair of nodes in a given tree. What is a good data structure for trees in such a program?

5.4 Recursions on Trees

The usefulness of trees is highlighted by the number of recursive operations on trees that can be written naturally and cleanly. Figure 5.13 suggests the general form of a recursive function $F(n)$ that takes a node n of a tree as argument. F first performs some steps (perhaps none), which we represent by action A_0. Then F calls itself on the first child, c_1, of n. During this recursive call, F will "explore" the subtree rooted at c_1, doing whatever it is F does to a tree. When that call returns to the call at node n, some other action — say A_1 — is performed. Then F is called on the second child of n, resulting in exploration of the second subtree, and so on, with actions at n alternating with calls to F on the children of n.



(a) General form of a tree.

```
F(n)
{
    action A0;
    F(c1);
    action A1;
    F(c2);
    action A2;
    ...
    F(ck);
    action Ak;
}
```

(b) General form of recursive function $F(n)$ on a tree.

Fig. 5.13. A recursive function on a tree.

Preorder

✦ **Example 5.13.** A simple recursion on a tree produces what is known as the *preorder listing* of the node labels of the tree. Here, action A_0 prints the label of the node, and the other actions do nothing other than some "bookkeeping" operations that enable us to visit each child of a given node. The effect is to print the labels as we would first meet them if we started at the root and circumnavigated the tree, visiting all the nodes in a counterclockwise tour. Note that we print the label of a node only the first time we visit that node. The circumnavigation is suggested by the arrow in Fig. 5.14, and the order in which the nodes are visited is $+a+*-b-c-*d*+$. The preorder listing is the sequence of node labels $+a*-bcd$.

Let us suppose that we use a leftmost-child–right-sibling representation of nodes in an expression tree, with labels consisting of a single character. The label of an

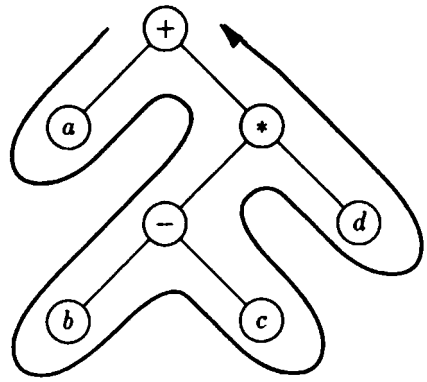


Fig. 5.14. An expression tree and its circumnavigation.

interior node is the arithmetic operator at that node, and the label of a leaf is a letter standing for an operand. Nodes and pointers to nodes can be defined as follows:

```
typedef struct NODE *pNODE;
struct NODE {
    char nodeLabel;
    pNODE leftmostChild, rightSibling;
};
```

The function **preorder** is shown in Fig. 5.15. In the explanation that follows, it is convenient to think of pointers to nodes as if they were the nodes themselves.

```
                void preorder(pNODE n)
                {
                    pNODE c; /* a child of node n */
(1)                 printf("%c\n", n->nodeLabel);
(2)                 c = n->leftmostChild;
(3)                 while (c != NULL) {
(4)                     preorder(c);
(5)                     c = c->rightSibling;
                    }
                }
```

Fig. 5.15. Preorder traversal function.

Action "A_0" consists of the following parts of the program in Fig. 5.15:

1. Printing the label of node n, at line (1),
2. Initializing c to be the leftmost child of n, at line (2), and
3. Performing the first test for `c != NULL`, at line (3).

Line (2) initializes a loop in which c becomes each child of n, in turn. Note that if n is a leaf, then c is assigned the value NULL at line (2).

We go around the while-loop of lines (3) to (5) until we run out of children of n. For each child, we call the function **preorder** recursively on that child, at line (4), and then advance to the next child, at line (5). Each of the actions A_i, for $i \geq 1$, consists of line (5), which moves c through the children of n, and the test at line (3) to see whether we have exhausted the children. These actions are for bookkeeping only; in comparison, line (1) in action A_0 does the significant step, printing the label.

The sequence of events for calling **preorder** on the root of the tree in Fig. 5.14 is summarized in Fig. 5.16. The character at the left of each line is the label of the node n at which the call of **preorder(n)** is currently being executed. Because no two nodes have the same label, it is convenient here to use the label of a node as its name. Notice that the characters printed are $+a*-bcd$, in that order, which is the same as the order of circumnavigation. ✦

```
      call preorder(+)
(+)       print +
(+)       call preorder(a)
(a)            print a
(+)       call preorder(*)
(*)            print *
(*)            call preorder(−)
(−)                 print −
(−)                 call preorder(b)
(b)                      print b
(−)                 call preorder(c)
(c)                      print c
(*)            call preorder(d)
(d)                 print d
```

Fig. 5.16. Action of recursive function preorder on tree of Fig. 5.14.

Postorder

✦ **Example 5.14.** Another common way to order the nodes of the tree, called *postorder*, corresponds to circumnavigating the tree as in Fig. 5.14 but listing a node the last time it is visited, rather than the first. For instance, in Fig. 5.14, the postorder listing is $abc-d*+$.

To produce a postorder listing of the nodes, the last action does the printing, and so a node's label is printed after the postorder listing function is called on all of its children, in order from the left. The other actions initialize the loop through the children or move to the next child. Note that if a node is a leaf, all we do is list the label; there are no recursive calls.

If we use the representation of Example 5.13 for nodes, we can create postorder listings by the recursive function **postorder** of Fig. 5.17. The action of this function when called on the root of the tree in Fig. 5.14 is shown in Fig. 5.18. The same convention regarding node names is used here as in Fig. 5.16. ✦

```
    void postorder(pNODE n)
    {
        pNODE c; /* a child of node n */
(1)     c = n->leftmostChild;
(2)     while (c != NULL) {
(3)         postorder(c);
(4)         c = c->rightSibling;
        }
(5)     printf("%c\n", n->nodeLabel);
    }
```

Fig. 5.17. Recursive postorder function.

```
    call postorder(+)
(+)     call postorder(a)
(a)         print a
(+)     call postorder(*)
(*)         call postorder(−)
(−)             call postorder(b)
(b)                 print b
(−)             call postorder(c)
(c)                 print c
(−)             print −
(*)         call postorder(d)
(d)             print d
(*)         print *
(+)     print +
```

Fig. 5.18. Action of recursive function **postorder** on tree of Fig. 5.14.

✦ **Example 5.15.** Our next example requires us to perform significant actions among all of the recursive calls on subtrees. Suppose we are given an expression tree with integers as operands, and with binary operators, and we wish to produce the numerical value of the expression represented by the tree. We can do so by executing the following recursive algorithm on the expression tree.

Evaluating an expression tree

BASIS. For a leaf we produce the integer value of the node as the value of the tree.

INDUCTION. Suppose we wish to compute the value of the expression formed by the subtree rooted at some node n. We evaluate the subexpressions for the two subtrees rooted at the children of n; these are the values of the operands for the operator at n. We then apply the operator labeling n to the values of these two subtrees, and we have the value of the entire subtree rooted at n.

We define a pointer to a node and a node as follows:

Prefix and Postfix Expressions

When we list the labels of an expression tree in preorder, we get the *prefix expression* equivalent to the given expression. Similarly, the list of the labels of an expression tree in postorder yields the equivalent *postfix expression*. Expressions in the ordinary notation, where binary operators appear between their operands, are called *infix expressions*. For instance, the expression tree of Fig. 5.14 has the infix expression $a + (b - c) * d$. As we saw in Examples 5.13 and 5.14, the equivalent prefix expression is $+a * -bcd$, and the equivalent postfix expression is $abc - d * +$.

Infix expression

An interesting fact about prefix and postfix notations is that, as long as each operator has a unique number of arguments (e.g., we cannot use the same symbol for binary and unary minus), then no parentheses are ever needed, yet we can still unambiguously group operators with their operands.

We can construct an infix expression from a prefix expression as follows. In the prefix expression, we find an operator that is followed by the required number of operands, with no embedded operators. In the prefix expression $+a * -bcd$, for example, the subexpression $-bc$ is such a string, since the minus sign, like all operators in our running example, takes two operands. We replace this subexpression by a new symbol, say $x = -bc$, and repeat the process of identifying an operator followed by its operands. In our example, we now work with $+a * xd$. At this point we identify the subexpression $y = *xd$ and reduce the remaining string to $+ay$. Now the remaining string is just an instance of an operator and its operands, and so we convert it to the infix expression $a + y$.

We may now reconstruct the remainder of the infix expression by retracing these steps. We observe that the subexpression $y = *xd$ in infix is $x * d$, and so we may substitute for y in $a + y$ to get $a + (x * d)$. Note that in general, parentheses are needed in infix expressions, although in this case, we can omit them because of the convention that $*$ takes precedence over $+$ when grouping operands. Then we substitute for $x = -bc$ the infix expression $b - c$, and so our final expression is $a + \big((b - c) * d\big)$, which is the same as that represented by the tree of Fig. 5.14.

For a postfix expression, we can use a similar algorithm. The only difference is that we look for an operator preceded by the requisite number of operands in order to decompose a postfix expression.

```
typedef struct NODE *pNODE;
struct NODE {
    char op;
    int value;
    pNODE leftmostChild, rightSibling;
};
```

The field `op` will hold either the character for an arithmetic operator, or the character `i`, which stands for "integer" and identifies a node as a leaf. If the node is a leaf, then the `value` field holds the integer represented; `value` is not used at interior nodes.

This notation allows operators with any number of arguments, although we shall write code on the simplifying assumption that all operators are binary. The code appears in Fig. 5.19.

```
    int eval(pNODE n)
    {
        int val1, val2; /* values of first and second subtrees */
(1)     if (n->op) == 'i') /* n points to a leaf */
(2)         return n->value;
        else {/* n points to an interior node */
(3)         val1 = eval(n->leftmostChild);
(4)         val2 = eval(n->leftmostChild->rightSibling);
(5)         switch (n->op) {
(6)             case '+': return val1 + val2;
(7)             case '-': return val1 - val2;
(8)             case '*': return val1 * val2;
(9)             case '/': return val1 / val2;
            }
        }
    }
```

Fig. 5.19. Evaluating an arithmetic expression.

If the node n is a leaf, then the test of line (1) succeeds and we return the integer label of that leaf at line (2). If the node is not a leaf, then we evaluate its left operand at line (3) and its right operand at line (4), storing the results in `val1` and `val2`, respectively. Note in connection with line (4) that the second child of a node n is the right sibling of the leftmost child of the node n. Lines (5) through (9) form a switch statement, in which we decide what the operator at n is and apply the appropriate operation to the values of the left and right operands.

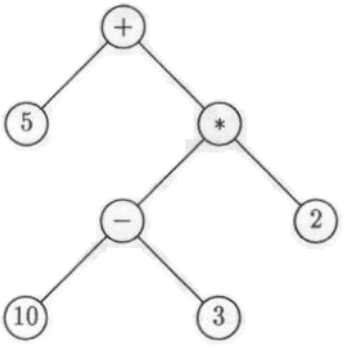


Fig. 5.20. An expression tree with integer operands.

For instance, consider the expression tree of Fig. 5.20. We see in Fig. 5.21 the sequence of calls and returns that are made at each node during the evaluation of this expression. As before, we have taken advantage of the fact that labels are unique and have named nodes by their labels. ✦

✦ **Example 5.16.** Sometimes we need to determine the height of each node in a

```
        call eval(+)
(+)         call eval(5)
 (5)            return 5
(+)         call eval(*)
 (*)            call eval(−)
(−)                 call eval(10)
(10)                    return 10
(−)                 call eval(3)
 (3)                    return 3
(−)                 return 7
 (*)            call eval(2)
 (2)                return 2
 (*)            return 14
(+)         return 19
```

Fig. 5.21. Actions of function `eval` at each node on tree of Fig. 5.20.

tree. The height of a node can be defined recursively by the following function:

Computing the height of a tree

BASIS. The height of a leaf is 0.

INDUCTION. The height of an interior node is 1 greater than the largest of the heights of its children.

We can translate this definition into a recursive program that computes the height of each node into a field `height`:

BASIS. At a leaf, set the height to 0.

INDUCTION. At an interior node, recursively compute the heights of the children, find the maximum, add 1, and store the result into the `height` field.

This program is shown in Fig. 5.22. We assume that nodes are structures of the form

```
typedef struct NODE *pNODE;
struct NODE {
    int height;
    pNODE leftmostChild, rightSibling;
};
```

The function `computeHt` takes a pointer to a node as argument and computes the height of that node in the field `height`. If we call this function on the root of a tree, it will compute the heights of all the nodes of that tree.

At line (1) we initialize the height of n to 0. If n is a leaf, we are done, because the test of line (3) will fail immediately, and so the height of any leaf is computed to be 0. Line (2) sets c to be (a pointer to) the leftmost child of n. As we go around the loop of lines (3) through (7), c becomes each child of n in turn. We recursively compute the height of c at line (4). As we proceed, the value in `n->height` will

Still More Defensive Programming

Several aspects of the program in Fig. 5.19 exhibit a careless programming style that we would avoid were it not our aim to illustrate some points concisely. Specifically, we are following pointers without checking first whether they are `NULL`. Thus, in line (1), n could be `NULL`. We really should begin the program by saying

```
if (n != NULL) /* then do lines (1) to (9) */
else /* print an error message */
```

Even if n is not `NULL`, in line (3) we might find that its `leftmostChild` field is `NULL`, and so we should check whether `n->leftmostChild` is `NULL`, and if so, print an error message and not call `eval`. Similarly, even if the leftmost child of n exists, that node might not have a right sibling, and so before line (4) we need to check that

```
n->leftmostChild->rightSibling != NULL
```

It is also tempting to rely on the assumption that the information contained in the nodes of the tree is correct. For example, if a node is an interior node, it is labeled by a binary operator, and we have assumed it has two children and the pointers followed in lines (3) and (4) cannot possibly be `NULL`. However, it may be possible that the operator label is incorrect. To handle this situation properly, we should add a default case to the switch statement to detect unanticipated operator labels.

As a general rule, relying on the assumption that inputs to programs will always be correct is simplistic at best; in reality, "whatever can go wrong, will go wrong." A program, if it is used more than once, is bound to see data that is not of the form the programmer envisioned. One cannot be too careful in practice – blindly following `NULL` pointers or assuming that input data is always correct are common programming errors.

```
    void computeHt(pNODE n)
    {
        pNODE c;

(1)     n->height = 0;
(2)     c = n->leftmostChild;
(3)     while (c != NULL) {
(4)         computeHt(c);
(5)         if (c->height >= n->height)
(6)             n->height = 1+c->height;
(7)         c = c->rightSibling;
        }
    }
```

Fig. 5.22. Procedure to compute the height of all the nodes of a tree.

be 1 greater than the height of the highest child seen so far, but 0 if we have not

seen any children. Thus, lines (5) and (6) allow us to increase the height of n if we find a new child that is higher than any previous child. Also, for the first child, the test of line (5) will surely be satisfied, and we set `n->height` to 1 more than the height of the first child. When we fall out of the loop because we have seen all the children, `n->height` has been set to 1 more than the maximum height of any of n's children. ✦

EXERCISES

5.4.1: Write a recursive program to count the number of nodes in a tree that is represented by leftmost-child and right-sibling pointers.

5.4.2: Write a recursive program to find the maximum label of the nodes of a tree. Assume that the tree has integer labels, and that it is represented by leftmost-child and right-sibling pointers.

5.4.3: Modify the program in Fig. 5.19 to handle trees containing unary minus nodes.

5.4.4*: Write a recursive program that computes for a tree, represented by leftmost-child and right-sibling pointers, the number of *left-right pairs*, that is, pairs of nodes n and m such that n is to the left of node m. For example, in Fig. 5.20, node 5 is to the left of the nodes labeled $*$, $-$, 10, 3, and 2; node 10 is to the left of nodes 3 and 2; and node $-$ is to the left of node 2. Thus, the answer for this tree is eight pairs. *Hint*: Let your recursive function return two pieces of information when called on a node n: the number of left-right pairs in the subtree rooted at n, and also the number of nodes in the subtree rooted at n.

5.4.5: List the nodes of the tree in Fig. 5.5 (see the Exercises for Section 5.2) in (a) preorder and (b) postorder.

5.4.6: For each of the expressions

i) $(x + y) * (x + z)$

ii) $\big((x - y) * z + (y - w)\big) * x$

iii) $\Big(\big(\big((a * x + b) * x + c\big) * x + d\big) * x + e\Big) * x + f$

do the following:

a) Construct the expression tree.
b) Find the equivalent prefix expression.
c) Find the equivalent postfix expression.

5.4.7: Convert the expression $ab + c * de - /f+$ from postfix to (a) infix and (b) prefix.

5.4.8: Write a function that "circumnavigates" a tree, printing the name of a node each time it is passed.

5.4.9: What are the actions A_0, A_1, and so forth, for the postorder function in Fig. 5.17? ("Actions" are as indicated in Fig. 5.13.)

5.5 Structural Induction

In Chapters 2 and 3 we saw a number of inductive proofs of properties of integers. We would assume that some statement is true about n, or about all integers less than or equal to n, and use this inductive hypothesis to prove the same statement is true about $n + 1$. A similar but not identical form of proof, called "structural induction," is useful for proving properties about trees. Structural induction is analogous to recursive algorithms on trees, and this form of induction is generally the easiest to use when we wish to prove something about trees.

Suppose we want to prove that a statement $S(T)$ is true for all trees T. For a basis, we show that $S(T)$ is true when T consists of a single node. For the induction, we suppose that T is a tree with root r and children $c_1, c_2, \ldots, c_k$, for some $k \geq 1$. Let $T_1, T_2, \ldots, T_k$ be the subtrees of T whose roots are $c_1, c_2, \ldots, c_k$, respectively, as suggested by Fig. 5.23. Then the inductive step is to assume that $S(T_1), S(T_2), \ldots, S(T_k)$ are all true and prove $S(T)$. If we do so, then we can conclude that $S(T)$ is true for all trees T. This form of argument is called *structural induction*. Notice that a structural induction does not make reference to the exact number of nodes in a tree, except to distinguish the basis (one node) from the inductive step (more than one node).

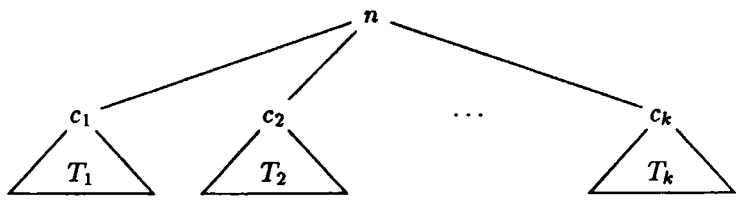


Fig. 5.23. A tree and its subtrees.

```
(1)     if (n->op) == 'i') /* n points to a leaf */
(2)         return n->value;
        else {/* n points to an interior node */
(3)         val1 = eval(n->leftmostChild);
(4)         val2 = eval(n->leftmostChild->rightSibling);
(5)         switch (n->op) {
(6)             case '+': return val1 + val2;
(7)             case '-': return val1 - val2;
(8)             case '*': return val1 * val2;
(9)             case '/': return val1 / val2;
            }
        }
```

Fig. 5.24. The body of the function `eval(n)` from Fig. 5.19.

✦ **Example 5.17.** A structural induction is generally needed to prove the correctness of a recursive program that acts on trees. As an example, let us reconsider the function **eval** of Fig. 5.19, the body of which we reproduce as Fig. 5.24. This function is applied to a tree T by being given a pointer to the root of T as the value of its argument n. It then computes the value of the expression represented by T. We shall prove by structural induction the following statement:

STATEMENT $S(T)$: The value returned by **eval** when called on the root of T equals the value of the arithmetic expression represented by T.

BASIS. For the basis, T consists of a single node. That is, the argument n is a (pointer to a) leaf. Since the **op** field has the value `'i'` when the node represents an operand, the test of line (1) in Fig. 5.24 succeeds, and the value of that operand is returned at line (2).

INDUCTION. Suppose the node n is not a (pointer to a) leaf. The inductive hypothesis is that $S(T')$ is true for each tree T' rooted at one of the children of n. We must use this reasoning to prove $S(T)$ for the tree T rooted at n.

Since our operators are assumed to be binary, n has two subtrees. By the inductive hypothesis, the values of **val1** and **val2** computed at lines (3) and (4) respectively, are the values of the left and right subtrees. Figure 5.25 suggests these two subtrees; **val1** holds the value of T_1 and **val2** holds the value of T_2.

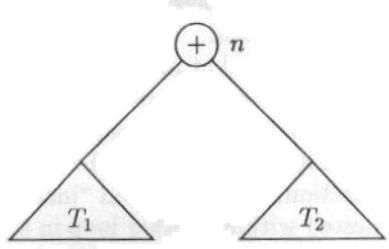


Fig. 5.25. The call *eval*(n) returns the sum of the values of T_1 and T_2.

If we examine the switch statement of lines (5) through (9), we see that whatever operator appears at the root n is applied to the two values *val*1 and *val*2. For example, if the root holds +, as in Fig. 5.25, then at line (5) the value returned is *val*1 + *val*2, as it should be for an expression that is the sum of the expressions of trees T_1 and T_2. We have now completed the inductive step.

We conclude that $S(T)$ holds for all expression trees T, and, therefore, the function **eval** correctly evaluates trees that represent expressions. ✦

✦ **Example 5.18.** Now let us consider the function **computeHt** of Fig. 5.22, the body of which we reproduce as Fig. 5.26. This function takes as argument a (pointer to a) node n and computes the height of n. We shall prove the following statement by structural induction:

```
(1)     n->height = 0;
(2)     c = n->leftmostChild;
(3)     while (c != NULL) {
(4)         computeHt(c);
(5)         if (c->height >= n->height)
(6)             n->height = 1+c->height;
(7)         c = c->rightSibling;
        }
```

Fig. 5.26. The body of the function `computeHt(n)` from Fig. 5.22.

STATEMENT $S(T)$: When **computeHt** is called on a pointer to the root of tree T, the correct height of each node in T is stored in the **height** field of that node.

BASIS. If the tree T is a single node n, then at line (2) of Fig. 5.26, c will be given the value **NULL**, since n has no children. Thus, the test of line (3) fails immediately, and the body of the while-loop is never executed. Since line (1) sets **n->height** to 0, which is the correct value for a leaf, we conclude that $S(T)$ holds when T has a single node.

INDUCTION. Now suppose n is the root of a tree T that is not a single node. Then n has at least one child. We may assume by the inductive hypothesis that when **computeHt(c)** is called at line (4), the correct height is installed in the **height** field of each node in the subtree rooted at c, including c itself. We need to show that the while-loop of lines (3) through (7) correctly sets **n->height** to 1 more than the maximum of the heights of the children of n. To do so, we need to perform another induction, which is nested "inside" the structural induction, just as one loop might be nested within another loop in a program. This induction is an "ordinary" induction, not a structural induction, and its statement is

STATEMENT $S'(i)$: After the loop of lines (3) to (7) has been executed i times, the value of **n->height** is 1 more than the largest of the heights of the first i children of n.

BASIS. The basis is $i = 1$. Since **n->height** is set to 0 outside the loop — at line (1) — and surely no height can be less than 0, the test of line (5) will be satisfied. Line (6) sets **n->height** to 1 more than the height of its first child.

INDUCTION. Assume that $S'(i)$ is true. That is, after i iterations of the loop, **n->height** is 1 larger than the largest height among the first i children. If there is an $(i+1)$st child, then the test of line (3) will succeed and we execute the body an $(i+1)$st time. The test of line (5) compares the new height with the largest of the previous heights. If the new height, **c->height**, is less than 1 plus the largest of the first i heights, no change to **n->height** will be made. That is correct, since the maximum height of the first $i+1$ children is the same as the maximum height of the first i children. However, if the new height is greater than the previous maximum,

A Template for Structural Induction

The following is an outline for building correct structural inductions.

1. Specify the statement $S(T)$ to be proved, where T is a tree.
2. Prove the basis, that $S(T)$ is true whenever T is a tree with a single node.
3. Set up the inductive step by letting T be a tree with root r and $k \geq 1$ subtrees, $T_1, T_2, \ldots, T_k$. State that you assume the inductive hypothesis: that $S(T_i)$ is true for each of the subtrees T_i, $i = 1, 2, \ldots, k$.
4. Prove that $S(T)$ is true under the assumptions mentioned in (3).

then the test of line (5) will succeed, and `n->height` is set to 1 more than the height of the $(i+1)$st child, which is correct.

We can now return to the structural induction. When the test of line (3) fails, we have considered all the children of n. The inner induction, $S'(i)$, tells us that when i is the total number of children, `n->height` is 1 more than the largest height of any child of n. That is the correct height for n. The inductive hypothesis S applied to each of the children of n tells us that the correct height has been stored in each of their `height` fields. Since we just saw that n's height has also been correctly computed, we conclude that all the nodes in T have been assigned their correct height.

We have now completed the inductive step of the structural induction, and we conclude that `computeHt` correctly computes the height of each node of every tree on which it is called. ✦

Why Structural Induction Works

The explanation for why structural induction is a valid proof method is similar to the reason ordinary inductions work: if the conclusion were false, there would be a smallest counterexample, and that counterexample would violate either the basis or the induction. That is, suppose there is a statement $S(T)$ for which we have proved the basis and the structural induction step, yet there are one or more trees for which S is false. Let T_0 be a tree such that $S(T_0)$ is false, but let T_0 have as few nodes as any tree for which S is false.

There are two cases. First, suppose that T_0 consists of a single node. Then $S(T_0)$ is true by the basis, and so this case cannot occur.

The only other possibility is that T_0 has more than one node — say, m nodes — and therefore T_0 consists of a root r with one or more children. Let the trees rooted at these children be $T_1, T_2, \ldots, T_k$. We claim that none of $T_1, T_2, \ldots, T_k$ can have more than $m-1$ nodes. For if one — say T_i — did have m or more nodes, then T_0, which consists of T_i and the root node r, possibly along with other subtrees, would have at least $m+1$ nodes. That contradicts our assumption that T_0 has exactly m nodes.

Now since each of the subtrees $T_1, T_2, \ldots, T_k$ has $m-1$ or fewer nodes, we know that these trees cannot violate S, because we chose T_0 to be as small as any

A Relationship between Structural and Ordinary Induction

There is a sense in which structural induction really offers nothing new. Suppose we have a statement $S(T)$ about trees that we want to prove by structural induction. We could instead prove

STATEMENT $S'(i)$: For all trees T of i nodes, $S(T)$ is true.

$S'(i)$ has the form of an ordinary induction on the integer i, with basis $i = 1$. It can be proved by complete induction, where we assume $S'(j)$ for all $j \leq i$, and prove $S'(i+1)$. This proof, however, would look exactly like the proof of $S(T)$, if we let T stand for an arbitrary tree of $i+1$ nodes.

tree making S false. Thus, we know that $S(T_1), S(T_2), \ldots, S(T_k)$ are all true. The inductive step, which we assume proved, tells us that $S(T_0)$ is also true. Again we contradict the assumption that T_0 violates S.

We have considered the two possible cases, a tree of one node or a tree with more than one node, and have found that in either case, T_0 cannot be a violation of S. Therefore, S has no violations, and $S(T)$ must be true for all trees T.

EXERCISES

5.5.1: Prove by structural induction that

a) The preorder traversal function of Fig. 5.15 prints the labels of the tree in preorder.

b) The postorder function in Fig. 5.17 lists the labels in postorder.

5.5.2*: Suppose that a trie with branching factor b is represented by nodes in the format of Fig. 5.6. Prove by structural induction that if a tree T has n nodes, then there are $1+(b-1)n$ `NULL` pointers among its nodes. How many non-`NULL` pointers are there?

Degree of a node

5.5.3*: The *degree* of a node is the number of children that node has.[2] Prove by structural induction that in any tree T, the number of nodes is 1 more than the sum of the degrees of the nodes.

5.5.4*: Prove by structural induction that in any tree T, the number of leaves is 1 more than the number of nodes that have right siblings.

5.5.5*: Prove by structural induction that in any tree T represented by the leftmost-child–right-sibling data structure, the number of `NULL` pointers is 1 more than the number of nodes.

5.5.6*: At the beginning of Section 5.2 we gave recursive and nonrecursive definitions of trees. Use a structural induction to show that every tree in the recursive sense is a tree in the nonrecursive sense.

[2] The branching factor and the degree are related concepts, but not the same. The branching factor is the maximum degree of any node in the tree.

A Fallacious Form of Tree Induction

It often is tempting to perform inductions on the number of nodes of the tree, where we assume a statement for n-node trees and prove it for $(n+1)$-node trees. This proof will be fallacious if we are not very careful.

When doing inductions on integers in Chapter 2, we suggested the proper methodology, in which we try to prove statement $S(n+1)$ by using $S(n)$; call this approach "leaning back." Sometimes one might be tempted to view this process as starting with $S(n)$ and proving $S(n+1)$; call this approach "pushing out." In the integer case, these are essentially the same idea. However, with trees, we cannot start by assuming the statement for an n-node tree, add a node somewhere, and claim that the result is proved for all $(n+1)$-node trees.

Danger: erroneous argument

For example, consider the claim $S(n)$: "all n-node trees have a path of length $n-1$." It is surely true for the basis, $n=1$. In a false "induction," we might argue: "Assume an n-node tree T has a path of length $n-1$, say to node v. Add a child u to v. We now have an $(n+1)$-node tree with a path of length n, proving the inductive step."

This argument is, of course, fallacious because it does not prove the result for all $(n+1)$-node trees, just some selected trees. A correct proof does not "push out" from n to $n+1$ nodes, because we do not thus reach all possible trees. Rather, we must "lean back" by starting with an arbitrary $(n+1)$-node tree and carefully selecting a node to remove to get an n-node tree.

5.5.7**: Show the converse of Exercise 5.5.6: every tree in the nonrecursive sense is a tree in the recursive sense.

5.6 Binary Trees

Left and right children

This section presents another kind of tree, called a *binary tree*, which is different from the "ordinary" tree introduced in Section 5.2. In a binary tree, a node can have at most two children, and rather than counting children from the left, there are two "slots," one for a *left child* and the other for a *right child*. Either or both slots may be empty.

Fig. 5.27. The two binary trees with two nodes.

✦ **Example 5.19.** Figure 5.27 shows two binary trees. Each has node n_1 as root. The first has n_2 as the left child of the root and no right child. The second has no

left child, and has n_2 as the right child of the root. In both trees, n_2 has neither a left nor a right child. These are the only binary trees with two nodes. ✦

We shall define binary trees recursively, as follows

BASIS. The empty tree is a binary tree.

INDUCTION. If r is a node, and T_1 and T_2 are binary trees, then there is a binary tree with root r, left subtree T_1, and right subtree T_2, as suggested in Fig. 5.28. That is, the root of T_1 is the left child of r, unless T_1 is the empty tree, in which case r has no left child. Similarly, the root of T_2 is the right child of r, unless T_2 is empty, in which case r has no right child.

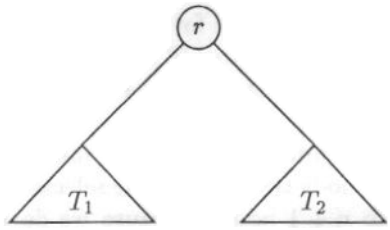


Fig. 5.28. Recursive construction of a binary tree.

Binary Tree Terminology

The notions of paths, ancestors, and descendants introduced in Section 5.2 also apply to binary trees. That is, left and right children are both regarded as "children." A path is still a sequence of nodes $m_1, m_2, \ldots, m_k$ such that m_{i+1} is a (left or right) child of m_i, for $i = 1, 2, \ldots, k-1$. This path is said to be from m_1 to m_k. The case $k = 1$ is permitted, where the path is just a single node.

The two children of a node, if they exist, are siblings. A leaf is a node with neither a left nor a right child; equivalently, a leaf is a node whose left and right subtrees are both empty. An interior node is a node that is not a leaf.

Path length, height, and depth are defined exactly as for ordinary trees. The length of a path in a binary tree is 1 less than the number of nodes; that is, the length is the number of parent-child steps along the path. The height of a node n is the length of the longest path from n to a descendant leaf. The height of a binary tree is the height of its root. The depth of a node n is the length of the path from the root to n.

✦ **Example 5.20.** Figure 5.29 shows the five shapes that a binary tree of three nodes can have. In each binary tree in Fig. 5.29, n_3 is a descendant of n_1, and there is a path from n_1 to n_3. Node n_3 is a leaf in each tree, while n_2 is a leaf in the middle tree and an interior node in the other four trees.

The height of n_3 is 0 in each tree, while the height of n_1 is 2 in all but the middle tree, where the height of n_1 is 1. The height of each tree is the same as the height of n_1 in that tree. Node n_3 is of depth 2 in all but the middle tree, where it is of depth 1. ✦

The Difference Between (Ordinary) Trees and Binary Trees

It is important to understand that while binary trees require us to distinguish whether a child is either a left child or a right child, ordinary trees require no such distinction. That is, binary trees are not just trees all of whose nodes have two or fewer children. Not only are the two trees in Fig. 5.27 different from each other, but they have no relation to the ordinary tree consisting of a root and a single child of the root:

There is another technical difference. While trees are defined to have at least one node, it is convenient to include the *empty tree*, the tree with no nodes, among the binary trees.

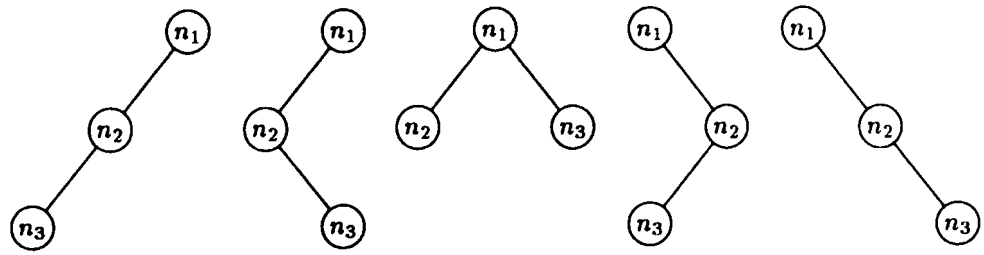


Fig. 5.29. The five binary trees with three nodes.

Data Structures for Binary Trees

There is one natural way to represent binary trees. Nodes are represented by records with two fields, `leftChild` and `rightChild`, pointing to the left and right children of the node, respectively. A `NULL` pointer in either of these fields indicates that the corresponding left or right subtree is missing — that is, that there is no left or right child, respectively.

A binary tree can be represented by a pointer to its root. The empty binary tree is represented naturally by `NULL`. Thus, the following type declarations represent binary trees:

```
typedef struct NODE *TREE;
struct NODE {
    TREE leftChild, rightChild;
};
```

Here, we call the type "pointer to node" by the name `TREE`, since the most common use for this type will be to represent trees and subtrees. We can interpret the `leftChild` and `rightChild` fields either as pointers to the children or as the left and right subtrees themselves.

Optionally, we can add to the structure for **NODE** a label field or fields, and/or we can add a pointer to the parent. Note that the type of the parent pointer is ***NODE**, or equivalently **TREE**.

Recursions on Binary Trees

There are many natural algorithms on binary trees that can be described recursively. The scheme for recursions is more limited than was the scheme of Fig. 5.13 for ordinary trees, since actions can only occur either before the left subtree is explored, between the exploration of the subtrees, or after both have been explored. The scheme for recursions on binary trees is suggested by Fig. 5.30.

```
{
    action A0;
    recursive call on left subtree;
    action A1;
    recursive call on right subtree;
    action A2;
}
```

Fig. 5.30. Template of a recursive algorithm on a binary tree.

✦ **Example 5.21.** Expression trees with binary operators can be represented by binary trees. These binary trees are special, because nodes have either two children or none. (Binary trees in general can have nodes with one child.) For instance, the expression tree of Fig. 5.14, reproduced here as Fig. 5.31, can be thought of as a binary tree.

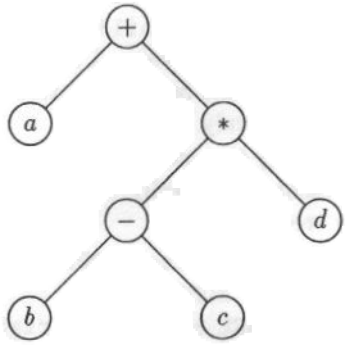


Fig. 5.31. The expression $a + (b - c) * d$ represented by a binary tree.

Suppose we use the type

```
typedef struct NODE *TREE;
struct NODE {
    char nodeLabel;
    TREE leftChild, rightChild
};
```

for nodes and trees. Then Fig. 5.32 shows a recursive function that lists the labels of the nodes of a binary tree T in preorder.

```
    void preorder(TREE t)
    {
(1)     if (t != NULL) {
(2)         printf("%c\n", t->nodeLabel)
(3)         preorder(t->leftChild);
(4)         preorder(t->rightChild);
        }
    }
```

Fig. 5.32. Preorder listing of binary trees.

The behavior of this function is similar to that of the function of the same name in Fig. 5.15 that was designed to work on ordinary trees. The significant difference is that when the function of Fig. 5.32 comes to a leaf, it calls itself on the (missing) left and right children. These calls return immediately, because when t is `NULL`, none of the body of the function except the test of line (1) is executed. We could save the extra calls if we replaced lines (3) and (4) of Fig. 5.32 by

```
(3)  if (t->leftChild != NULL) preorder(t->leftChild);
(4)  if (t->rightChild != NULL) preorder(t->rightChild);
```

However, that would not protect us against a call to `preorder` from another function, with `NULL` as the argument. Thus, we would have to leave the test of line (1) in place for safety. ✦

EXERCISES

5.6.1: Write a function that prints an inorder listing of the (labels of the) nodes of a binary tree. Assume that the nodes are represented by records with left-child and right-child pointers, as described in this section.

5.6.2: Write a function that takes a binary expression tree and prints a fully parenthesized version of the represented expression. Assume the same data structure as in Exercise 5.6.1.

5.6.3*: Repeat Exercise 5.6.2 but print only the needed parentheses, assuming the usual precedence and associativity of arithmetic operators.

5.6.4: Write a function that produces the height of a binary tree.

5.6.5: Define a node of a binary tree to be a *full* if it has both a left and a right child. Prove by structural induction that the number of full nodes in a binary tree is 1 fewer than the number of leaves.

Inorder Traversals

In addition to preorder and postorder listings of binary trees, there is another ordering of nodes that makes sense for binary trees only. An *inorder* listing of the nodes of a binary tree is formed by listing each node after exploring the left subtree, but before exploring the right subtree (i.e., in the position for action A_1 of Fig. 5.30). For example, on the tree of Fig. 5.31, the inorder listing would be $a + b - c * d$.

A preorder traversal of a binary tree that represents an expression produces the prefix form of that expression, and a postorder traversal of the same tree produces the postfix form of the expression. The inorder traversal almost produces the ordinary, or infix, form of an expression, but the parentheses are missing. That is, the tree of Fig. 5.31 represents the expression $a + (b - c) * d$, which is not the same as the inorder listing, $a + b - c * d$, but only because the necessary parentheses are missing from the latter.

To be sure that needed parentheses are present, we could parenthesize all operators. In this modified inorder traversal, action A_0, the step performed before exploring the left subtree, checks whether the label of the node is an operator and, if so, prints `'('`, a left parenthesis. Similarly, action A_2, performed after exploring both subtrees, prints a right parenthesis, `')'`, if the label is an operator. The result, applied to the binary tree of Fig. 5.31, would be $\Big(a + \big((b - c) * d\big)\Big)$, which has the needed pair of parentheses around $b - c$, along with two pairs of parentheses that are redundant.

5.6.6: Suppose we represent a binary tree by the left-child, right-child record type. Prove by structural induction that the number of `NULL` pointers is 1 greater than the number of nodes.

5.6.7**: Trees can be used to represent recursive calls. Each node represents a recursive call of some function F, and its children represent the calls made by F. In this exercise, we shall consider the recursion for $\binom{n}{m}$ given in Section 4.5, based on the recursion $\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$. Each call can be represented by a binary tree. If a node corresponds to the computation of $\binom{n}{m}$, and the basis cases ($m = 0$ and $m = n$) do not apply, then the left child represents $\binom{n-1}{m}$ and the left child represents $\binom{n-1}{m-1}$. If the node represents a basis case, then it has neither left nor right child.

a) Prove by structural induction that a binary tree with root corresponding to $\binom{n}{m}$ has exactly $2\binom{n}{m} - 1$ nodes.

b) Use (a) to show that the running time of the recursive algorithm for $\binom{n}{m}$ is $O\Big(\binom{n}{m}\Big)$. Note that this running time is therefore also $O(2^n)$, but the latter is a smooth-but-not-tight bound.

5.7 Binary Search Trees

A common activity found in a variety of computer programs is the maintenance of

Structural Inductions on Binary Trees

A structural induction can be applied to a binary tree as well as to an ordinary tree. There is, in fact, a somewhat simpler scheme to use, in which the basis is an empty tree. Here is a summary of the technique.

1. Specify the statement $S(T)$ to be proved, where T is a binary tree.
2. Prove the basis, that $S(T)$ is true if T is the empty tree.
3. Set up the inductive step by letting T be a tree with root r and subtrees T_L and T_R. State that you assume the inductive hypothesis: that $S(T_L)$ and $S(T_R)$ are true.
4. Prove that $S(T)$ is true under the assumptions mentioned in (3).

a set of values from which we wish to

1. Insert elements into the set,
2. Delete elements from the set, and
3. Look up an element to see whether it is currently in the set.

One example is a dictionary of English words, where from time to time we insert a new word, such as `fax`, delete a word that has fallen into disuse, such as `aegilops`, or look up a string of letters to see whether it is a word (as part of a spelling-checker program, for instance).

Dictionary

Because this example is so familiar, a set upon which we can execute the operations *insert*, *delete*, and *lookup*, as defined above, is called a *dictionary*, no matter what the set is used for. As another example of a dictionary, a professor might keep a roll of the students in a class. Occasionally, a student will be added to the class (an insert), or will drop the class (a delete), or it will be necessary to tell whether a certain student is registered for the class (a lookup).

One good way to implement a dictionary is with a binary search tree, which is a kind of labeled binary tree. We assume that the labels of nodes are chosen from a set with a "less than" order, which we shall write as $<$. Examples include the reals or integers with the usual less than order; or character strings, with the lexicographic or alphabetic order represented by $<$.

Binary search tree property

A *binary search tree* (*BST*) is a labeled binary tree in which the following property holds at every node x in the tree: all nodes in the left subtree of x have labels less than the label of x, and all nodes in the right subtree have labels greater than the label of x. This property is called the *binary search tree property*.

✦ **Example 5.22.** Figure 5.33 shows a binary search tree for the set

{`Hairy, Bashful, Grumpy, Sleepy, Sleazy, Happy`}

where the $<$ order is lexicographic. Note that the names in the left subtree of the root are all lexicographically less than `Hairy`, while those in the right subtree are all lexicographically greater. This property holds at every node of the tree. ✦

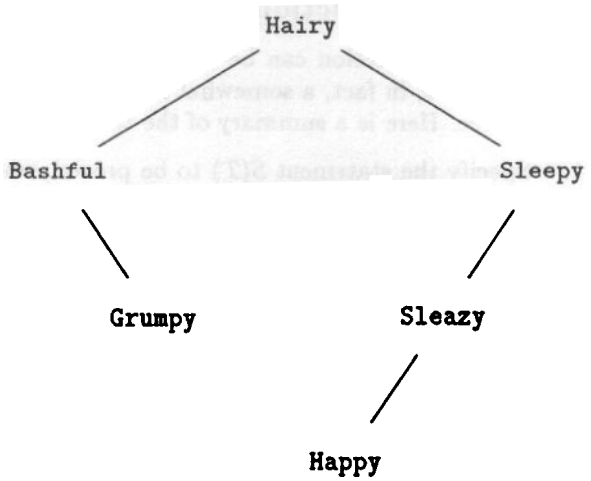


Fig. 5.33. Binary search tree with six nodes labeled by strings.

Implementation of a Dictionary as a Binary Search Tree

We can represent a binary search tree as any labeled binary tree. For example, we might define the type **NODE** by

```
typedef struct NODE *TREE;
struct NODE {
    ETYPE element;
    TREE leftChild, rightChild;
};
```

A binary search tree is represented by a pointer to the root node of the binary search tree. The type of an element, **ETYPE**, should be set appropriately. Throughout the programs of this section, we shall assume **ETYPE** is **int** so that comparisons between elements can be done simply using the arithmetic comparison operators <, == and >. In the examples involving lexicographic comparisons, we assume the comparisons in the programs will be done by the appropriate comparison functions *lt*, *eq*, and *gt* as discussed in Section 2.2.

Looking Up an Element in a Binary Search Tree

Suppose we want to look for an element x that may be in a dictionary represented by a binary search tree T. If we compare x with the element at the root of T, we can take advantage of the BST property to locate x quickly or determine that x is not present. If x is at the root, we are done. Otherwise, if x is less than the element at the root, x could be found only in the left subtree (by the BST property); and if x is greater, then it could be only in the right subtree (again, because of the BST property). That is, we can express the *lookup* operation by the following recursive algorithm.

BASIS. If the tree T is empty, then x is not present. If T is not empty, and x appears at the root, then x is present.

Abstract Data Types

A collection of operations, such as *insert*, *delete*, and *lookup*, that may be performed on a set of objects or a certain kind is sometimes called an *abstract data type* or ADT. The concept is also variously called a *class*, or a *module*. We shall study several abstract data types in Chapter 7, and in this chapter, we shall see one more, the priority queue.

An ADT can have more than one abstract implementation. For example, we shall see in this section that the binary search tree is a good way to implement the dictionary ADT. Lists are another plausible, though usually less efficient, way to implement the dictionary ADT. Section 7.6 covers hashing, another good implementation of the dictionary.

Each abstract implementation can, in turn, be implemented concretely by several different data structures. As an example, we shall use the left-child–right-child implementation of binary trees as a data structure implementing a binary search tree. This data structure, along with the appropriate functions for *insert*, *delete*, and *lookup*, becomes an implementation of the dictionary ADT.

An important reason for using ADT's in programs is that the data underlying the ADT is accessible only through the operations of the ADT, such as *insert*. This restriction is a form of defensive programming, protecting against accidental alteration of data by functions that manipulate the data in unexpected ways. A second important reason for using ADT's is that they allow us to redesign the data structures and functions implementing their operations, perhaps to improve the efficiency of operations, without having to worry about introducing errors into the rest of the program. There can be no new errors if the only interface to the ADT is through correctly rewritten functions for its operations.

INDUCTION. If T is not empty but x is not at the root, let y be the element at the root of T. If $x < y$ look up x only in the left subtree of the root, and if $x > y$ look up x only in the right subtree of y. The BST property guarantees that x cannot be in the subtree we do not search.

✦ **Example 5.23.** Suppose we want to look up `Grumpy` in the binary search tree of Fig. 5.33. We compare `Grumpy` with `Hairy` at the root and find that `Grumpy` precedes `Hairy` in lexicographic order. We thus call `lookup` on the left subtree.

The root of the left subtree is `Bashful`, and we compare this label with `Grumpy`, finding that the former precedes the latter. We thus call `lookup` recursively on the right subtree of `Bashful`. Now we find `Grumpy` at the root of this subtree and return `TRUE`. These steps would be carried out by a function modeled after Fig. 5.34 that dealt with lexicographic comparisons. ✦

More concretely, the recursive function `lookup(x,T)` in Fig. 5.34 implements this algorithm, using the left-child–right-child data structure. Note that `lookup` returns a value of type `BOOLEAN`, which is a defined type synonymous with `int`, but with the intent that only defined values `TRUE` and `FALSE`, defined to be 1 and 0, respectively, will be used. Type `BOOLEAN` was introduced in Section 1.6. Also, note that `lookup` is written only for types that can be compared by =, <, and so on. It would require rewriting for data like the character strings used in Example 5.23.

At line (1), `lookup` determines whether T is empty. If not, then at line (3) `lookup` determines whether x is stored at the current node. If x is not there, then `lookup` recursively searches the left subtree or right subtree depending on whether x is less than or greater than the element stored at the current node.

```
    BOOLEAN lookup(ETYPE x, TREE T)
    {
(1)     if (T == NULL)
(2)         return FALSE;
(3)     else if (x == T->element)
(4)
(5)             (x < T->element)
(6)             rn lookup(x, T->leftChild);
                x must be > T->element */
(7)             rn lookup(x, T->rightChild);
    }
```

Fig. 5.34. Function `lookup(x,T)` returns `TRUE` if x is in T, `FALSE` otherwise.

Inserting an Element into a Binary Search Tree

Adding a new element x to a binary search tree T is straightforward. The following recursive algorithm sketches the idea:

BASIS. If T is an empty tree, replace T by a tree consisting of a single node and place x at that node. If T is not empty and its root has element x, then x is already in the dictionary, and we do nothing.

INDUCTION. If T is not empty and does not have x at its root, then insert x into the left subtree if x is less than the element at the root, or insert x into the right subtree if x is greater than the element at the root.

The function `insert(x,T)` shown in Fig. 5.35 implements this algorithm for the left-child–right-child data structure. When we find that the value of T is `NULL` at line (1), we create a new node, which becomes the tree T. This tree is created by lines (2) through (5) and returned at line (10).

If x is not found at the root of T, then, at lines (6) through (9), `insert` is called on the left or right subtree, whichever is appropriate. The subtree, modified by the insertion, becomes the new value of the left or right subtree of the root of T at lines (7) or (9), respectively. Line (10) returns the augmented tree.

Notice that if x is at the root of T, then none of the tests of lines (1), (6), and (8) succeed. In this case, `insert` returns T without doing anything, which is correct, since x is already in the tree.

✦ **Example 5.24.** Let us continue with Example 5.23, understanding that technically, the comparison of character strings requires slightly different code from that of Fig. 5.35, in which arithmetic comparisons like < are replaced by calls to suitably defined functions like *lt*. Figure 5.36 shows the binary search tree of Fig. 5.33

```
    TREE insert(ETYPE x, TREE T)
    {
(1)     if (T == NULL) {
(2)         T = (TREE) malloc(sizeof(struct NODE));
(3)         T->element = x;
(4)         T->leftChild == NULL;
(5)         T->rightChild == NULL;
        }
(6)     else if (x < T->element)
(7)         T->leftChild = insert(x, T->leftChild);
(8)     else if (x > T->element)
(9)         T->rightChild = insert(x, T->rightChild);
(10)    return T;
    }
```

Fig. 5.35. Function `insert(x,T)` adds x to T.

after we insert **Filthy**. We begin by calling **insert** at the root, and we find that **Filthy** < **Hairy**. Thus, we call **insert** on the left child, at line (7) of Fig. 5.35. The result is that we find **Filthy** > **Bashful**, and so we call **insert** on the right child, at line (9). That takes us to **Grumpy**, which follows **Filthy** in lexicographic order, and we call **insert** on the left child of **Grumpy**.

The pointer to the left child of **Grumpy** is **NULL**, so at line (1) we discover that we must create a new node. This one-node tree is returned to the call of **insert** at the node for **Grumpy**, and the tree is installed as the value of the left child of **Grumpy** at line (7). The modified tree with **Grumpy** and **Filthy** is returned to the call of **insert** at the node labeled **Bashful**, and this modified tree becomes the right child of **Bashful**. Then, continuing up the tree, the new tree rooted at **Bashful** becomes the left child of the root of the entire tree. The final tree is shown in Fig. 5.36. ✦

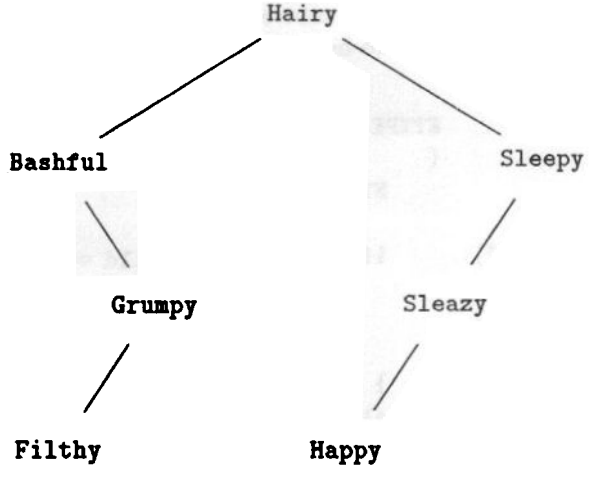


Fig. 5.36. Binary search tree after inserting **Filthy**.

Deleting an Element from a Binary Search Tree

Deleting an element x from a binary search tree is a little more complicated than *lookup* or *insert*. To begin, we may locate the node containing x; if there is no such node, we are done, since x is not in the tree to begin with. If x is at a leaf, we can simply delete the leaf. If x is at an interior node n, however, we cannot delete that node, because to do so would disconnect the tree.

We must rearrange the tree in some way so that the BST property is maintained and yet x is no longer present. There are two cases. First, if n has only one child, we can replace n by that child, and the BST property will be maintained.

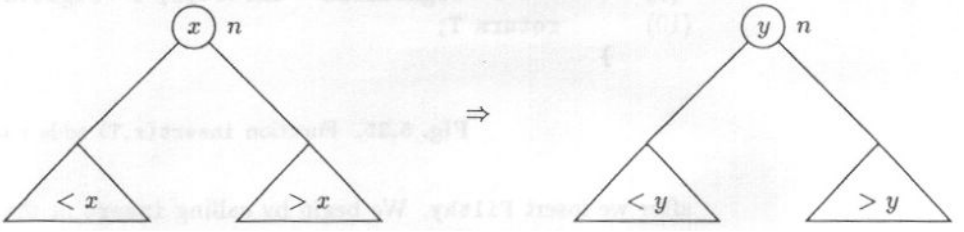


Fig. 5.37. To delete x, remove the node containing y, the smallest element in the right subtree, and then replace the label x by y at node n.

Second, suppose n has both children present. One strategy is to find the node m with label y, the smallest element in the right subtree of n, and replace x by y in node n, as suggested by Fig. 5.37. We can then remove node m from the right subtree.

The BST property continues to hold. The reason is that x is greater than everything in the left subtree of n, and so y, being greater than x (because y is in the right subtree of n), is also greater than everything in the left subtree of n. Thus, as far as the left subtree of n is concerned, y is a suitable element at n. As far as the right subtree of n is concerned, y is also suitable as the root, because y was chosen to be the smallest element in the right subtree.

```
     ETYPE deletemin(TREE *pT)
     {
         ETYPE min;

(1)      if ((*pT)->leftChild == NULL) {
(2)          min = (*pT)->element;
(3)          (*pT) = (*pT)->rightChild;
(4)          return min;
         }
         else
(5)          return deletemin(&((*pT)->leftChild));
     }
```

Fig. 5.38. Function `deletemin(pT)` removes and returns the smallest element from T.

It is convenient to define a function **deletemin(pT)**, shown in Fig. 5.38, to remove the node containing the smallest element from a nonempty binary search tree and to return the value of that smallest element. We pass to the function an argument that is the address of the pointer to the tree **T**. All references to **T** in the function are done indirectly through this pointer.

Call by reference

This style of tree manipulation, where we pass the function an argument that is a pointer to a place where a pointer to a node (i.e., a tree) is found, is called *call by reference*. It is essential in Fig. 5.38 because at line (3), where we have found a pointer to a node m whose left child is **NULL**, we wish to replace this pointer by another pointer — the pointer in the **rightChild** field of m. If the argument of *deletemin* were a pointer to a node, then the change would take place locally to the call to deletemin, and there would not actually be a change to the pointers in the tree itself. Incidentally, we could use the call-by-reference style to implement *insert* as well. In that case, we could modify the tree directly and not have to return a revised tree as we did in Fig. 5.35. We leave such a revised *insert* function as an exercise.

Now, let us see how Fig. 5.38 works. We locate the smallest element by following left children until we find a node whose left child is **NULL** at line (1) of Fig. 5.38. The element y at this node m must be the smallest in the subtree. To see why, first observe that y is smaller than the element at any ancestor of m in the subtree, because we have followed only left children. The only other nodes in the subtree are either in the right subtree of m, in which case their elements are surely larger than y by the BST property, or in the right subtree of one of m's ancestors. But elements in the right subtrees are greater than the element at some ancestor of m, and therefore greater than y, as suggested by Fig. 5.39.

Having found the smallest element in the subtree, we record this value at line (2), and at line (3) we replace the node of the smallest element by its right subtree. Note that when we delete the smallest element from the subtree, we always have the easy case of deletion, because there is no left subtree.

The only remaining point regarding **deletemin** is that when the test of line (1) fails, meaning that we are not yet at the smallest element, we proceed to the left child. That step is accomplished by the recursive call at line (5).

The function **delete(x,pT)** is shown in Fig. 5.40. If **pT** points to an empty tree T, there is nothing to do, and the test of line (1) makes sure that nothing is done. Otherwise, the tests of lines (2) and (4) handle the cases where x is not at the root, and we are directed to the left or right subtree, as appropriate. If we reach line (6), then x must be at the root of T, and we must replace the root node. Line (6) tests for the possibility that the left child is **NULL**, in which case we simply replace T by its right subtree at line (7). Similarly, if at line (8) we find that the right child is **NULL** then at line (9) we replace T by its left subtree. Note that if both children of the root are **NULL**, then we replace T by **NULL** at line (7).

The remaining case, where neither child is **NULL**, is handled at line (10). We call **deletemin**, which returns the smallest element, y, of the right subtree and also deletes y from that subtree. The assignment of line (10) replaces x by y at the root of T.

✦ **Example 5.25.** Figure 5.41 shows what would happen if we used a function similar to **delete** (but able to compare character strings) to remove **Hairy** from the

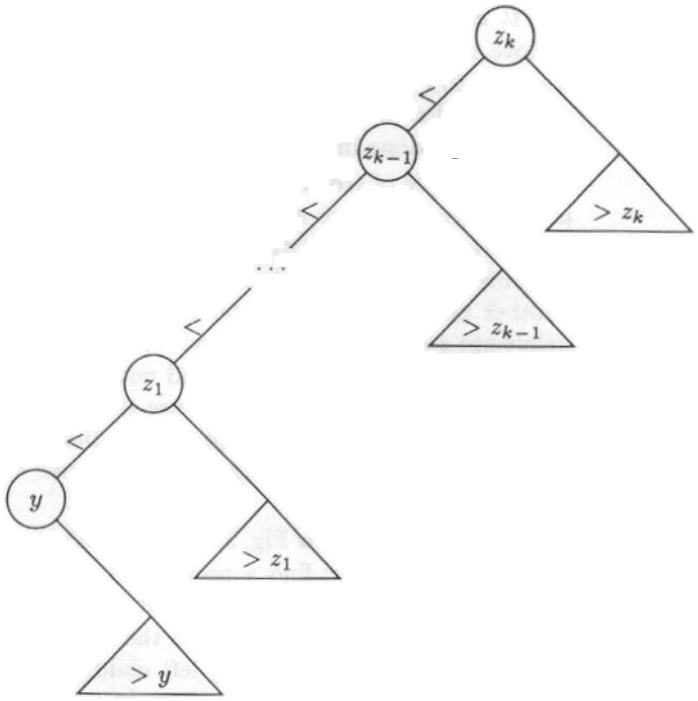


Fig. 5.39. All the other elements in the right subtree are greater than y.

```
    void delete(ETYPE x, TREE *pT)
    {
(1)     if ((*pT) != NULL)
(2)         if (x < (*pT)->element)
(3)             delete(x, &((*pT)->leftChild));
(4)         else if (x > (*pT)->element)
(5)             delete(x, &((*pT)->rightChild));
            else /* here, x is at the root of (*pT) */
(6)             if ((*pT)->leftChild == NULL)
(7)                 (*pT) = (*pT)->rightChild;
(8)             else if ((*pT)->rightChild == NULL)
(9)                 (*pT) = (*pT)->leftChild;
                else /* here, neither child is NULL */
(10)                (*pT)->element =
                            deletemin(&((*pT)->rightChild));
    }
```

Fig. 5.40. Function `delete(x,pT)` removes the element x from T.

binary search tree of Fig. 5.36. Since **Hairy** is at a node with two children, **delete** calls the function **deletemin**, which removes and returns the smallest element, **Happy**, from the right subtree of the root. **Happy** then becomes the label of the root of the tree, the node at which **Hairy** was stored. ✦

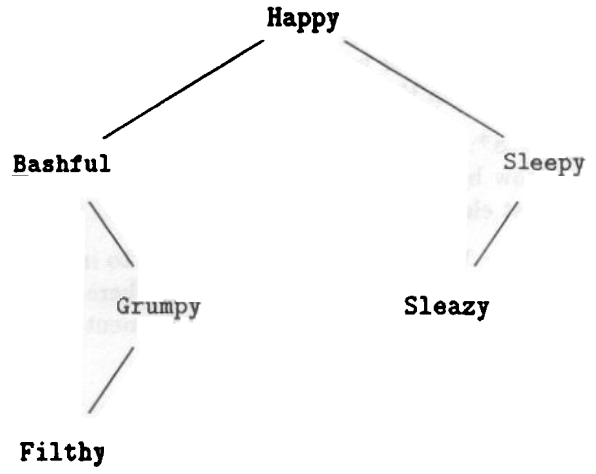


Fig. 5.41. Binary search tree after deleting Hairy.

EXERCISES

5.7.1: Suppose that we use a leftmost-child–right-sibling implementation for binary search trees. Rewrite the functions that implement the dictionary operations *insert*, *delete*, and *lookup* to work for this data structure.

5.7.2: Show what happens to the binary search tree of Fig. 5.33 if we insert the following dwarfs in order: `Doc`, `Dopey`, `Inky`, `Blinky`, `Pinky`, and `Sue`. Then show what happens when we delete in order: `Doc`, `Sleazy`, and `Hairy`.

5.7.3: Rewrite the functions `lookup`, `insert`, and `delete` to use lexicographic comparisons on strings instead of arithmetic comparisons on integers.

5.7.4*: Rewrite the function `insert` so that the tree argument is passed by reference.

5.7.5*: We wrote `delete` in the "call by reference" style. However, it is also possible to write it in a style like that of our `insert` function, where it takes a tree as argument (rather than a pointer to a tree) and returns the tree missing the deleted element. Write this version of the dictionary operation *delete*. *Note*: It is not really possible to have *deletemin* return a revised tree, since it must also return the minimum element. We could rewrite *deletemin* to return a structure with both the new tree and the minimum element, but that approach is not recommended.

5.7.6: Instead of handling the deletion of a node with two children by finding the least element in the right subtree, we could also find the greatest element in the left subtree and use that to replace the deleted element. Rewrite the functions `delete` and `deletemin` from Figs. 5.38 and 5.40 to incorporate this modification.

5.7.7*: Another way to handle *delete* when we need to remove the element at a node n that has parent p, (nonempty) left child l, and (nonempty) right child r is to find the node m holding the least element in the right subtree of n. Then, make r a left or right child of p, whichever n was, and make l the left child of m (note that

m cannot previously have had a left child). Show why this set of changes preserves the BST property. Would you prefer this strategy to the one described in Section 5.7? *Hint*: For both methods, consider their effect on the lengths of paths. As we shall see in the next section, short paths make the operations run fast.

5.7.8*: In this exercise, refer to the binary search tree represented in Fig. 5.39. Show by induction on i that if $1 \leq i \leq k$, then $y < z_i$. Then, show that y is the least element in the tree rooted at z_k.

5.7.9: Write a complete C program to implement a dictionary that stores integers. Accept commands of the form `x i`, where `x` is one of the letters `i` (insert), `d` (delete), and `l` (lookup). Integer i is the argument of the command, the integer to be inserted, deleted, or searched for.

5.8 Efficiency of Binary Search Tree Operations

The binary search tree provides a reasonably fast implementation of a dictionary. First, notice that each of the operations *insert*, *delete*, and *lookup* makes a number of recursive calls equal to the length of the path followed (but this path must include the route to the smallest element of the right subtree, in case `deletemin` is called). Also, a simple analysis of the functions `lookup`, `insert`, `delete`, and `deletemin` tells us that each operation takes $O(1)$ time, plus the time for one recursive call. Moreover, since this recursive call is always made at a child of the current node, the height of the node in each successive call decreases by at least 1.

Thus, if $T(h)$ is the time taken by any of these functions when called with a pointer to a node of height h, we have the following recurrence relation upper-bounding $T(h)$:

BASIS. $T(0) = O(1)$. That is, when called on a leaf, the call either terminates without further calls or makes a recursive call with a `NULL` argument and then returns without further calls. All of this work takes $O(1)$ time.

INDUCTION. $T(h) \leq T(h-1) + O(1)$ for $h \geq 1$. That is, the time taken by a call on any interior node is $O(1)$ plus the time for a recursive call, which is on a node of height at most $h-1$. If we make the reasonable assumption that $T(h)$ increases with increasing h, then the time for the recursive call is no greater than $T(h-1)$.

The solution to the recurrence for $T(h)$ is $O(h)$, as discussed in Section 3.9. Thus, the running time of each dictionary operation on a binary search tree of n nodes is at most proportional to the height of the tree. But what is the height of a typical binary search tree of n nodes?

The Worst Case

In the worst case, all the nodes in the binary tree will be arranged in a single path, like the tree of Fig. 5.42. That tree would result, for example, from taking a list of k elements in sorted order and inserting them one at a time into an initially empty tree. There are also trees in which the single path does not consist of right children

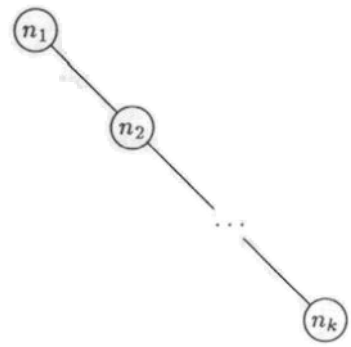


Fig. 5.42. A degenerate binary tree.

only but is a mixture of right and left children, with the path taking a turn either left or right at each interior node.

The height of a k-node tree like Fig. 5.42 is clearly $k - 1$. We thus expect that *lookup*, *insert*, and *delete* will take $O(k)$ time on a dictionary of k elements, if the representation of that dictionary happens to be one of these unfortunate trees. Intuitively, if we need to look for element x, on the average we shall find it halfway down the path, requiring us to look at $k/2$ nodes. If we fail to find x, we shall likewise have to search down the tree until we come to the place where x would be found, which will also be halfway down, on the average. Since each of the operations *lookup*, *insert*, and *delete* requires searching for the element involved, we know that these operations each take $O(k)$ time on the average, given one of the bad trees of the form of Fig. 5.42.

The Best Case

However, a binary tree need not grow long and thin like Fig. 5.42; it could be short and bushy like Fig. 5.43. A tree like the latter, where every interior node down to some level has both children present and the next level has all the leaves, is called a *full* or *complete* tree.

Complete tree

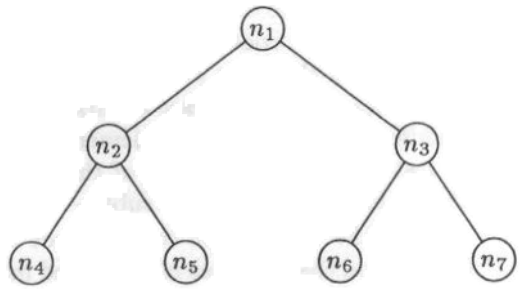


Fig. 5.43. Full binary tree with seven nodes.

A complete binary tree of height h has $2^{h+1} - 1$ nodes. We can prove this claim by induction on the height h.

BASIS. If $h = 0$, the tree consists of a single node. Since $2^{0+1} - 1 = 1$, the basis case holds.

INDUCTION. Suppose that a complete binary tree of height h has $2^{h+1} - 1$ nodes, and consider a complete binary tree of height $h + 1$. This tree consists of one node at the root and left and right subtrees that are complete binary trees of height h. For example, the height-2 complete binary tree of Fig. 5.43 consists of the root, n_1; a left subtree containing n_2, n_4, and n_5, which is a complete binary tree of height 1; and a right subtree consisting of the remaining three nodes, which is another complete binary tree of height 1. Now the number of nodes in two complete binary trees of height h is $2(2^h - 1)$, by the inductive hypothesis. When we add the root node, we find that a complete binary tree of height $h+1$ has $2(2^h - 1)+1 = 2^{h+1} - 1$ nodes, which proves the inductive step.

Now we can invert this relationship and say that a complete binary tree of $k = 2^{h+1} - 1$ nodes has height h. Equivalently, $k+1 = 2^{h+1}$. If we take logarithms, then $\log_2(k+1) = h+1$, or approximately, h is $O(\log k)$. Since the running time of *lookup*, *insert*, and *delete* is proportional to the height of the tree, we can conclude that on a complete binary tree, these operations take time that is logarithmic in the number of nodes. That performance is much better than the linear time taken for pessimal trees like Fig. 5.42. As the dictionary size becomes large, the running time of the dictionary operations grows much more slowly than the number of elements in the set.

The Typical Case

Is Fig. 5.42 or Fig. 5.43 the more common case? Actually, neither is common in practice, but the complete tree of Fig. 5.43 offers efficiency of the dictionary operations that is closer to what the typical case provides. That is, *lookup*, *insert*, and *delete* take logarithmic time, on the average.

A proof that the typical binary tree offers logarithmic time for the dictionary operations is difficult. The essential point of the proof is that the expected value of the length of a path from the root of such a tree to a random node is $O(\log n)$. A recurrence equation for this expected value is given in the exercises.

However, we can see intuitively why that should be the correct running time, as follows. The root of a binary tree divides the nodes, other than itself, into two subtrees. In the most even division, a k-node tree will have two subtrees of about $k/2$ nodes each. That case occurs if the root element happens to be exactly in the middle of the sorted list of elements. In the worst division, the root element is first or last among the elements of the dictionary and the division is into one subtree that is empty and another subtree of $k - 1$ nodes.

On the average, we could expect the root element to be halfway between the middle and the extremes in the sorted list; and we might expect that, on the average, about $k/4$ nodes go into the smaller subtree and $3k/4$ nodes into the larger. Let us assume that as we move down the tree we always move to the root of the larger subtree at each recursive call and that similar assumptions apply to the distribution of elements at each level. At the first level, the larger subtree will be divided in the 1:3 ratio, leaving a largest subtree of $(3/4)(3k/4)$, or $9k/16$, nodes at the second level. Thus, at the dth-level, we would expect the largest subtree to have about $(3/4)^d k$ nodes.

When d becomes sufficiently large, the quantity $(3/4)^d k$ will be close to 1, and we can expect that, at this level, the largest subtree will consist of a single leaf.

Thus, we ask, For what value of d is $(3/4)^d k \leq 1$? If we take logarithms to the base 2, we get

$$d \log_2(3/4) + \log_2 k \leq \log_2 1 \tag{5.1}$$

Now $\log_2 1 = 0$, and the quantity $\log_2(3/4)$ is a negative constant, about -0.4. Thus we can rewrite (5.1) as $\log_2 k \leq 0.4d$, or $d \geq (\log_2 k)/0.4 = 2.5 \log_2 k$.

Put another way, at a depth of about two and a half times the logarithm to the base 2 of the number of nodes, we expect to find only leaves (or to have found the leaves at higher levels). This argument justifies, but does not prove, the statement that the typical binary search tree will have a height that is proportional to the logarithm of the number of nodes in the tree.

EXERCISES

5.8.1: If tree T has height h and branching factor b, what are the largest and smallest numbers of nodes that T can have?

5.8.2**: Perform an experiment in which we choose one of the $n!$ orders for n different values and insert the values in this order into an initially empty binary search tree. Let $P(n)$ be the expected value of the depth of the node at which a particular value v among the n values is found after this experiment.

a) Show that, for $n \geq 2$,

$$P(n) = 1 + \frac{2}{n^2} \sum_{k=1}^{n-1} kP(k)$$

b) Prove that $P(n)$ is $O(\log n)$.

5.9 Priority Queues and Partially Ordered Trees

So far, we have seen only one abstract data type, the dictionary, and one implementation for it, the binary search tree. In this section we shall study another abstract data type and one of its most efficient implementations. This ADT, called a *priority queue*, is a set of elements each of which has an associated *priority*. For example, the elements could be records and the priority could be the value of one field of the record. The two operations associated with the priority queue ADT are the following:

1. Inserting an element into the set (*insert*).
2. Finding and deleting from the set an element of highest priority (this combined operation is called *deletemax*). The deleted element is returned by this function.

✦ **Example 5.26.** A time-shared operating system accepts requests for service from various sources, and these jobs may not all have the same priority. For example, at highest priority may be the system processes; these would include the "daemons" that watch for incoming data, such as the signal generated by a keystroke at a terminal or the arrival of a packet of bits over a local area network. Then may come user processes, the commands issued by ordinary users. Below these we may

have certain background jobs such as backup of data to tape or long calculations that the user has designated to run with a low priority.

Jobs can be represented by records consisting of an integer ID for the job and an integer for the job's priority. That is, we might use the structure

```
struct ETYPE {
    int jobID;
    int priority;
};
```

for elements of a priority queue. When a new job is initiated, it gets an ID and a priority. We then execute the *insert* operation for this element on the priority queue of jobs waiting for service. When a processor becomes available, the system goes to the priority queue and executes the *deletemax* operation. The element returned by this operation is a waiting job of highest priority, and that is the one executed next. ✦

✦ **Example 5.27.** We can implement a sorting algorithm using the priority queue ADT. Suppose we are given the sequence of integers $a_1, a_2, \ldots, a_n$ to sort. We insert each into a priority queue, using the element's value as its priority. If we then execute *deletemax* n times, the integers will be selected highest first, or in the reverse of their sorted (lowest-first) order. We shall discuss this algorithm in more detail in the next section; it is known as heapsort. ✦

Partially Ordered Trees

An efficient way to implement a priority queue is by a *partially ordered tree* (*POT*), which is a labeled binary tree with the following properties:

1. The labels of the nodes are elements with a "priority"; that priority may be the value of an element or the value of some component of an element.

2. The element stored at a node has at least as large a priority as the elements stored at the children of that node.

POT property

Property 2 implies that the element at the root of any subtree is always a largest element of that subtree. We call property 2 the *partially ordered tree property*, or *POT property*.

✦ **Example 5.28.** Figure 5.44 shows a partially ordered tree with 10 elements. Here, as elsewhere in this section, we shall represent elements by their priorities, as if the element and the priority were the same thing. Note that equal elements can appear on different levels in the tree. To see that the POT property is satisfied at the root, note that 18, the element there, is no less than the elements 18 and 16 found at its children. Similarly, we can check that the POT property holds at every interior node. Thus, Fig. 5.44 is a partially ordered tree. ✦

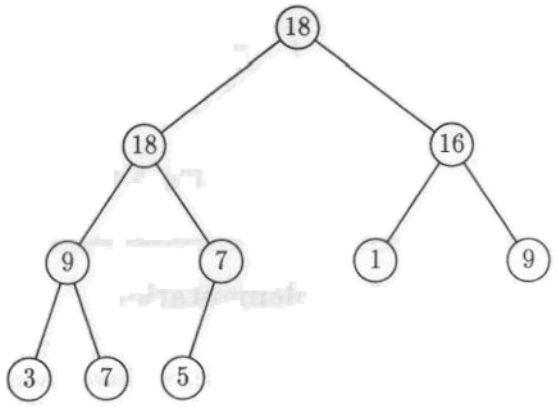


Fig. 5.44. Partially ordered tree with 10 nodes.

Partially ordered trees provide a useful abstract implementation for priority queues. Briefly, to execute *deletemax* we find the node at the root, which must be the maximum, and replace it by the rightmost node on the bottom level. However, when we do so, the POT property may be violated, and so we must restore that property by "bubbling down" the element newly placed at the root until it finds a suitable level where it is smaller than its parent but at least as large as any of its children. To execute *insert*, we can add a new leaf at the bottom level, as far left as possible, or at the left end of a new level if the bottom level is full. Again there may be a violation of the POT property, and if so, we "bubble up" the new element until it finds its rightful place.

Balanced POTs and Heaps

We say that a partially ordered tree is *balanced* if all possible nodes exist at all levels except the bottommost, and the leaves at the bottommost level are as far to the left as possible. This condition implies that if the tree has n nodes, then no path to a node from the root is longer than $\log_2 n$. The tree in Fig. 5.44 is a balanced POT.

Balanced POTs can be implemented using an array data structure called a *heap*, which provides a fast, compact implementation of the priority queue ADT. A heap is simply an array A with a special interpretation for the element indices. We start with the root in $A[1]$; $A[0]$ is not used. Following the root, the levels appear in order. Within a level, the nodes are ordered from left to right.

Thus, the left child of the root is in $A[2]$, and the right child of the root is in $A[3]$. In general, the left child of the node in $A[i]$ is in $A[2i]$ and the right child is in $A[2i+1]$, if these children exist in the partially ordered tree. The balanced nature of the tree allows this representation. The POT property of the elements implies that if $A[i]$ has two children, then $A[i]$ is at least as large as $A[2i]$ and $A[2i+1]$, and if $A[i]$ has one child, then $A[i]$ is at least as large as $A[2i]$.

✦ **Example 5.29.** The heap for the balanced partially ordered tree in Fig. 5.44 is shown in Fig. 5.45. For instance, $A[4]$ holds the value 9; this array element represents the left child of the left child of the root in Fig. 5.44. The children of this node are found in $A[8]$ and $A[9]$. Their elements, 3 and 7, are each no greater

1	2	3	4	5	6	7	8	9	10
18	18	16	9	7	1	9	3	7	5

Fig. 5.45. Heap for Fig. 5.44.

Layers of Implementation

It is useful to compare our two ADT's, the dictionary and the priority queue, and to notice that, in each case, we have given one abstract implementation and one data structure for that implementation. There are other abstract implementations for each, and other data structures for each abstract implementation. We promised to discuss other abstract implementations for the dictionary, such as the hash table, and in the exercises of Section 5.9 we suggest that the binary search tree may be a suitable abstract implementation for the priority queue. The table below summarizes what we already know about abstract implementations and data structures for the dictionary and the priority queue.

ADT	ABSTRACT IMPLEMENTATION	DATA STRUCTURE
dictionary	binary search tree	left-child–right-child structure
priority queue	balanced partially ordered tree	heap

than 9, as is required by the POT property. Array element $A[5]$, which corresponds to the right child of the left child of the root, has a left child in $A[10]$. It would have a right child in $A[11]$, but the partially ordered tree has only 10 elements at the moment, and so $A[11]$ is not part of the heap. ✦

While we have shown tree nodes and array elements as if they were the priorities themselves, in principle an entire record appears at the node or in the array. As we shall see, we shall have to do much swapping of elements between children and parents in a partially ordered tree or its heap representation. Thus, it is considerably more efficient if the array elements themselves are pointers to the records representing the objects in the priority queue and these records are stored in another array "outside" the heap. Then we can simply swap pointers, leaving the records in place.

Performing Priority Queue Operations on a Heap

Throughout this section and the next, we shall represent a heap by a global array `A[1..MAX]` of integers. We assume that elements are integers and are equal to their priorities. When elements are records, we can store pointers to the records in the array and determine the priority of an element from a field in its record.

Suppose that we have a heap of $n-1$ elements satisfying the POT property, and we add an nth element in $A[n]$. The POT property continues to hold everywhere,

except perhaps between $A[n]$ and its parent. Thus, if $A[n]$ is larger than $A[n/2]$, the element at the parent, we must swap these elements. Now there may be a violation of the POT property between $A[n/2]$ and its parent. If so, we recursively "bubble up" the new element until it either reaches a position where the parent has a larger element or reaches the root.

The C function **bubbleUp** to perform this operation is shown in Fig. 5.46. It makes use of a function **swap(A,i,j)** that exchanges the elements in $A[i]$ and $A[j]$; this function is also defined in Fig. 5.46. The operation of **bubbleUp** is simple. Given argument i indicating the node that, with its parent, possibly violates the POT property, we test whether $i = 1$ (that is, whether we are already at the root, so that no POT violation can occur), and if not, whether the element $A[i]$ is greater than the element at its parent. If so, we swap $A[i]$ with its parent and recursively call **bubbleUp** at the parent.

```
void swap(int A[], int i, int j)
{
    int temp;

    temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}

void bubbleUp(int A[], int i)
{
    if (i > 1 && A[i] > A[i/2]) {
        swap(A, i, i/2);
        bubbleUp(A, i/2);
    }
}
```

Fig. 5.46. The function **swap** exchanges array elements, and the function **bubbleUp** pushes a new element of a heap into its rightful place.

✦ **Example 5.30.** Suppose we start with the heap of Fig. 5.45 and we add an eleventh element, with priority 13. This element goes in $A[11]$, giving us the array

1	2	3	4	5	6	7	8	9	10	11
18	18	16	9	7	1	9	3	7	5	13

We now call **bubbleUp(A,11)**, which compares $A[11]$ with $A[5]$ and finds that we must swap these elements because $A[11]$ is larger. That is, $A[5]$ and $A[11]$ violate the POT property. Thus, the array becomes

1	2	3	4	5	6	7	8	9	10	11
18	18	16	9	13	1	9	3	7	5	7

Now we call **bubbleUp(A,5)**. This results in comparison of $A[2]$ and $A[5]$. Since $A[2]$ is larger, there is no POT violation, and **bubbleUp(A,5)** does nothing. We have now restored the POT property to the array. ✦

Implementation of insert

We now show how to implement the priority queue operation *insert*. Let n be the current number of elements in the priority queue, and assume **A[1..n]** already satisfies the POT property. We increment n and then store the element to be inserted into the new **A[n]**. Finally, we call **bubbleUp(A,n)**. The code for *insert* is shown in Fig. 5.47. The argument x is the element to be inserted, and the argument *pn* is a pointer to the current size of the priority queue. Note that n must be passed by reference — that is, by a pointer to n — so that when n is incremented the change has an affect that is not local only to *insert*. A check that $n < MAX$ is omitted.

```
void insert(int A[], int x, int *pn)
{
    (*pn)++;
    A[*pn] = x;
    bubbleUp(A, *pn);
}
```

Fig. 5.47. Priority queue operation *insert* implemented on a heap.

To implement the priority queue operation *deletemax*, we need another operation on heaps or partially ordered trees, this time to bubble down an element at the root that may violate the POT property. Suppose that $A[i]$ is a potential violator of the POT property, in that it may be smaller than one or both of its children, $A[2i]$ and $A[2i+1]$. We can swap $A[i]$ with one of its children, but we must be careful which one. If we swap with the larger of the children, then we are sure not to introduce a POT violation between the two former children of $A[i]$, one of which has now become the parent of the other.

Bubbling down

The function **bubbleDown** of Fig. 5.48 implements this operation. After selecting a child with which to swap $A[i]$, it calls itself recursively to eliminate a possible POT violation between the element $A[i]$ in its new position — which is now **A[2i]** or **A[2i+1]** — and one of its new children. The argument n is the number of elements in the heap, or, equivalently, the index of the last element.

This function is a bit tricky. We have to decide which child of $A[i]$ to swap with, if any, and the first thing we do is assume that the larger child is $A[2i]$, at line (1) of Fig. 5.48. If the right child exists (i.e., $child < n$) and the right child is the larger, then the tests of line (2) are met and at line (3) we make *child* be the right child of $A[i]$.

Now at line (4) we test for two things. First, it is possible that $A[i]$ really has no children in the heap. We therefore check whether $A[i]$ is an interior node by asking whether $child \leq n$. The second test of line (4) is whether $A[i]$ is less than $A[child]$. If both these conditions are met, then at line (5) we swap $A[i]$ with its larger child, and at line (6) we recursively call **bubbleDown**, to push the offending element further down, if necessary.

Implementation of deletemax

We can use **bubbleDown** to implement the priority queue operation *deletemax* as shown in Fig. 5.49. The function **deletemax** takes as arguments an array A and

```
    void bubbleDown(int A[], int i, int n)
    {
        int child;

(1)     child = 2*i;
(2)     if (child < n && A[child+1] > A[child])
(3)         ++child;
(4)     if (child <= n && A[i] < A[child]) {
(5)         swap(A, i, child);
(6)         bubbleDown(A, child, n);
        }
    }
```

Fig. 5.48. `bubbleDown` pushes a POT violator down to its proper position.

a pointer *pn* to the number n that is the number of elements currently in the heap. We omit a test that $n > 0$.

In line (1), we swap the element at the root, which is to be deleted, with the last element, in `A[n]`. Technically, we should return the deleted element, but, as we shall see, it is convenient to put it in `A[n]`, which will no longer be part of the heap.

At line (2), we decrement n by 1, effectively deleting the largest element, now residing in the old `A[n]`. Since the root may now violate the POT property, we call `bubbleDown(A,1,n)` at line (3), which will recursively push the offending element down until it either reaches a point where it is no less than either of its children, or becomes a leaf; either way, there is no violation of the POT property.

```
    void deletemax(int A[], int *pn)
    {
(1)     swap(A, 1, *pn);
(2)     --(*pn);
(3)     bubbleDown(A, 1, *pn);
    }
```

Fig. 5.49. Priority queue operation *deletemax* implemented by a heap.

✦ **Example 5.31.** Suppose we start with the heap of Fig. 5.45 and execute *deletemax*. After swapping $A[1]$ and $A[10]$, we set n to 9. The heap then becomes

1	2	3	4	5	6	7	8	9
5	18	16	9	7	1	9	3	7

When we execute `bubbleDown(A,1,9)`, we set *child* to 2. Since $A[2] \geq A[3]$, we do not increment *child* at line (3) of Fig. 5.48. Then since *child* $\leq n$ and $A[1] < A[2]$, we swap these elements, to obtain the array

1	2	3	4	5	6	7	8	9
18	5	16	9	7	1	9	3	7

We then call **bubbleDown(A,2,9)**. That requires us to compare $A[4]$ with $A[5]$ at line (2), and we find that the former is larger. Thus, *child* = 4 at line (4) of Fig. 5.48. When we find that $A[2] < A[4]$, we swap these elements and call **bubbleDown(A,4,9)** on the array

1	2	3	4	5	6	7	8	9
18	9	16	5	7	1	9	3	7

Next, we compare $A[8]$ and $A[9]$, finding that the latter is larger, so that *child* = 9 at line (4) of **bubbleDown(A,4,9)**. We again perform the swap, since $A[4] < A[9]$, resulting in the array

1	2	3	4	5	6	7	8	9
18	9	16	7	7	1	9	3	5

Next, we call **bubbleDown(A,9,9)**. We set *child* to 18 at line (1), and the first test of line (2) fails, because *child* $< n$ is false. Similarly, the test of line (4) fails, and we make no swap or recursive call. The array is now a heap with the POT property restored. ✦

Running Time of Priority Queue Operations

The heap implementation of priority queues offers $O(\log n)$ running time per *insert* or *deletemax* operation. To see why, let us first consider the *insert* program of Fig. 5.47. This program evidently takes $O(1)$ time for the first two steps, plus whatever the call to **bubbleUp** takes. Thus, we need to determine the running time of **bubbleUp**.

Informally, we notice that each time **bubbleUp** calls itself recursively, we are at a node one position closer to the root. Since a balanced partially ordered tree has height approximately $\log_2 n$, the number of recursive calls is $O(\log_2 n)$. Since each call to **bubbleUp** takes time $O(1)$ plus the time of the recursive call, if any, the total time should be $O(\log n)$.

More formally, let $T(i)$ be the running time of **bubbleUp(A,i)**. Then we can create a recurrence relation for $T(i)$ as follows.

BASIS. If $i = 1$, then $T(i)$ is $O(1)$, since it is easy to check that the **bubbleUp** program of Fig. 5.46 does not make any recursive calls and only the test of the if-statement is executed.

INDUCTION. If $i > 1$, then the if-statement test may fail anyway, because $A[i]$ does not need to rise further. If the test succeeds, then we execute *swap*, which takes $O(1)$ time, plus a recursive call to **bubbleUp** with an argument $i/2$ (or slightly less if i is odd). Thus $T(i) \leq T(i/2) + O(1)$.

We thus have, for some constants a and b, the recurrence

$T(1) = a$
$T(i) = T(i/2) + b$ for $i > 1$

as an upper bound on the running time of **bubbleUp**. If we expand $T(i/2)$ we get

$$T(i) = T(i/2^j) + bj \tag{5.2}$$

for each j. As in Section 3.10, we choose the value of j that makes $T(i/2^j)$ simplest. In this case, we make j equal to $\log_2 i$, so that $i/2^j = 1$. Thus, (5.2) becomes $T(i) = a + b \log_2 i$; that is, $T(i)$ is $O(\log i)$. Since **bubbleUp** is $O(\log i)$, so is *insert*.

Now consider *deletemax*. We can see from Fig. 5.49 that the running time of *deletemax* is $O(1)$ plus the running time of **bubbleDown**. The analysis of **bubbleDown**, in Fig. 5.48, is essentially the same as that of **bubbleUp**. We omit it and conclude that **bubbleDown** and *deletemax* also take $O(\log n)$ time.

EXERCISES

5.9.1: Starting with the heap of Fig. 5.45, show what happens when we

a) Insert 3
b) Insert 20
c) Delete the maximum element
d) Again delete the maximum element

5.9.2: Prove Equation (5.2) by induction on i.

5.9.3: Prove by induction on the depth of the POT-property violation that the function **bubbleUp** of Fig. 5.46 correctly restores a tree with one violation to a tree that has the POT property.

5.9.4: Prove that the function **insert(A,x,n)** makes **A** into a heap of size n, if **A** was previously a heap of size $n - 1$. You may use Exercise 5.9.3. What happens if **A** was not previously a heap?

5.9.5: Prove by induction on the height of the POT-property violation that the function **bubbleDown** of Fig. 5.48 correctly restores a tree with one violation to a tree that has the POT property.

5.9.6: Prove that **deletemax(A,n)** makes a heap of size n into one of size $n - 1$. What happens if **A** was not previously a heap?

5.9.7: Prove that **bubbleDown(A,1,n)** takes $O(\log n)$ time on a heap of length n.

5.9.8**: What is the probability that an n-element heap, with distinct element priorities chosen at random, is a partially ordered tree? If you cannot derive the general rule, write a recursive function to compute the probability as a function of n.

5.9.9: We do not need to use a heap to implement a partially ordered tree. Suppose we use the conventional left-child–right-child data structure for binary trees. Show how to implement the functions **bubbleDown**, **insert**, and **deletemax** using this structure instead of the heap structure.

5.9.10*: A binary search tree can be used as an abstract implementation of a priority queue. Show how the operations *insert* and *deletemax* can be implemented using a binary search tree with the left-child–right-child data structure. What is the running time of these operations (a) in the worst case and (b) on the average?

5.10 Heapsort: Sorting with Balanced POTs

We shall now describe the algorithm known as *heapsort*. It sorts an array `A[1..n]` in two phases. In the first phase, heapsort gives A the POT property. The second phase of heapsort repeatedly selects the largest remaining element from the heap until the heap consists of only the smallest element, whereupon the array A is sorted.

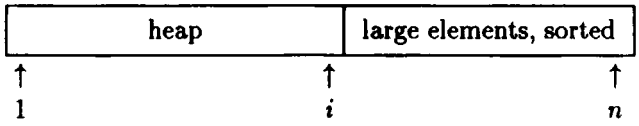


Fig. 5.50. Condition of array A during heapsort.

Figure 5.50 shows the array A during the second phase. The initial part of the array has the POT property, and the remaining part has its elements sorted in nondecreasing order. Furthermore, the elements in the sorted part are the largest $n - i$ elements in the array. During the second phase, i is allowed to run from n down to 1, so that the heap, initially the entire array A, eventually shrinks until it is only the smallest element, located in `A[1]`. In more detail, the second phase consists of the following steps.

1. $A[1]$, the largest element in `A[1..i]`, is exchanged with $A[i]$. Since all elements in `A[i+1..n]` are as large as or larger than any of `A[1..i]`, and since we just moved the largest of the latter group of elements to position i, we know that `A[i..n]` are the largest $n - i + 1$ elements and are in sorted order.
2. The value i is decremented, reducing the size of the heap by 1.
3. The POT property is restored to the initial part of the array by bubbling down the element at the root, which we just moved to $A[1]$.

✦ **Example 5.32.** Consider the array in Fig. 5.45, which has the POT property. Let us go through the first iteration of the second phase. In the first step, we exchange $A[1]$ and $A[10]$ to get:

1	2	3	4	5	6	7	8	9	10
5	18	16	9	7	1	9	3	7	18

The second step reduces the heap size to 9, and the third step restores the POT property to the first nine elements by calling `bubbleDown(1)`. In this call, $A[1]$ and $A[2]$ are exchanged:

1	2	3	4	5	6	7	8	9	10
18	5	16	9	7	1	9	3	7	18

Then, $A[2]$ and $A[4]$ are exchanged:

1	2	3	4	5	6	7	8	9	10
18	9	16	5	7	1	9	3	7	18

Finally, $A[4]$ and $A[9]$ are exchanged:

1	2	3	4	5	6	7	8	9	10
18	9	16	7	7	1	9	3	5	18

At this point, `A[1..9]` has the POT property.

The second iteration of phase 2 begins by swapping the element 18 in `A[1]` with the element 5 in `A[9]`. After bubbling 5 down, the array becomes

1	2	3	4	5	6	7	8	9	10
16	9	9	7	7	1	5	3	18	18

At this stage, the last two elements of the array are the two largest elements, in sorted order.

Phase 2 continues until the array is completely sorted:

1	2	3	4	5	6	7	8	9	10
1	3	5	7	7	9	9	16	18	18

✦

Heapifying an Array

We could describe heapsort informally as follows:

```
for (i = 1; i <= n; i++)
    insert(a_i);
for (i = 1; i <= n; i++)
    deletemax
```

To implement this algorithm, we insert the n elements $a_1, a_2, \ldots, a_n$ to be sorted into a heap that is initially empty. We then perform *deletemax* n times, getting the elements in largest-first order. The arrangement of Fig. 5.50 allows us to store the deleted elements in the tail of the array, as we shrink the heap portion of that array.

Since we just argued in the last section that *insert* and *deletemax* take $O(\log n)$ time each, and since we evidently execute each operation n times, we have an $O(n \log n)$ sorting algorithm, which is comparable to merge sort. In fact, heapsort can be superior to merge sort in a situation in which we only need a few of the largest elements, rather than the entire sorted list. The reason is that we can make the array be a heap in $O(n)$ time, rather than $O(n \log n)$ time, if we use the function `heapify` of Fig. 5.51.

```
void heapify(int A[], int n)
{
    int i;

    for (i = n/2; i >= 1; i--)
        bubbleDown(A, i, n);
}
```

Fig. 5.51. Heapifying an array.

Running Time of Heapify

At first, it might appear that the $n/2$ calls to **bubbleDown** in Fig. 5.51 should take $O(n \log n)$ time in total, because $\log n$ is the only upper bound we know on the running time of **bubbleDown**. However, we can get a tighter bound, $O(n)$, if we exploit the fact that most of the sequences that bubble down elements are very short.

To begin, we did not even have to call **bubbleDown** on the second half of the array, because all the elements there are leaves. On the second quarter of the array, that is,

```
A[(n/4)+1..n/2]
```

we may call **bubbleDown** once, if the element is smaller than either of its children; but those children are in the second half, and therefore are leaves. Thus, in the second quarter of A, we call **bubbleDown** at most once. Similarly, in the second eighth of the array, we call **bubbleDown** at most twice, and so on. The number of calls to **bubbleDown** in the various regions of the array is indicated in Fig. 5.52.

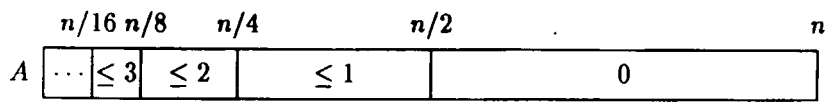


Fig. 5.52. The number of calls to **bubbleDown** decreases rapidly as we go through the array from low to high indices.

Let us count the number of calls to **bubbleDown** made by **heapify**, including recursive calls. From Fig. 5.52 we see that it is possible to divide **A** into *zones*, where the ith zone consists of **A[j]** for j greater than $n/2^{i+1}$ but no greater than $n/2^i$. The number of elements in zone i is thus $n/2^{i+1}$, and there are at most i calls to **bubbleDown** for each element in zone i. Further, the zones $i > \log_2 n$ are empty, since they contain at most $n/2^{1+\log_2 n} = 1/2$ element. The element **A[1]** is the sole occupant of zone $\log_2 n$. We thus need to compute the sum

$$\sum_{i=1}^{\log_2 n} in/2^{i+1} \tag{5.3}$$

We can provide an upper bound on the finite sum (5.3) by extending it to an infinite sum and then pulling out the factor $n/2$:

$$\frac{n}{2}\sum_{i=1}^{\infty} i/2^i \tag{5.4}$$

We must now get an upper bound on the sum in (5.4). This sum, $\sum_{i=1}^{\infty} i/2^i$, can be written as

$$(1/2) + (1/4 + 1/4) + (1/8 + 1/8 + 1/8) + (1/16 + 1/16 + 1/16 + 1/16) + \cdots$$

We can write these inverse powers of 2 as the triangle shown in Fig. 5.53. Each row is an infinite geometric series with ratio 1/2, which sums to twice the first term in the series, as indicated at the right edge of Fig. 5.53. The row sums form another geometric series, which sums to 2.

$$\begin{array}{ccccccccccc}
1/2 & + & 1/4 & + & 1/8 & + & 1/16 & + & \cdots & = & 1 \\
 & & 1/4 & + & 1/8 & + & 1/16 & + & \cdots & = & 1/2 \\
 & & & & 1/8 & + & 1/16 & + & \cdots & = & 1/4 \\
 & & & & & & 1/16 & + & \cdots & = & 1/8 \\
 & & & & & & & & & = &
\end{array}$$

Fig. 5.53. Arranging $\sum_{i=1}^{\infty} i/2^i$ as a triangular sum.

It follows that (5.4) is upper-bounded by $(n/2) \times 2 = n$. That is, the number of calls to `bubbleDown` in the function **heapify** is no greater than n. Since we have already established that each call takes $O(1)$ time, exclusive of any recursive calls, we conclude that the total time taken by **heapify** is $O(n)$.

The Complete Heapsort Algorithm

The C program for heapsort is shown in Fig. 5.54. It uses an array of integers `A[1..MAX]` for the heap. The elements to be sorted are inserted in `A[1..n]`. The definitions of the function declarations in Fig. 5.54 are contained in Sections 5.9 and 5.10.

Line (1) calls **heapify**, which turns the n elements to be sorted into a heap; and line (2) initializes `i`, which marks the end of the heap, to n. The loop of lines (3) and (4) applies **deletemax** $n - 1$ times. We should examine the code of Fig. 5.49 again to observe that `deletemax(A,i)` swaps the maximum element of the remaining heap — which is always in `A[1]` — with $A[i]$. As a side effect, i is decremented by 1, so that the size of the heap shrinks by 1. The element "deleted" by **deletemax** at line (4) is now part of the sorted tail of the array. It is less than or equal to any element in the previous tail, `A[i+1..n]`, but greater than or equal to any element still in the heap. Thus, the claimed property is maintained; all the heap elements precede all the elements of the tail.

Running Time of Heapsort

We have just established that **heapify** in line (1) takes time proportional to n. Line (2) clearly takes $O(1)$ time. Since i decreases by 1 each time around the loop of lines (3) and (4), the number of times around the loop is $n - 1$. The call to **deletemax** at line (4) takes $O(\log n)$ time. Thus, the total time for the loop is $O(n \log n)$. That time dominates lines (1) and (2), and so the running time of **heapsort** is $O(n \log n)$ on n elements.

```
#include <stdio.h>

#define MAX 100

int A[MAX+1];

void bubbleDown(int A[], int i, int n);
void deletemax(int A[], int *pn);
void heapify(int A[], int n);
void heapsort(int A[], int n);
void swap(int A[], int i, int j);

main()
{
    int i, n, x;

    n = 0;
    while (n < MAX && scanf("%d", &x) != EOF)
        A[++n] = x;
    heapsort(A, n);
    for (i = 1; i <= n; i++)
        printf("%d\n", A[i]);
}

void heapsort(int A[], int n)
{
    int i;

(1)     heapify(A, n);
(2)     i = n;
(3)     while (i > 1)
(4)         deletemax(A, &i);
}
```

Fig. 5.54. Heapsorting an array.

EXERCISES

5.10.1: Apply heapsort to the list of elements 3, 1, 4, 1, 5, 9, 2, 6, 5.

5.10.2*: Give an $O(n)$ running time algorithm that finds the $\sqrt{n}$ largest elements in a list of n elements.

5.11 Summary of Chapter 5

The reader should take away the following points from Chapter 5:

- Trees are an important data model for representing hierarchical information.

- Many data structures involving combinations of arrays and pointers can be used to implement trees, and the data structure of choice depends on the operations performed on the tree.
- Two of the most important representations for tree nodes are the leftmost-child–right-sibling representation and the trie (array of pointers to children).
- Recursive algorithms and proofs are well suited for trees. A variant of our basic induction scheme, called structural induction, is effectively a complete induction on the number of nodes in a tree.
- The binary tree is a variant of the tree model in which each node has (optional) left and right children.
- A binary search tree is a labeled binary tree with the "binary search tree property" that all the labels in the left subtree precede the label at a node, and all labels in the right subtree follow the label at the node.
- The dictionary abstract data type is a set upon which we can perform the operations *insert*, *delete*, and *lookup*. The binary search tree efficiently implements dictionaries.
- A priority queue is another abstract data type, a set upon which we can perform the operations *insert* and *deletemax*.
- A partially ordered tree is a labeled binary tree with the property that the label at any node is at least as great as the label at its children.
- Balanced partially ordered trees, where the nodes fully occupy levels from the root to the lowest level, where only the leftmost positions are occupied, can be implemented by an array structure called a heap. This structure provides an $O(\log n)$ implementation of a priority queue and leads to an $O(n \log n)$ sorting algorithm called heapsort.

5.12 Bibliographic Notes for Chapter 5

The trie representation of trees is from Fredkin [1960]. The binary search tree was invented independently by a number of people, and the reader is referred to Knuth [1973] for a history as well as a great deal more information on various kinds of search trees. For more advanced applications of trees, see Tarjan [1983].

Williams [1964] devised the heap implementation of balanced partially ordered trees. Floyd [1964] describes an efficient version of heapsort.

Floyd, R. W. [1964]. "Algorithm 245: Treesort 3," *Comm. ACM* **7**:12, pp. 701.

Fredkin, E. [1960]. "Trie memory," *Comm. ACM* **3**:4, pp. 490–500.

Knuth, D. E. [1973]. *The Art of Computer Programming*, Vol. III, *Sorting and Searching*, 2nd ed., Addison-Wesley, Reading, Mass.

Tarjan, R. E. [1983]. *Data Structures and Network Algorithms*, SIAM Press, Philadelphia.

Williams, J. W. J. [1964]. "Algorithm 232: Heapsort," *Comm. ACM* **7**:6, pp. 347–348.

CHAPTER 9

The Graph Data Model

A graph is, in a sense, nothing more than a binary relation. However, it has a powerful visualization as a set of points (called nodes) connected by lines (called edges) or by arrows (called arcs). In this regard, the graph is a generalization of the tree data model that we studied in Chapter 5. Like trees, graphs come in several forms: directed/undirected, and labeled/unlabeled.

Also like trees, graphs are useful in a wide spectrum of problems such as computing distances, finding circularities in relationships, and determining connectivities. We have already seen graphs used to represent the structure of programs in Chapter 2. Graphs were used in Chapter 7 to represent binary relations and to illustrate certain properties of relations, like commutativity. We shall see graphs used to represent automata in Chapter 10 and to represent electronic circuits in Chapter 13. Several other important applications of graphs are discussed in this chapter.

9.1 What This Chapter Is About

The main topics of this chapter are

- The definitions concerning directed and undirected graphs (Sections 9.2 and 9.10).
- The two principal data structures for representing graphs: adjacency lists and adjacency matrices (Section 9.3).
- An algorithm and data structure for finding the connected components of an undirected graph (Section 9.4).
- A technique for finding minimal spanning trees (Section 9.5).
- A useful technique for exploring graphs, called "depth-first search" (Section 9.6).

- Applications of depth-first search to test whether a directed graph has a cycle, to find a topological order for acyclic graphs, and to determine whether there is a path from one node to another (Section 9.7).
- Dijkstra's algorithm for finding shortest paths (Section 9.8). This algorithm finds the minimum distance from one "source" node to every node.
- Floyd's algorithm for finding the minimum distance between any two nodes (Section 9.9).

Many of the algorithms in this chapter are examples of useful techniques that are much more efficient than the obvious way of solving the given problem.

9.2 Basic Concepts

Directed graph

A *directed graph*, consists of

1. A set N of *nodes* and

Nodes and arcs

2. A binary relation A on N. We call A the set of *arcs* of the directed graph. Arcs are thus pairs of nodes.

Graphs are drawn as suggested in Fig. 9.1. Each node is represented by a circle, with the name of the node inside. We shall usually name the nodes by integers starting at 0, or we shall use an equivalent enumeration. In Fig. 9.1, the set of nodes is $N = \{0, 1, 2, 3, 4\}$.

Each arc (u, v) in A is represented by an arrow from u to v. In Fig. 9.1, the set of arcs is

$$A = \{(0,0),\ (0,1),\ (0,2),\ (1,3),\ (2,0),\ (2,1),\ (2,4),\ (3,2),\ (3,4),\ (4,1)\}$$

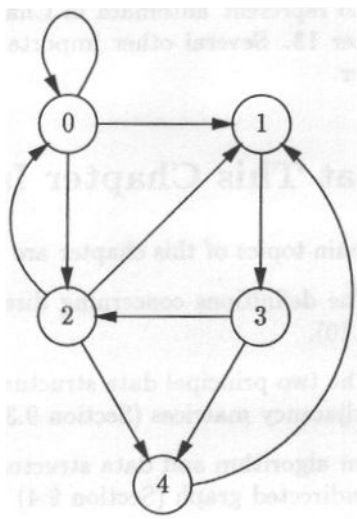


Fig. 9.1. Example of a directed graph.

Head and tail

In text, it is customary to represent an arc (u, v) as $u \rightarrow v$. We call v the *head* of the arc and u the *tail* to conform with the notion that v is at the head of the

arrow and u is at its tail. For example, $0 \rightarrow 1$ is an arc of Fig. 9.1; its head is node 1 and its tail is node 0. Another arc is $0 \rightarrow 0$; such an arc from a node to itself is called a *loop*. For this arc, both the head and the tail are node 0.

Loop

Predecessors and Successors

When $u \rightarrow v$ is an arc, we can also say that u is a *predecessor* of v, and that v is a *successor* of u. Thus, the arc $0 \rightarrow 1$ tells us that 0 is a predecessor of 1 and that 1 is a successor of 0. The arc $0 \rightarrow 0$ tells us that node 0 is both a predecessor and a successor of itself.

Labels

As for trees, it is permissible to attach a *label* to each node. Labels will be drawn near their node. Similarly, we can label arcs by placing the label near the middle of the arc. Any type can be used as a node label or an arc label. For instance, Fig. 9.2 shows a node named 1, with a label "dog," a node named 2, labeled "cat," and an arc $1 \rightarrow 2$ labeled "bites."

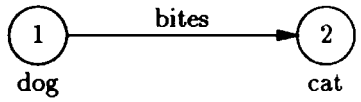


Fig. 9.2. A labeled graph with two nodes.

Again as with trees, we should not confuse the name of a node with its label. Node names must be unique in a graph, but two or more nodes can have the same label.

Paths

A *path* in a directed graph is a list of nodes $(v_1, v_2, \ldots, v_k)$ such that there is an arc from each node to the next, that is, $v_i \rightarrow v_{i+1}$ for $i = 1, 2, \ldots, k-1$. The *length* of the path is $k-1$, the number of arcs along the path. For example, $(0, 1, 3)$ is a path of length two in Fig. 9.1.

Length of a path

The trivial case $k = 1$ is permitted. That is, any node v by itself is a path of length zero from v to v. This path has no arcs.

Cyclic and Acyclic Graphs

A *cycle* in a directed graph is a path of length 1 or more that begins and ends at the same node. The *length of the cycle* is the length of the path. Note that a trivial path of length 0 is not a cycle, even though it "begins and ends at the same node." However, a path consisting of a single arc $v \rightarrow v$ is a cycle of length 1.

Length of a cycle

✦ **Example 9.1.** Consider the graph of Fig. 9.1. There is a cycle $(0, 0)$ of length 1 because of the loop $0 \rightarrow 0$. There is a cycle $(0, 2, 0)$ of length 2 because of the arcs $0 \rightarrow 2$ and $2 \rightarrow 0$. Similarly, $(1, 3, 2, 1)$ is a cycle of length 3, and $(1, 3, 2, 4, 1)$ is a cycle of length 4. ✦

Equivalent cycles

Note that a cycle can be written to start and end at any of its nodes. That is, the cycle $(v_1, v_2, \ldots, v_k, v_1)$ could also be written as $(v_2, \ldots, v_k, v_1, v_2)$ or as $(v_3, \ldots, v_k, v_1, v_2, v_3)$, and so on. For example, the cycle $(1, 3, 2, 4, 1)$ could also have been written as $(2, 4, 1, 3, 2)$.

Simple cycle

On every cycle, the first and last nodes are the same. We say that a cycle $(v_1, v_2, \ldots, v_k, v_1)$ is *simple* if no node appears more than once among $v_1, \ldots, v_k$; that is, the only repetition in a simple cycle occurs at the final node.

✦ **Example 9.2.** All the cycles in Example 9.1 are simple. In Fig. 9.1 the cycle $(0, 2, 0)$ is simple. However, there are cycles that are not simple, such as $(0, 2, 1, 3, 2, 0)$ in which node 2 appears twice. ✦

Given a nonsimple cycle containing node v, we can find a simple cycle containing v. To see why, write the cycle to begin and end at v, as in $(v, v_1, v_2, \ldots, v_k, v)$. If the cycle is not simple, then either

1. v appears three or more times, or
2. There is some node u other than v that appears twice; that is, the cycle must look like $(v, \ldots, u, \ldots, u, \ldots, v)$.

In case (1), we can remove everything up to, but not including, the next-to-last occurrence of v. The result is a shorter cycle from v to v. In case (2), we can remove the section from u to u, replacing it by a single occurrence of u, to get the cycle $(v, \ldots, u, \ldots, v)$. The result must still be a cycle in either case, because each arc of the result is present in the original cycle, and therefore is present in the graph.

It may be necessary to repeat this transformation several times before the cycle becomes simple. Since the cycle always gets shorter with each iteration, eventually we must arrive at a simple cycle. What we have just shown is that if there is a cycle in a graph, then there must be at least one simple cycle.

✦ **Example 9.3.** Given the cycle $(0, 2, 1, 3, 2, 0)$, we can remove the first 2 and the following 1, 3 to get the simple cycle $(0, 2, 0)$. In physical terms, we started with the cycle that begins at 0, goes to 2, then 1, then 3, back to 2, and finally back to 0. The first time we are at 2, we can pretend it is the second time, skip going to 1 and 3, and proceed right back to 0.

For another example, consider the nonsimple cycle $(0, 0, 0)$. As 0 appears three times, we remove the first 0, that is, everything up to but not including the next-to-last 0. Physically, we have replaced the path in which we went around the loop $0 \rightarrow 0$ twice by the path in which we go around once. ✦

Cyclic graph

If a graph has one or more cycles, we say the graph is *cyclic*. If there are no cycles, the graph is said to be *acyclic*. By what we just argued about simple cycles, a graph is cyclic if and only if it has a simple cycle, because if it has any cycles at all, it will have a simple cycle.

✦ **Example 9.4.** We mentioned in Section 3.8 that we could represent the calls

Calling graph

made by a collection of functions with a directed graph called the "calling graph." The nodes are the functions, and there is an arc $P \rightarrow Q$ if function P calls function Q. For instance, Fig. 9.3 shows the calling graph associated with the merge sort algorithm of Section 2.9.

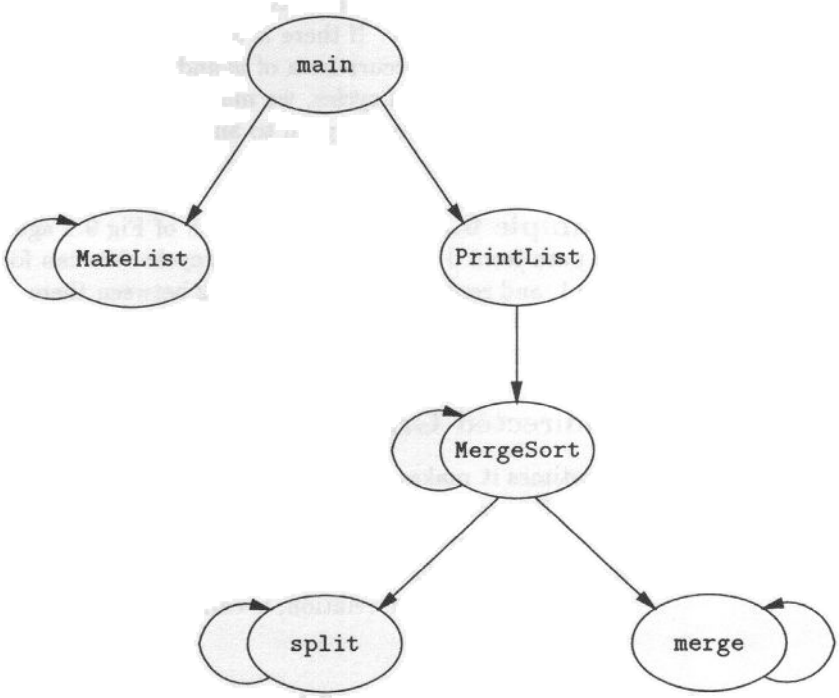
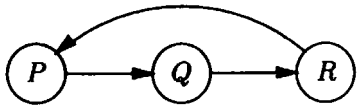


Fig. 9.3. Calling graph for the mergesort algorithm.

Direct and indirect recursion

The existence of a cycle in the calling graph implies a recursion in the algorithm. In Fig. 9.3 there are four simple cycles, one around each of the nodes **MakeList**, **MergeSort**, **split**, and **merge**. Each cycle is a trivial loop. Recall that all these functions call themselves, and thus are recursive. Recursions in which a function calls itself are by far the most common kind, and each of these appears as a loop in the calling graph. We call these recursions *direct*. However, one occasionally sees an *indirect* recursion, in which there is a cycle of length greater than 1. For instance, the graph



represents a function P that calls function Q, which calls function R, which calls function P. ◆

Acyclic Paths

A path is said to be *acyclic* if no node appears more than once on the path. Clearly, no cycle is acyclic. The argument that we just gave to show that for every cycle there is a simple cycle also demonstrates the following principle. If there is any path at all from u to v, then there is an acyclic path from u to v. To see why, start with any path from u to v. If there is a repetition of some node w, which could be u or v, replace the two occurrences of w and everything in between by one occurrence of w. As for the case of cycles, we may have to repeat this process several times, but eventually we reduce the path to an acyclic path.

✦ **Example 9.5.** Consider the graph of Fig 9.1 again. The path $(0, 1, 3, 2, 1, 3, 4)$ is a path from 0 to 4 that contains a cycle. We can focus on the two occurrences of node 1, and replace them, and the $3, 2$ between them, by 1, leaving $(0, 1, 3, 4)$, which is an acyclic path because no node appears twice. We could also have obtained the same result by focusing on the two occurrences of node 3. ✦

Undirected Graphs

Edge

Neighbors

Sometimes it makes sense to connect nodes by lines that have no direction, called *edges*. Formally, an edge is a set of two nodes. The edge $\{u, v\}$ says that nodes u and v are connected in both directions.[1] If $\{u, v\}$ is an edge, then nodes u and v are said to be *adjacent* or to be *neighbors*. A graph with edges, that is, a graph with a symmetric arc relation, is called an *undirected graph*.

✦ **Example 9.6.** Figure 9.4 represents a partial road map of the Hawaiian Islands, indicating some of the principal cities. Cities with a road between them are indicated by an edge, and the edge is labeled by the driving distance. It is natural to represent roads by edges, rather than arcs, because roads are normally two-way. ✦

Paths and Cycles in Undirected Graphs

A *path* in an undirected graph is a list of nodes $(v_1, v_2, \ldots, v_k)$ such that each node and the next are connected by an edge. That is, $\{v_i, v_{i+1}\}$ is an edge for $i = 1, 2, \ldots, k-1$. Note that edges, being sets, do not have their elements in any particular order. Thus, the edge $\{v_i, v_{i+1}\}$ could just as well appear as $\{v_{i+1}, v_i\}$.

The *length* of the path $(v_1, v_2, \ldots, v_k)$ is $k-1$. As with directed graphs, a node by itself is a path of length 0.

Defining cycles in undirected graphs is a little tricky. The problem is that we do not want to consider a path such as (u, v, u), which exists whenever there is an edge $\{u, v\}$, to be a cycle. Similarly, if $(v_1, v_2, \ldots, v_k)$ is a path, we can traverse it forward and backward, but we do not want to call the path

$$(v_1, v_2, \ldots, v_{k-1}, v_k, v_{k-1}, \ldots, v_2, v_1)$$

---

[1] Note that the edge is required to have exactly two nodes. A singleton set consisting of one node is not an edge. Thus, although an arc from a node to itself is permitted, we do not permit a looping edge from a node to itself. Some definitions of "undirected graph" do permit such loops.

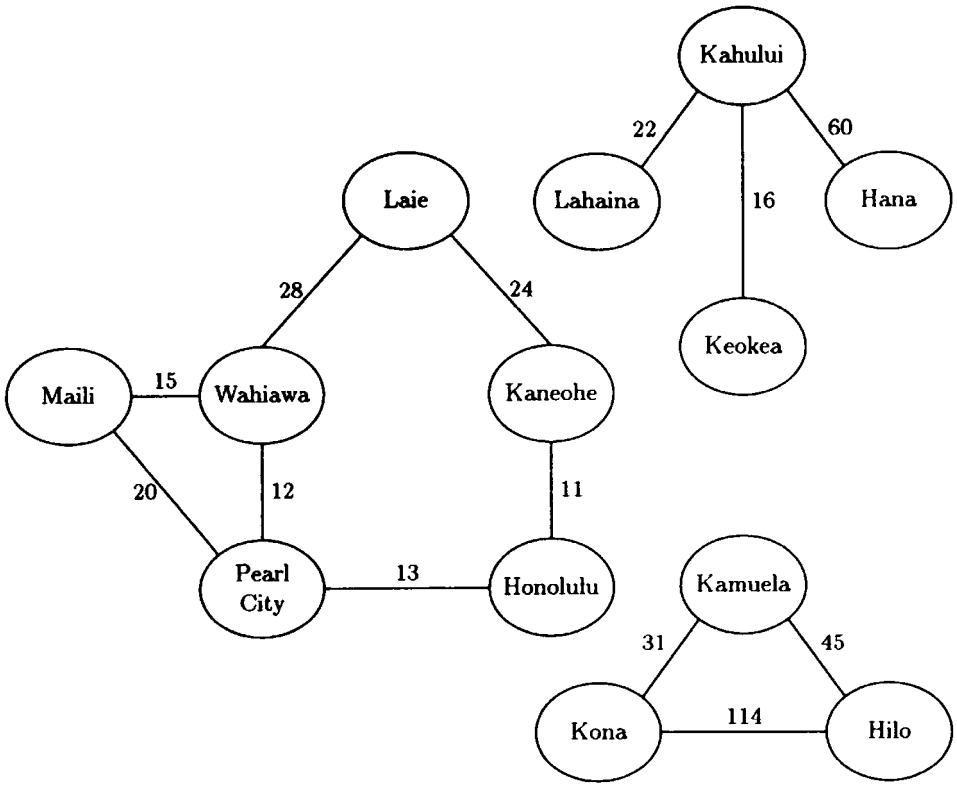


Fig. 9.4. An undirected graph representing roads in three Hawaiian Islands Oahu, Maui, and Hawaii (clockwise from the left).

a cycle.

Simple cycle

Perhaps the easiest approach is to define a *simple cycle* in an undirected graph to be a path of length three or more that begins and ends at the same node, and with the exception of the last node does not repeat any node. The notion of a nonsimple cycle in an undirected graph is not generally useful, and we shall not pursue this concept.

Equivalent cycles

As with directed cycles, we regard two undirected cycles as the same if they consist of the same nodes in the same order, with a different starting point. Undirected cycles are also the same if they consist of the same nodes in reverse order. Formally, the simple cycle $(v_1, v_2, \ldots, v_k)$ is equivalent, for each i between 1 and k, to the cycle $(v_i, v_{i+1}, \ldots, v_k, v_1, v_2, \ldots, v_{i-1})$ and to the cycle

$$(v_i, v_{i-1}, \ldots, v_1, v_k, v_{k-1}, \ldots, v_{i+1})$$

✦ **Example 9.7.** In Fig. 9.4,

(Wahiawa, Pearl City, Maili, Wahiawa)

is a simple cycle of length three. It could have been written equivalently as

(Maili, Wahiawa, Pearl City, Maili)

by starting at Maili and proceeding in the same order around the circle. Likewise, it could have been written to start at Pearl City and proceed around the circle in reverse order:

(Pearl City, Maili, Wahiawa, Pearl City)

For another example,

(Laie, Wahiawa, Pearl City, Honolulu, Kaneohe, Laie)

is a simple cycle of length five. ✦

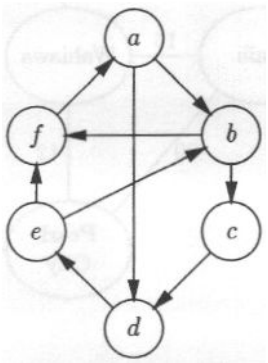


Fig. 9.5. Directed graph for Exercises 9.2.1 and 9.2.2.

EXERCISES

9.2.1: Consider the graph of Fig. 9.5.

a) How many arcs are there?
b) How many acyclic paths are there from node a to node d? What are they?
c) What are the predecessors of node b?
d) What are the successors of node b?
e) How many simple cycles are there? List them. Do not repeat paths that differ only in the starting point (see Exercise 9.2.8).
f) List all the nonsimple cycles of length up to 7.

9.2.2: Consider the graph of Fig. 9.5 to be an undirected graph, by replacing each arc $u \rightarrow v$ by an edge $\{u, v\}$.

a) Find all the paths from a to d that do not repeat any node.
b) How many simple cycles are there that include all six nodes? List these cycles.
c) What are the neighbors of node a?

9.2.3*: If a graph has 10 nodes, what is the largest number of arcs it can have? What is the smallest possible number of arcs? In general, if a graph has n nodes, what are the minimum and maximum number of arcs?

9.2.4*: Repeat Exercise 9.2.3 for the edges of an undirected graph.

9.2.5**: If a directed graph is acyclic and has n nodes, what is the largest possible number of arcs?

9.2.6: Find an example of indirect recursion among the functions so far in this book.

9.2.7: Write the cycle $(0, 1, 2, 0)$ in all possible ways.

9.2.8*: Let G be a directed graph and let R be the relation on the cycles of G defined by $(u_1, \ldots, u_k, u_1)R(v_1, \ldots, v_k, v_1)$ if and only if $(u_1, \ldots, u_k, u_1)$ and $(v_1, \ldots, v_k, v_1)$ represent the same cycle. Show that R is an equivalence relation on the cycles of G.

9.2.9*: Show that the relation S defined on the nodes of a graph by uSv if and only if $u = v$ or there is some cycle that includes both nodes u and v, is an equivalence relation.

9.2.10*: When we discussed simple cycles in undirected graphs, we mentioned that two cycles were really the same if they were the same nodes, either in order, or in reverse order, but with a different starting point. Show that the relation R consisting of pairs of representations for the same simple cycle is an equivalence relation.

9.3 Implementation of Graphs

There are two standard ways to represent a graph. One, called *adjacency lists*, is familiar from the implementation of binary relations in general. The second, called *adjacency matrices*, is a new way to represent binary relations, and is more suitable for relations where the number of pairs is a sizable fraction of the total number of pairs that could possibly exist over a given domain. We shall consider these representations, first for directed graphs, then for undirected graphs.

Adjacency Lists

Let nodes be named either by the integers $0, 1, \ldots, MAX - 1$ or by an equivalent enumerated type. In general, we shall use `NODE` as the type of nodes, but we may suppose that `NODE` is a synonym for `int`. Then we can use the generalized characteristic-vector approach, introduced in Section 7.9, to represent the set of arcs. This representation is called *adjacency lists*. We define linked lists of nodes by

```
typedef struct CELL *LIST;
struct CELL {
    NODE nodeName;
    LIST next;
};
```

and then create an array

```
LIST successors[MAX];
```

That is, the entry `successors[u]` contains a pointer to a linked list of all the successors of node u.

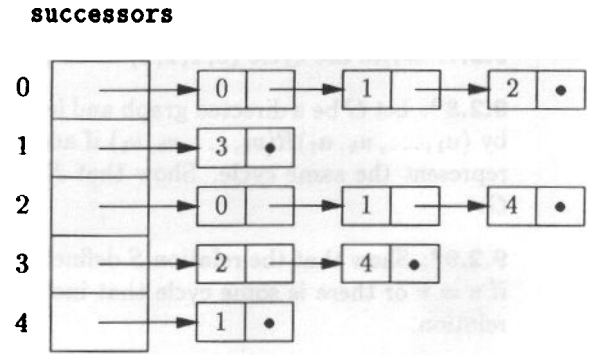


Fig. 9.6. Adjacency-list representation of the graph shown in Fig. 9.1.

✦ **Example 9.8.** The graph of Fig. 9.1 can be represented by the adjacency lists shown in Fig. 9.6. We have sorted the adjacency lists by node number, but the successors of a node can appear in any order on its adjacency list. ✦

Adjacency Matrices

Another common way to represent directed graphs is as *adjacency matrices*. We can create a two-dimensional array

```
BOOLEAN arcs[MAX][MAX];
```

in which the value of `arcs[u][v]` is `TRUE` if there is an arc $u \rightarrow v$, and `FALSE` if not.

✦ **Example 9.9.** The adjacency matrix for the graph of Fig. 9.1 is shown in Fig. 9.7. We use 1 for `TRUE` and 0 for `FALSE`. ✦

	0	1	2	3	4
0	1	1	1	0	0
1	0	0	0	1	0
2	1	1	0	0	1
3	0	0	1	0	1
4	0	1	0	0	0

Fig. 9.7. Adjacency matrix representing the graph of Fig. 9.1.

Operations on Graphs

We can see some of the distinctions between the two graph representations if we consider some simple operations on graphs. Perhaps the most basic operation is to determine whether there is an arc $u \rightarrow v$ from a node u to a node v. In the adjacency matrix, it takes $O(1)$ time to look up `arcs[u][v]` to see whether the entry there is `TRUE` or not.

Comparison of Adjacency Matrices and Adjacency Lists

Dense and sparse graphs

We tend to prefer adjacency matrices when the graphs are *dense*, that is, when the number of arcs is near the maximum possible number, which is n^2 for a graph of n nodes. However, if the graph is *sparse*, that is, if most of the possible arcs are not present, then the adjacency-list representation may save space. To see why, note that an adjacency matrix for an n-node graph has n^2 bits (provided we represent `TRUE` and `FALSE` by single bits rather than integers as we have done in this section).

In a typical computer, a structure consisting of an integer and a pointer, like our adjacency list cells, will use 32 bits to represent the integer and 32 bits to represent the pointer, or 64 bits total. Thus, if the number of arcs is a, we need about $64a$ bits for the lists, and $32n$ bits for the array of n headers. The adjacency list will use less space than the adjacency matrix if $32n + 64a < n^2$, that is, if $a < n^2/64 - n/2$. If n is large, we can neglect the $n/2$ term and approximate the previous inequality by $a < n^2/64$, that is, if fewer than 1/64th of the possible arcs are actually present. More detailed arguments favoring one or the other representation are presented when we discuss operations on graphs. The following table summarizes the preferred representations for various operations.

OPERATION	DENSE GRAPH	SPARSE GRAPH
Look up an arc	Adjacency matrix	Either
Find successors	Either	Adjacency lists
Find predecessors	Adjacency matrix	Either

With adjacency lists, it takes $O(1)$ time to find the header of the adjacency list for u. We must then traverse this list to the end if v is not there, or half the way down the list on the average if v is present. If there are a arcs and n nodes in the graph, then we take time $O(1 + a/n)$ on the average to do the lookup. If a is no more than a constant factor times n, this quantity is $O(1)$. However, the larger a is when compared with n, the longer it takes to tell whether an arc is present using the adjacency list representation. In the extreme case where a is around n^2, its maximum possible value, there are around n nodes on each adjacency list. In this case, it takes $O(n)$ time on the average to find a given arc. Put another way, the denser a graph is, the more we prefer the adjacency matrix to adjacency lists, when we need to look up a given arc.

On the other hand, we often need to find all the successors of a given node u. Using adjacency lists, we go to `successors[u]` and traverse the list, in average time $O(a/n)$, to find all the successors. If a is comparable to n, then we find all the successors of u in $O(1)$ time. But with adjacency matrices, we must examine the entire row for node u, taking $O(n)$ time no matter what a is. Thus, for graphs with a small number of edges per node, adjacency lists are much faster than adjacency matrices when we need to examine all the successors of a given node.

However, suppose we want to find all the predecessors of a given node v. With an adjacency matrix, we can examine the column for v; a 1 in the row for u means that u is a predecessor of v. This examination takes $O(n)$ time. The adjacency-list representation gives us no help finding predecessors. We must examine the adjacency list for every node u, to see if that list includes v. Thus, we may examine

A Matter of Degree

In- and Out-degree

The number of arcs out of a node v is called the *out-degree* of v. Thus, the out-degree of a node equals the length of its adjacency list; it also equals the number of 1's in the row for v in the adjacency matrix. The number of arcs into node v is the *in-degree* of v. The in-degree measures the number of times v appears on the adjacency list of some node, and it is the number of 1's found in the column for v in the adjacency matrix.

Degree of a graph

In an undirected graph, we do not distinguish between edges coming in or going out of a node. For an undirected graph, the *degree* of node v is the number of neighbors of v, that is, the number of edges $\{u, v\}$ containing v for some node u. Remember that in a set, order of members is unimportant, so $\{u, v\}$ and $\{v, u\}$ are the same edge, and are counted only once. The *degree of an undirected graph* is the maximum degree of any node in the graph. For example, if we regard a binary tree as an undirected graph, its degree is 3, since a node can only have edges to its parent, its left child, and its right child. For a directed graph, we can say that the *in-degree of a graph* is the maximum of the in-degrees of its nodes, and likewise, the *out-degree of a graph* is the maximum of the out-degrees of its nodes.

all the cells of all the adjacency lists, and we shall probably examine most of them. Since the number of cells in the entire adjacency list structure is equal to a, the number of arcs of the graph, the time to find predecessors using adjacency lists is thus $O(a)$ on a graph of a arcs. Here, the advantage goes to the adjacency matrix; and the denser the graph, the greater the advantage.

Implementing Undirected Graphs

Symmetric adjacency matrix

If a graph is undirected, we can pretend that each edge is replaced by arcs in both directions, and represent the resulting directed graph by either adjacency lists or an adjacency matrix. If we use an adjacency matrix, the matrix is *symmetric*. That is, if we call the matrix **edges**, then $edges[u][v] = edges[v][u]$. If we use an adjacency-list representation, then the edge $\{u, v\}$ is represented twice. We find v on the adjacency list for u and we find u on the list for v. That arrangement is often useful, since we cannot tell in advance whether we are more likely to follow the edge $\{u, v\}$ from u to v or from v to u.

	Laie	Kaneohe	Honolulu	PearlCity	Maili	Wahiawa
Laie	0	1	0	0	0	1
Kaneohe	1	0	1	0	0	0
Honolulu	0	1	0	1	0	0
PearlCity	0	0	1	0	1	1
Maili	0	0	0	1	0	1
Wahiawa	1	0	0	1	1	0

Fig. 9.8. Adjacency-matrix representation of an undirected graph from Fig. 9.4.

✦ **Example 9.10.** Consider how to represent the largest component of the undirected graph of Fig. 9.4 (which represents six cities on the island of Oahu). For the moment, we shall ignore the labels on the edges. The adjacency matrix representation is shown in Fig. 9.8. Notice that the matrix is symmetric.

Figure 9.9 shows the representation by adjacency lists. In both cases, we are using an enumeration type

```
enum CITYTYPE {Laie, Kaneohe, Honolulu,
    PearlCity, Maili, Wahiawa};
```

to index arrays. That arrangement is somewhat rigid, since it does not allow any changes in the set of nodes of the graph. We shall give a similar example shortly where we name nodes explicitly by integers, and use city names as node labels, for more flexibility in changing the set of nodes. ✦

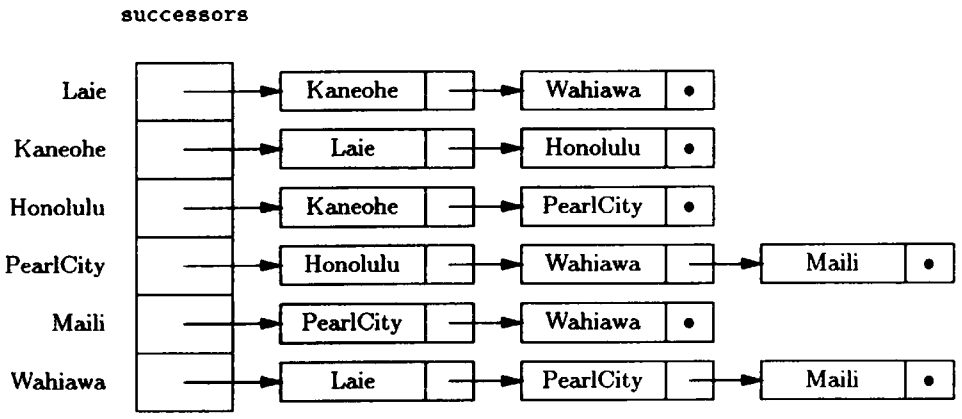


Fig. 9.9. Adjacency-list representation of an undirected graph from Fig. 9.4.

Representing Labeled Graphs

Suppose a graph has labels on its arcs (or edges if it is undirected). Using an adjacency matrix, we can replace the 1 that represents the presence of arc $u \rightarrow v$ in the graph by the label of this arc. It is necessary that we have some value that is permissible as a matrix entry but cannot be mistaken for a label; we use this value to represent the absence of an arc.

If we represent the graph by adjacency lists, we add to the cells forming the lists an additional field **nodeLabel**. If there is an arc $u \rightarrow v$ with label L, then on the adjacency list for node u we shall find a cell with v in its **nodeName** field and L in its **nodeLabel** field. That value represents the label of the arc.

We represent labels on nodes in a different way. For an adjacency matrix, we simply create another array, say **NodeLabels**, and let **NodeLabels[u]** be the label of node u. When we use adjacency lists, we already have an array of headers indexed by nodes. We change this array so that it has elements that are structures, one field for the node label and one field pointing to the beginning of the adjacency list.

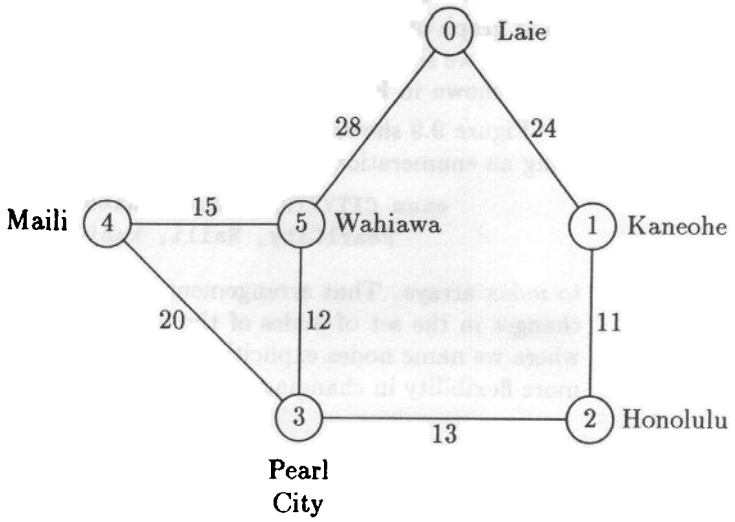


Fig. 9.10. Map of Oahu with nodes named by integers and labeled by cities.

cities

0	Laie
1	Kaneohe
2	Honolulu
3	PearlCity
4	Maili
5	Wahiawa

distances

	0	1	2	3	4	5
0	−1	24	−1	−1	−1	28
1	24	−1	11	−1	−1	−1
2	−1	11	−1		−1	−1
3	−1	−1		−1	20	12
4	−1	−1	−1		−1	15
5	28	−1	−1		15	−1

Fig. 9.11. Adjacency-matrix representation of a directed graph.

✦ **Example 9.11.** Let us again represent the large component of the graph of Fig. 9.4, but this time, we shall incorporate the edge labels, which are distances. Furthermore, we shall give the nodes integer names, starting with 0 for Laie, and proceeding clockwise. The city names themselves are indicated by node labels. We shall take the type of node labels to be character arrays of length 32. This representation is more flexible than that of Example 9.10, since if we allocate extra places in the array, we can add cities should we wish. The resulting graph is redrawn

in Fig. 9.10, and the adjacency matrix representation is in Fig. 9.11.

Notice that there are really two parts to this representation: the array `cities`, indicating the city that each of the integers 0 through 5 stands for, and the matrix `distances`, indicating the presence or absence of edges and the labels of present edges. We use -1 as a value that cannot be mistaken for a label, since in this example, labels, representing distances, must be positive.

We could declare this structure as follows:

```
typedef char CITYTYPE[32];
typedef CITYTYPE cities[MAX];
int distances[MAX][MAX];
```

Here, MAX is some number at least 6; it limits the number of nodes that can ever appear in our graph. `CITYTYPE` is defined to be 32-character arrays, and the array `cities` gives the labels of the various nodes. For example, we expect *cities*[0] to be `"Laie"`.

An alternative representation of the graph of Fig. 9.10 is by adjacency lists. Suppose the constant MAX and the type `CITYTYPE` are as above. We define the types `CELL` and `LIST` by

```
typedef struct CELL *LIST;
struct CELL {
    NODE nodeName;
    int distance;
    LIST next;
};
```

Next, we declare the array `cities` by

```
struct {
    CITYTYPE city;
    LIST adjacent;
} cities[MAX];
```

Figure 9.12 shows the graph of Fig. 9.10 represented in this manner. ✦

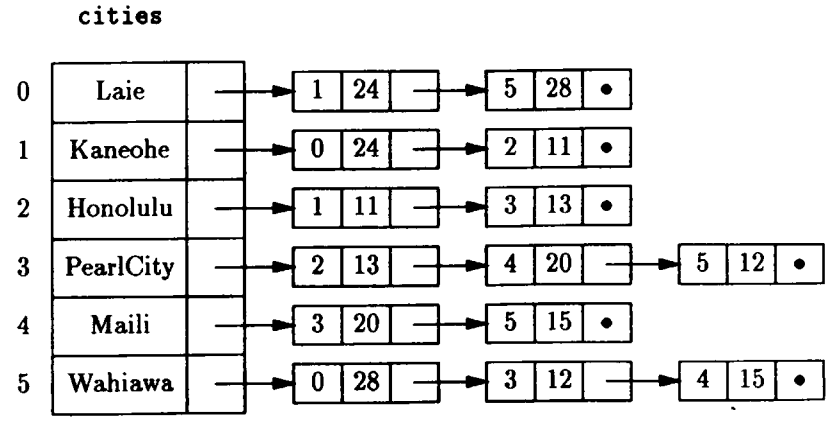


Fig. 9.12. Adjacency-list representation of graph with node and edge labels.

EXERCISES

9.3.1: Represent the graph of Fig. 9.5 (see the exercises of Section 9.2) by

a) Adjacency lists

b) An adjacency matrix

Give the appropriate type definitions in each case.

9.3.2: Suppose the arcs of Fig. 9.5 were instead edges (i.e., the graph were undirected). Repeat Exercise 9.3.1 for the undirected graph.

9.3.3: Let us label each of the arcs of the directed graph of Fig. 9.5 by the character string of length 2 consisting of the tail followed by the head. For example, the arc $a \rightarrow b$ is labeled by the character string `ab`. Also, suppose each node is labeled by the capital letter corresponding to its name. For instance, the node named a is labeled `A`. Repeat Exercise 9.3.1 for this labeled, directed graph.

9.3.4*: What is the relationship between the adjacency-matrix representation of an unlabeled graph and the characteristic-vector representation of a set of arcs?

9.3.5*: Prove by induction on n that in an undirected graph of n nodes, the sum of the degrees of the nodes is twice the number of edges. *Note*. A proof without using induction is also possible, but here an inductive proof is required.

9.3.6: Design algorithms to insert and delete arcs from an (a) adjacency-matrix (b) adjacency-list representation of a directed graph.

9.3.7: Repeat Exercise 9.3.6 for an undirected graph.

9.3.8: We can add a "predecessor list" to the adjacency-list representation of a directed or undirected graph. When is this representation preferred for the operations of

a) Looking up an arc?

b) Finding all successors?

c) Finding all predecessors?

Consider both dense and sparse graphs in your analysis.

9.4 Connected Components of an Undirected Graph

Connected graph

We can divide any undirected graph into one or more *connected components*. Each connected component is a set of nodes with paths from any member of the component to any other. Moreover, the connected components are maximal, that is, for no node in the component is there a path to any node outside the component. If a graph consists of a single connected component, then we say the graph is *connected*.

Physical Interpretation of Connected Components

If we are given a drawing of an undirected graph, it is easy to see the connected components. Imagine that the edges are strings. If we pick up any node, the connected component of which it is a member will come up with it, and members of all other connected components will stay where they are. Of course, what is easy to do by "eyeball" is not necessarily easy to do by computer. An algorithm to find the connected components of a graph is the principal subject of this section.

✦ **Example 9.12.** Consider again the graph of the Hawaiian Islands in Fig. 9.4. There are three connected components, corresponding to three islands. The largest component consists of Laie, Kaneohe, Honolulu, Pearl City, Maili, and Wahiawa. These are cities on the island of Oahu, and they are clearly mutually connected by roads, that is, by paths of edges. Also, clearly, there are no roads leading from Oahu to any other island. In graph-theoretic terms, there are no paths from any of the six cities mentioned above to any of the other cities in Fig. 9.4.

A second component consists of the cities of Lahaina, Kahului, Hana, and Keokea; these are cities on the island of Maui. The third component is the cities of Hilo, Kona, and Kamuela, on the "big island" of Hawaii. ✦

Connected Components as Equivalence Classes

Another useful way to look at connected components is that they are the equivalence classes of the equivalence relation P defined on the nodes of the undirected graph by: uPv if and only if there is a path from u to v. It is easy to check that P is an equivalence relation.

1. P is reflexive, that is, uPu for any node u, since there is a path of length 0 from any node to itself.
2. P is symmetric. If uPv, then there is a path from u to v. Since the graph is undirected, the reverse sequence of nodes is also a path. Thus vPu.
3. P is transitive. Suppose uPw and wPv are true. Then there is a path, say

   $$(x_1, x_2, \ldots, x_j)$$

   from u to w. Here, $u = x_1$ and $w = x_j$. Also, there is a path $(y_1, y_2, \ldots, y_k)$ from w to v where $w = y_1$ and $v = y_k$. If we put these paths together, we get a path from u to v, namely

   $$(u = x_1, x_2, \cdots, x_j = w = y_1, y_2, \cdots, y_k = v)$$

✦ **Example 9.13.** Consider the path

(Honolulu, PearlCity, Wahiawa, Maili)

from Honolulu to Maili in Fig. 9.10. Also consider the path

(Maili, PearlCity, Wahiawa, Laie)

from Maili to Laie in the same graph. If we put these paths together, we get a path from Honolulu to Laie:

(Honolulu, PearlCity, Wahiawa, Maili, PearlCity, Wahiawa, Laie)

It happens that this path is cyclic. As mentioned in Section 9.2, we can always remove cycles to get an acyclic path. In this case, one way to do so is to replace the two occurrences of Wahiawa and the nodes in between by one occurrence of Wahiawa to get

(Honolulu, PearlCity, Wahiawa, Laie)

which is an acyclic path from Honolulu to Laie. ✦

Since P is an equivalence relation, it partitions the set of nodes of the undirected graph in question into equivalence classes. The class containing node v is the set of nodes u such that vPu, that is, the set of nodes connected to v by a path. Moreover, another property of equivalence classes is that if nodes u and v are in different classes, then it is not possible that uPv; that is, there is never a path from a node in one equivalence class to a node in another. Thus, the equivalence classes defined by the "path" relation P are exactly the connected components of the graph.

An Algorithm for Computing the Connected Components

Suppose we want to construct the connected components of a graph G. One approach is to begin with a graph G_0 consisting of the nodes of G with none of the edges. We then consider the edges of G, one at a time, to construct a sequence of graphs $G_0, G_1, \ldots$, where G_i consists of the nodes of G and the first i edges of G.

BASIS. G_0 consists of only the nodes of G with none of the edges. Every node is in a component by itself.

INDUCTION. Suppose we have the connected components for the graph G_i after considering the first i edges, and we now consider the $(i+1)$st edge, $\{u, v\}$.

1. If u and v are in the same component of G_i, then G_{i+1} has the same set of connected components as G_i, because the new edge does not connect any nodes that were not already connected.

2. If u and v are in different components, we merge the components containing u and v to get the connected components for G_{i+1}. Figure 9.13 suggests why there is a path from any node x in the component of u, to any node y in the component of v. We follow the path in the first component from x to u, then the edge $\{u, v\}$, and finally the path from v to y that we know exists in the second component.

When we have considered all edges in this manner, we have the connected components of the full graph.

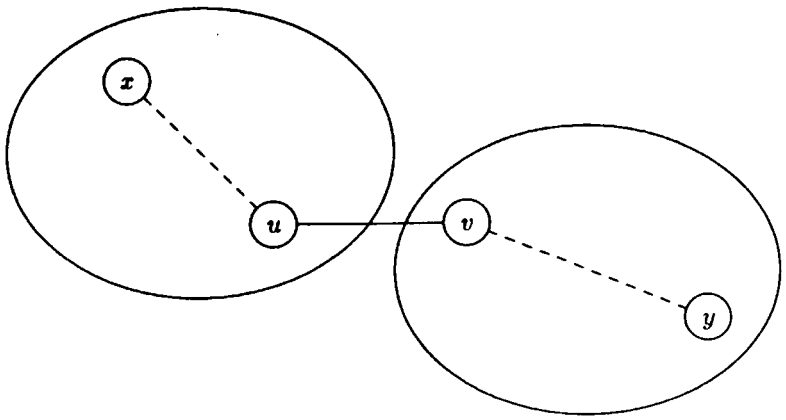


Fig. 9.13. Adding edge $\{u, v\}$ connects the components containing u and v.

✦ **Example 9.14.** Let us consider the graph of Fig. 9.4. We can consider edges in any order, but for reasons having to do with an algorithm in the next section, let us list the edges in order of the edge labels, smallest first. This list of edges is shown in Fig. 9.14.

Initially, all thirteen nodes are in components of their own. When we consider edge 1, {Kaneohe, Honolulu}, we merge these two nodes into a single component. The second edge, {Wahiawa, PearlCity}, merges those two cities. The third edge is {PearlCity, Honolulu}. That edge merges the components containing these two cities. Presently, each of these components contains two cities, so we now have one component with four cities, namely

{Wahiawa, PearlCity, Honolulu, Kaneohe}

All other cities are still in components by themselves.

EDGE	CITY 1	CITY 2	DISTANCE
1	Kaneohe	Honolulu	11
2	Wahiawa	PearlCity	12
3	PearlCity	Honolulu	13
4	Wahiawa	Maili	15
5	Kahului	Keokea	16
6	Maili	PearlCity	20
7	Lahaina	Kahului	22
8	Laie	Kaneohe	24
9	Laie	Wahiawa	28
10	Kona	Kamuela	31
11	Kamuela	Hilo	45
12	Kahului	Hana	60
13	Kona	Hilo	114

Fig. 9.14. Edges of Fig. 9.4 in order of labels.

Edge 4 is {Maili, Wahiawa} and adds Maili to the large component. The fifth edge is {Kahului, Keokea}, which merges these two cities into a component. When we consider edge 6, {Maili, PearlCity}, we see a new phenomenon: both ends of the edge are already in the same component. We therefore do no merging with edge 6.

Edge 7 is {Lahaina, Kahului}, and it adds the node Lahaina to the component {Kahului, Keokea}, forming the component {Lahaina, Kahului, Keokea}. Edge 8 adds Laie to the largest component, which is now

{Laie, Kaneohe, Honolulu, PearlCity, Wahiawa, Maili}

The ninth edge, {Laie, Wahiawa}, connects two cities in this component and is thus ignored.

Edge 10 groups Kamuela and Kona into a component, and edge 11 adds Hilo to this component. Edge 12 adds Hana to the component of

{Lahaina, Kahului, Keokea}

Finally, edge 13, {Hilo, Kona}, connects two cities already in the same component. Thus,

{Laie, Kaneohe, Honolulu, PearlCity, Wahiawa, Maili}
{Lahaina, Kahului, Keokea, Hana}
{Kamuela, Hilo, Kona}

is the final set of connected components. ✦

A Data Structure for Forming Components

If we consider the algorithm described informally above, we need to be able to do two things quickly:

1. Given a node, find its current component.

2. Merge two components into one.

There are a number of data structures that can support these operations. We shall study one simple idea that gives surprisingly good performance. The key is to put the nodes of each component into a tree.[2] The component is represented by the root of the tree. The two operations above can now be implemented as follows:

1. To find the component of a node in the graph, we go to the representative of that node in the tree and follow the path in that tree to the root, which represents the component.

2. To merge two different components, we make the root of one component a child of the root of the other.

---

[2] It is important to understand that, in what follows, the "tree" and the "graph" are distinct structures. There is a one-to-one correspondence between the nodes of the graph and the nodes of the tree; that is, each tree node represents a graph node. However, the parent-child edges of the tree are not necessarily edges in the graph.

✦ **Example 9.15.** Let us follow the steps of Example 9.14, showing the trees created at certain steps. Initially, every node is in a one-node tree by itself. The first edge, {Kaneohe, Honolulu}, causes us to merge two one-node trees, {Kaneohe} and {Honolulu}, into one two-node tree, {Kaneohe, Honolulu}. Either node could be made a child of the other. Let us suppose that Honolulu is made the child of the root Kaneohe.

Similarly, the second edge, {Wahiawa, PearlCity}, merges two trees, and we may suppose that PearlCity is made the child of the root Wahiawa. At this point, the current collection of components is represented by the two trees in Fig. 9.15 and nine one-node trees.

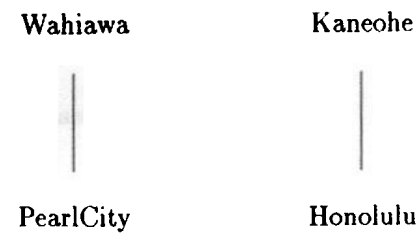


Fig. 9.15. The first two nontrivial trees as we merge components.

The third edge, {PearlCity, Honolulu}, merges these two components. Let us suppose that Wahiawa is made a child of the other root, Kaneohe. Then the resulting component is represented by the tree of Fig. 9.16.

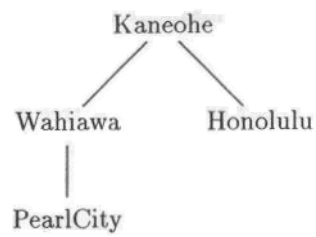


Fig. 9.16. Tree representing component of four nodes.

When we consider the fourth edge, {Wahiawa, Maili}, we merge Maili into the component represented by the tree of Fig. 9.16. We could either make Maili a child of Kaneohe, or make Kaneohe a child of Maili. We prefer the former, since that keeps the height of the tree small, while making the root of the large component a child of the root of the small component tends to make paths in the tree larger. Large paths, in turn, cause us to take more time following a path to the root, which we need to do to determine the component of a node. By following that policy and making arbitrary decisions when components have the same height, we might wind up with the three trees in Fig. 9.17 that represent the three final connected components. ✦

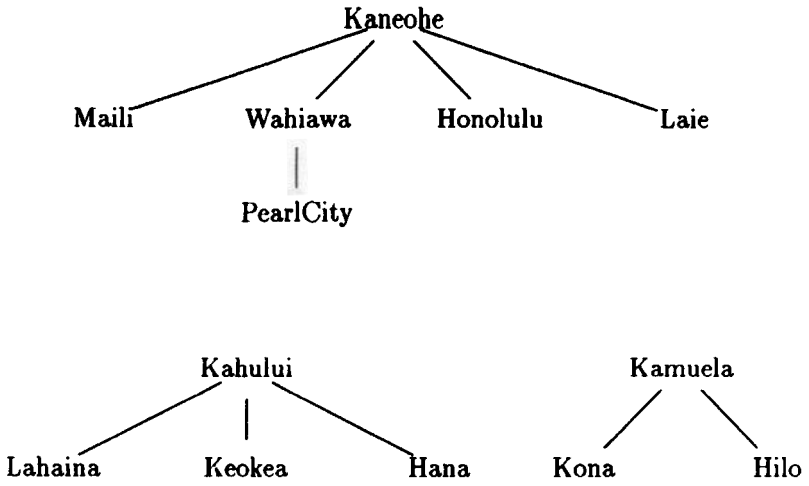


Fig. 9.17. Trees representing final connected components using tree-merging algorithm.

Following the lesson of Example 9.15, we formulate a policy that whenever we merge two trees, the root of lesser height becomes a child of the root with greater height. Ties can be broken arbitrarily. The important gain from this policy is that heights of trees can only grow logarithmically with the number of nodes in the trees, and in practice, the height is often smaller. Therefore, when we follow a path from a tree node to its root, we take at most time proportional to the logarithm of the number of nodes in the tree. We can derive the logarithmic bound by proving the following statement by induction on the height h.

STATEMENT $S(h)$: A tree of height h, formed by the policy of merging lower into higher, has at least 2^h nodes.

BASIS. The basis is $h = 0$. Such a tree must be a single node, and since $2^0 = 1$, the statement $S(0)$ is true.

INDUCTION. Suppose $S(h)$ is true for some $h \geq 0$, and consider a tree T of height $h+1$. At some time during the formation of T by mergers, the height first reached $h+1$. The only way to get a tree of height $h+1$ is to make the root of some tree T_1, of height h, a child of the root of some tree T_2. T is T_1 plus T_2, plus perhaps other nodes that were added later, as suggested by Fig. 9.18.

Now T_1, by the inductive hypothesis, has at least 2^h nodes. Since its root was made a child of the root of T_2, the height of T_2 is also at least h. Thus, T_2 also has at least 2^h nodes. T consists of T_1, T_2, and perhaps more, so T has at least $2^h + 2^h = 2^{h+1}$ nodes. That statement is $S(h+1)$, and we have proved the inductive step.

We now know that if a tree has n nodes and height h, it must be that $n \geq 2^h$. Taking logarithms of both sides, we have $\log_2 n \geq h$; that is, the height of the tree

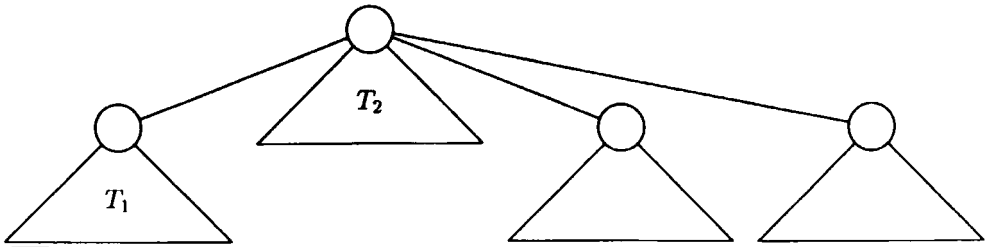


Fig. 9.18. Forming a tree of height $h + 1$.

cannot be greater than the logarithm of the number of nodes. Consequently, when we follow any path from a node to its root, we take $O(\log n)$ time.

We shall now describe in more detail the data structure that implements these ideas. First, suppose that there is a type **NODE** representing nodes. As before, we assume the type **NODE** is **int** and **MAX** is at least the number of nodes in the graph. For our example of Fig. 9.4, we shall let **MAX** be 13.

We shall also assume that there is a list **edges** consisting of cells of type **EDGE**. These cells are defined by

```
typedef struct EDGE *EDGELIST;
struct EDGE {
    NODE node1, node2;
    EDGELIST next;
};
```

Finally, for each node of the graph, we need a corresponding tree node. Tree nodes will be structures of type **TREENODE**, consisting of

1. A parent pointer, so that we can build a tree on the graph's nodes, and follow the tree to its root. A root node will be identified by having **NULL** as its parent.
2. The height of the tree of which a given node is the root. The height will only be used if the node is presently a root.

We may thus define type **TREENODE** by

```
typedef struct TREENODE *TREE;
struct TREENODE {
    int height;
    TREE parent;
};
```

We shall define an array

```
TREE nodes[MAX];
```

to associate with each graph node a node in some tree. It is important to realize that each entry in the array **nodes** is a pointer to a node in the tree, yet this entry is the sole representative of the node in the graph.

Two important auxiliary functions are shown in Fig. 9.19. The first, **find**, takes a node a, gets a pointer to the corresponding tree node, x, and follows the parent pointers in x and its ancestors, until it comes to the root. This search for the root is performed by lines (2) and (3). If the root is found, a pointer to the root is returned at line (4). Note that at line (1), the type **NODE** must be **int** so it may

```
     /* return the root of the tree containing the tree node x
        corresponding to graph node a */
     TREE find(NODE a, TREE nodes[]);
     {
         TREE x;

(1)      x = nodes[a];
(2)      while (x->parent != NULL)
(3)          x = x->parent;
(4)      return x;
     }


     /* merge the trees with roots x and y into one tree,
           by making the root of the lower a child of
           the root of the higher */
     void merge(TREE x, TREE y)
     {
         TREE higher, lower;

(5)      if (x->height > y->height) {
(6)          higher = x;
(7)          lower = y;
         }
         else {
(8)          higher = y;
(9)          lower = x;
         }
(10)     lower->parent = higher;
(11)     if (lower->height == higher->height)
(12)         ++(higher->height);
     }
```

Fig. 9.19. Auxiliary functions `find` and `merge`.

be used to index the array `nodes`.

The second function, `merge`,[3] takes pointers to two tree nodes, x and y, which must be the roots of distinct trees for the merger to work properly. The test of line (5) determines which of the roots has the greater height; ties are broken in favor of y. The higher is assigned to the local variable `higher` and the lower to the local variable `lower` at lines (6–7) or lines (8–9), whichever is appropriate. Then at line (10) the lower is made a child of the higher and at lines (11) and (12) the height of the higher, which is now the root of the combined tree, is incremented by one if the heights of T_1 and T_2 are equal. The height of the lower remains as it was, but it is now meaningless, because the lower is no longer a root.

The heart of the algorithm to find connected components is shown in Fig. 9.20.

[3] Do not confuse this function with a function of the same name used for merge sorting in Chapters 2 and 3.

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 13
typedef int NODE;
typedef struct EDGE *EDGELIST;
struct EDGE {
    NODE node1, node2;
    EDGELIST next;
};

typedef struct TREENODE *TREE
struct TREENODE {
    int height;
    TREE parent;
};

TREE find(NODE a, TREE nodes[]);
void merge(TREE x, TREE y);
EDGELIST makeEdges();

main()
{
    NODE u;
    TREE a, b;
    EDGELIST e;
    TREE nodes[MAX];

    /* initialize nodes so each node is in a tree by itself */
(1) for (u = 0; u < MAX; u++) {
(2)     nodes[u] = (TREE) malloc(sizeof(struct TREENODE));
(3)     nodes[u]->parent = NULL;
(4)     nodes[u]->height = 0;
    }

    /* initialize e as the list of edges of the graph */
(5) e = makeEdges();

    /* examine each edge, and if its ends are in different
       components, then merge them */
(6) while (e != NULL) {
(7)     a = find(e->node1, nodes);
(8)     b = find(e->node2, nodes);
(9)     if (a != b)
(10)        merge(a, b);
(11)    e = e->next;
    }
}
```

Fig. 9.20. C program to find connected components.

Better Algorithms for Connected Components

We shall see, when we learn about depth-first search in Section 9.6, that there is actually a better way to compute connected components, one that takes only $O(m)$ time, instead of $O(m \log n)$ time. However, the data structure given in Section 9.4 is useful in its own right, and we shall see in Section 9.5 another program that uses this data structure.

We assume that the function **makeEdges()** turns the graph at hand into a list of edges. The code for this function is not shown.

Lines (1) through (4) of Fig. 9.20 go down the array **nodes**, and for each node, a tree node is created at line (2). Its **parent** field is set to **NULL** at line (3), making it the root of its own tree, and its **height** field is set to 0 at line (4), reflecting the fact that the node is alone in its tree.

Line (5) then initializes **e** to point to the first edge on the list of edges, and the loop of lines (6) through (11) examines each edge in turn. At lines (7) and (8) we find the roots of the two ends of the current edge. Then at line (9) we test to see if these roots are different tree nodes. If so, the ends of the current edge are in different components, and we merge these components at line (10). If the two ends of the edge are in the same component, we skip line (10), so no change to the collection of trees is made. Finally, line (11) advances us along the list of edges.

Running Time of the Connected Components Algorithm

Let us determine how long the algorithm of Fig. 9.20 takes to process a graph. Suppose the graph has n nodes, and let m be the larger of the number of nodes and the number of edges.[4] First, let us examine the auxiliary functions. We argued that the policy of merging lower trees into higher ones guarantees that the path from any tree node to its root cannot be longer than $\log n$. Thus, **find** takes $O(\log n)$ time.

Next, let us examine the function **merge** from Fig. 9.19. Each of its statements takes $O(1)$ time. Since there are no loops or function calls, the entire function takes $O(1)$ time.

Finally, let us examine the main program of Fig. 9.20. The body of the for-loop of lines (1) to (4) takes $O(1)$ time, and the loop is iterated n times. Thus, the time for lines (1) through (4) is $O(n)$. Let us assume line (5) takes $O(m)$ time. Finally, consider the while-loop of lines (6) to (11). In the body, lines (7) and (8) each take $O(\log n)$ time, since they are calls to a function, **find**, that we just determined takes $O(\log n)$ time. Lines (9) and (11) clearly take $O(1)$ time. Line (10) likewise takes $O(1)$ time, because we just determined that function **merge** takes $O(1)$ time. Thus, the entire body takes $O(\log n)$ time. The while-loop iterates m times, where m is the number of edges. Thus, the time for this loop is $O(m \log n)$, that is, the number of iterations times the bound on the time for the body.

In general, then, the running time of the entire program can be expressed as $O(n + m + m \log n)$. However, m is at least n, and so the $m \log n$ term dominates the other terms. Thus, the running time of the program in Fig. 9.20 is $O(m \log n)$.

[4] It is normal to think of m as the number of edges, but in some graphs, there are more nodes than edges.

CITY 1	CITY 2	DISTANCE
Marquette	Sault Ste. Marie	153
Saginaw	Flint	31
Grand Rapids	Lansing	60
Detroit	Lansing	78
Escanba	Sault Ste. Marie	175
Ann Arbor	Detroit	28
Ann Arbor	Battle Creek	89
Battle Creek	Kalamazoo	21
Menominee	Escanba	56
Kalamazoo	Grand Rapids	45
Escanba	Marquette	78
Battle Creek	Lansing	40
Flint	Detroit	58

Fig. 9.21. Some distances within the state of Michigan.

EXERCISES

9.4.1: Figure 9.21 lists some cities in the state of Michigan and the road mileage between them. For the purposes of this exercise, ignore the mileage. Construct the connected components of the graph by examining each edge in the manner described in this section.

9.4.2*: Prove, by induction on k, that a connected component of k nodes has at least $k - 1$ edges.

9.4.3*: There is a simpler way to implement "merge" and "find," in which we keep an array indexed by nodes, giving the component of each node. Initially, each node is in a component by itself, and we name the component by the node. To find the component of a node, we simply look up the corresponding array entry. To merge components, we run down the array, changing each occurrence of the first component to the second.

a) Write a C program to implement this algorithm.

b) As a function of n, the number of nodes, and m, the larger of the number of nodes and edges, what is the running time of this program?

c) For certain numbers of edges and nodes, this implementation is actually better than the one described in the section. When?

9.4.4*: Suppose that instead of merging lower trees into higher trees in the connected components algorithm of this section, we merge trees with fewer nodes into trees with a larger number of nodes. Is the running time of the connected-components algorithm still $O(m \log n)$?

9.5 Minimal Spanning Trees

There is an important generalization of the connected components problem, in which we are given an undirected graph with edges labeled by numbers (integers or reals). We must not only find the connected components, but for each component we must find a tree connecting the nodes of that component. Moreover, this tree must be *minimal*, meaning that the sum of the edge labels is as small as possible.

Unrooted, unordered trees

The trees talked about here are not quite the same as the trees of Chapter 5. Here, no node is designated the root, and there is no notion of children or of order among the children. Rather, when we speak of "trees" in this section, we mean unrooted, unordered trees, which are just undirected graphs that have no simple cycles.

Spanning tree

A *spanning tree* for an undirected graph G is the nodes of G together with a subset of the edges of G that

1. Connect the nodes; that is, there is a path between any two nodes using only edges in the spanning tree.
2. Form an unrooted, unordered tree; that is, there are no (simple) cycles.

If G is a single connected component, then there is always a spanning tree. A *minimal spanning tree* is a spanning tree the sum of whose edge labels is as small as that of any spanning tree for the given graph.

✦ **Example 9.16.** Let graph G be the connected component for the island of Oahu, as in Fig. 9.4 or Fig. 9.10. One possible spanning tree is shown in Fig. 9.22. It is formed by deleting the edges {Maili, Wahiawa} and {Kaneohe, Laie}, and retaining the other five edges. The *weight*, or sum of edge labels, for this tree is 84. As we shall see, that is not a minimum. ✦

Weight of a tree

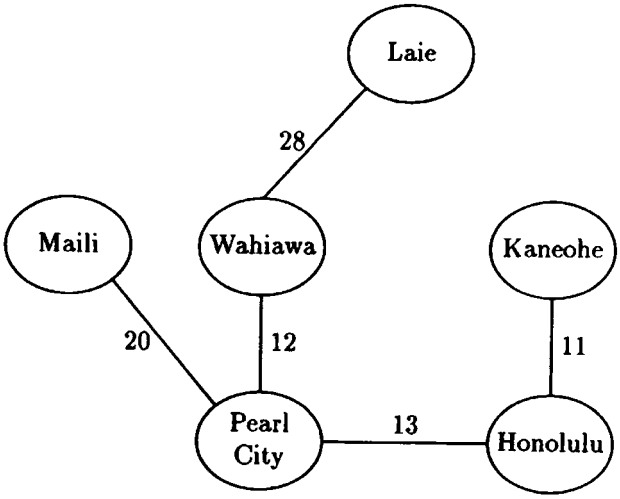
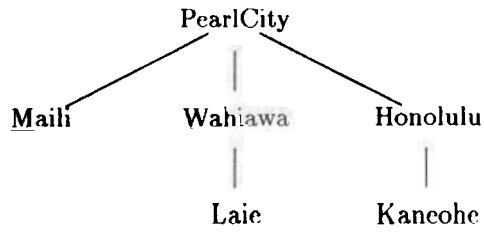


Fig. 9.22. A spanning tree for the island of Oahu.

Rooted and Unrooted Trees

The notion of an unrooted tree should not seem too strange. In fact, we can choose any node of an unrooted tree to be the root. That gives a direction to all edges, away from the root, or from parent to child. Physically, it is as if we picked up the unrooted tree by a node, letting the rest of the tree dangle from the selected node. For example, we could make Pearl City the root of the spanning tree in Fig. 9.22, and it would look like this:



We can order the children of each node if we wish, but the order will be arbitrary, bearing no relation to the original unrooted tree.

Finding a Minimal Spanning Tree

Kruskal's algorithm

There are a number of algorithms to find minimal spanning trees. We shall exhibit one, called *Kruskal's algorithm,* that is a simple extension to the algorithm discussed in the last section for finding connected components. The changes needed are

1. We are required to consider edges in increasing order of their labels. (We happened to choose that order in Example 9.14, but it was not required for connected components.)
2. As we consider edges, if an edge has its ends in different components, then we select that edge for the spanning tree and merge components, as in the algorithm of the previous section. Otherwise, we do not select the edge for the spanning tree, and, of course, we do not merge components.

✦ **Example 9.17.** The Acme Surfboard Wax Company has offices in the thirteen cities shown in Fig. 9.4. It wishes to rent dedicated data transmission lines from the phone company, and we shall suppose that the phone lines run along the roads that are indicated by edges in Fig. 9.4. Between islands, the company must use satellite transmission, and the cost will be proportional to the number of components. However, for the ground transmission lines, the phone company charges by the mile.[5] Thus, we wish to find a minimal spanning tree for each connected component of the graph of Fig. 9.4.

If we divide the edges by component, then we can run Kruskal's algorithm on

[5] This is one possible way to charge for leased telephone lines. One finds a minimal spanning tree connecting the desired sites, and the charge is based on the weight of that tree, regardless of how the phone connections are provided physically.

each component separately. However, if we do not already know the components, then we must consider all the edges together, smallest label first, in the order of Fig. 9.14. As in Section 9.4, we begin with each node in a component by itself.

We first consider the edge {Kaneohe, Honolulu}, the edge with the smallest label. This edge merges these two cities into one component, and because we perform a merge operation, we select that edge for the minimal spanning tree. Edge 2 is {Wahiawa, PearlCity}, and since that edge also merges two components, it is selected for the spanning tree. Likewise, edges 3 and 4, {PearlCity, Honolulu} and {Wahiawa, Maili}, merge components, and are therefore put in the spanning tree.

Edge 5, {Kahului, Keokea}, merges these two cities, and is also accepted for the spanning tree, although this edge will turn out to be part of the spanning tree for the Maui component, rather than the Oahu component as was the case for the four previous edges.

Edge 6, {Maili, PearlCity}, connects two cities that are already in the same component. Thus, this edge is rejected for the spanning tree. Even though we shall have to pick some edges with larger labels, we cannot pick {Maili, PearlCity}, because to do so would form a cycle of the cities Maili, Wahiawa, and Pearl City. We cannot have a cycle in the spanning tree, so one of the three edges must be excluded. As we consider edges in order of label, the last edge of the cycle considered must have the largest label, and is the best choice to exclude.

Edge 7, {Lahaina, Kahului}, and edge 8, {Laie, Kaneohe}, are both accepted for the spanning tree, because they merge components. Edge 9, {Laie, Wahiawa}, is rejected because its ends are in the same component. We accept edges 10 and 11; they form the spanning tree for the "big island" component, and we accept edge 12 to complete the Maui component. Edge 13 is rejected, because it connects Kona and Hilo, which were merged into the same component by edges 10 and 11. The resulting spanning trees of the components are shown in Fig. 9.23. ✦

Why Kruskal's Algorithm Works

We can prove that Kruskal's algorithm produces a spanning tree whose weight is as small as that of any spanning tree for the given graph. Let G be an undirected, connected graph. For convenience, let us add infinitesimal amounts to some labels, if necessary, so that all labels are distinct, and yet the sum of the added infinitesimals is not as great as the difference between two edges of G that have different labels. As a result, G with the new labels will have a unique minimal spanning tree, which will be one of the minimal spanning trees of G with the original weights.

Then, let $e_1, e_2, \ldots, e_m$ be all the edges of G, in order of their labels, smallest first. Note that this order is also the order in which Kruskal's algorithm considers the edges. Let K be the spanning tree for G with the adjusted labels produced by Kruskal's algorithm, and let T be the unique minimal spanning tree for G.

We shall prove that K and T are really the same. If they are different, then there must be at least one edge that is in one but not the other. Let e_i be the first such edge in the ordering of edges; that is, each of $e_1, \ldots, e_{i-1}$ is either in both K and T, or in neither of K and T. There are two cases, depending on whether e_i is in K or is in T. We shall show a contradiction in each case, and thus conclude that e_i does not exist; thus $K = T$, and K is the minimal spanning tree for G.

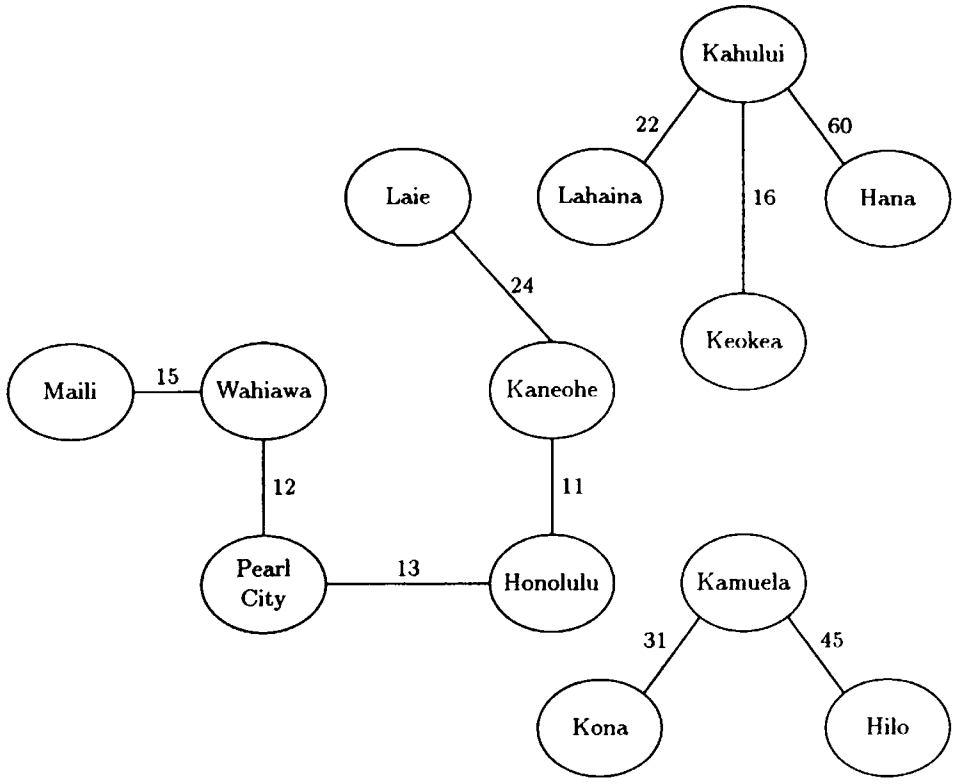


Fig. 9.23. Spanning trees for the graph of Fig. 9.4.

Greed Sometimes Pays

Greedy algorithm

Kruskal's algorithm is a good example of a *greedy algorithm*, in which we make a series of decisions, each doing what seems best at the time. The local decisions are which edge to add to the spanning tree being formed. In each case, we pick the edge with the least label that does not violate the definition of "spanning tree" by completing a cycle. Often, the overall effect of locally optimal decisions is not globally optimum. However, in the case of Kruskal's algorithm, it can be shown that the result is globally optimal; that is, a spanning tree of minimal weight results.

Case 1. Edge e_i is in T but not in K. If Kruskal's algorithm rejects e_i, then e_i must form a cycle with some path P of edges previously selected for K, as suggested in Fig. 9.24. Thus, the edges of P are all found among $e_1, \ldots, e_{i-1}$. However, T and K agree about these edges; that is, if the edges of P are in K, then they are also in T. But since T has e_i as well, P plus e_i form a cycle in T, contradicting our assumption that T was a spanning tree. Thus, it is not possible that e_i is in T but not in K.

Case 2. Edge e_i is in K but not in T. Let e_i connect the nodes u and v. Since T is connected, there must be some acyclic path in T between u and v; call it path Q. Since Q does not use edge e_i, Q plus e_i forms a simple cycle in the graph G.

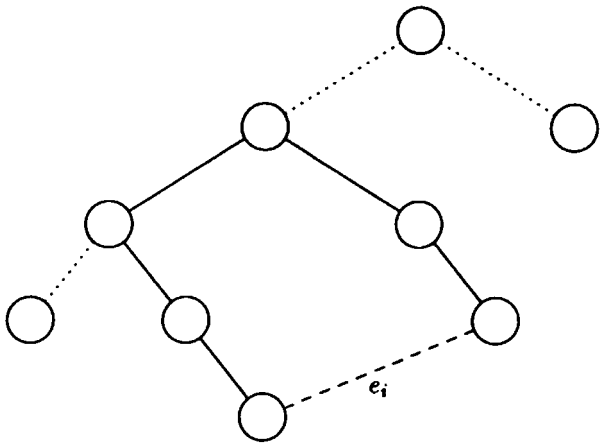


Fig. 9.24. Path P (solid lines) is in T and K; edge e_i is in T only.

There are two subcases, depending on whether or not e_i has a higher label than all the edges on path Q.

a) Edge e_i has the highest label. Then all the edges on Q are among $\{e_1, \ldots, e_{i-1}\}$. Remember that T and K agree on all edges before e_i, and so all the edges of Q are also edges of K. But e_i is also in K, which implies K has a cycle. We thus rule out the possibility that e_i has a higher label than any of the edges of path Q.

b) There is some edge f on path Q that has a higher label than e_i. Suppose f connects nodes w and x. Figure 9.25 shows the situation in tree T. If we remove edge f from T, and add edge e_i, we do not form a cycle, because path Q was broken by the removal of f. The resulting collection of edges has a lower weight than T, because f has a higher label than e_i. We claim the resulting edges still connect all the nodes. To see why, notice that w and x are still connected; there is a path that follows Q from w to u, then follows the edge e_i, then the path Q from v to x. Since $\{w, x\}$ was the only edge removed, if its endpoints are still connected, surely all nodes are connected. Thus, the new set of edges is a spanning tree, and its existence contradicts the assumption that T was minimal.

We have now shown that it is impossible for e_i to be in K but not in T. That rules out the second case. Since it is impossible that e_i is in one of T and K, but not the other, we conclude that K really is the minimal spanning tree T. That is, Kruskal's algorithm always finds a minimal spanning tree.

Running Time of Kruskal's Algorithm

Suppose we run Kruskal's algorithm on a graph of n nodes. As in the previous section, let m be the larger of the number of nodes and the number of edges, but remember that typically the number of edges is the larger. Let us suppose that the graph is represented by adjacency lists, so we can find all the edges in $O(m)$ time.

To begin, we must sort the edges by label, which takes $O(m \log m)$ time, if we use an efficient sorting algorithm such as merge sort. Next, we consider the edges, taking $O(m \log n)$ time to do all the merges and finds, as discussed in the

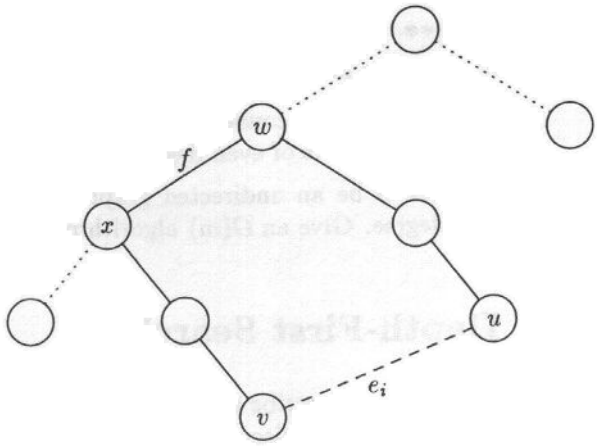


Fig. 9.25. Path Q (solid) is in T.
We can add edge e_i to T and remove the edge f.

previous section. It appears that the total time for Kruskal's algorithm is thus $O(m(\log n + \log m))$.

However, notice that $m \leq n^2$, because there are only $n(n-1)/2$ pairs of nodes. Thus, $\log m \leq 2 \log n$, and $m(\log n + \log m) \leq 3m \log n$. Since constant factors can be neglected within a big-oh expression, we conclude that Kruskal's algorithm takes $O(m \log n)$ time.

EXERCISES

9.5.1: Draw the tree of Fig. 9.22 if Wahiawa is selected as the root.

9.5.2: Use Kruskal's algorithm to find minimal spanning trees for each of the components of the graph whose edges and labels are listed in Fig. 9.21 (see the exercises for Section 9.4).

9.5.3**: Prove that if G is a connected, undirected graph of n nodes, and T is a spanning tree for G, then T has $n-1$ edges. *Hint*: We need to do an induction on n. The hard part is to show that T must have some node v with degree 1; that is, T has exactly one edge containing v. Consider what would happen if for every node u, there were at least two edges of T containing u. By following edges into and out of a sequence of nodes, we would eventually find a cycle. Since T is supposedly a spanning tree, it could not have a cycle, which gives us a contradiction.

9.5.4*: Once we have selected $n-1$ edges, it is not necessary to consider any more edges for possible inclusion in the spanning tree. Describe a variation of Kruskal's algorithm that does not sort all the edges, but puts them in a priority queue, with the negative of the edge's label as its priority (i.e., shortest edge is selected first by *deleteMax*). Show that if a spanning tree can be found among the first $m/\log m$ edges, then this version of Kruskal's algorithm takes only $O(m)$ time.

9.5.5*: Suppose we find a minimal spanning tree T for a graph G. Let us then add to G the edge $\{u, v\}$ with weight w. Under what circumstances will T be a minimal spanning tree of the new graph?

Euler circuit

9.5.6**: An *Euler circuit* for an undirected graph G is a path that starts and ends at the same node and contains each edge of G exactly once.

a) Show that a connected, undirected graph has an Euler circuit if and only if each node is of even degree.

b) Let G be an undirected graph with m edges in which every node is of even degree. Give an $O(m)$ algorithm to construct an Euler circuit for G.

9.6 Depth-First Search

We shall now describe a graph-exploration method that is useful for directed graphs. In Section 5.4 we discussed the preorder and postorder traversals of trees, where we start at the root and recursively explore the children of each node we visit. We can apply almost the same idea to any directed graph.[6] From any node, we recursively explore its successors.

However, we must be careful if the graph has cycles. If there is a cycle, we can wind up calling the exploration function recursively around the cycle forever. For instance, consider the graph of Fig. 9.26. Starting at node a, we might decide to explore node b next. From b we might explore c first, and from c we could explore b first. That gets us into an infinite recursion, where we alternate exploring from b and c. In fact, it doesn't matter in what order we choose to explore successors of b and c. Either we shall get caught in some other cycle, or we eventually explore c from b and explore b from c, infinitely.

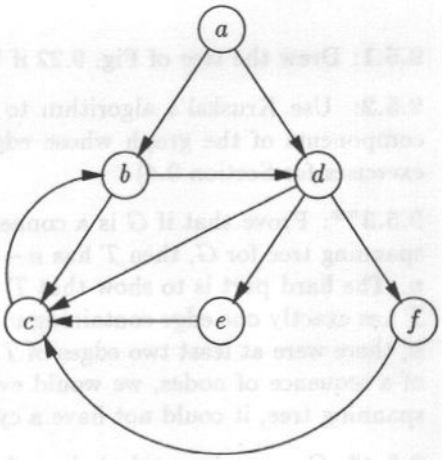


Fig. 9.26. An example directed graph.

There is a simple solution to our problem: We mark nodes as we visit them, and never revisit marked nodes. Then, any node we can reach from our starting node will be reached, but no previously visited node will be revisited. We shall see

[6] Notice that a tree can be thought of as a special case of a directed graph, if we regard the arcs of the tree as directed from parent to child. In fact, a tree is always an acyclic graph as well.

that the time taken by this exploration takes time proportional to the number of arcs explored.

The search algorithm is called *depth-first search* because we find ourselves going as far from the initial node (as "deep") as fast as we can. It can be implemented with a simple data structure. Again, let us assume that the type `NODE` is used to name nodes and that this type is `int`. We represent arcs by adjacency lists. Since we need a "mark" for each node, which can take on the values `VISITED` and `UNVISITED`, we shall create an array of structures to represent the graph. These structures will contain both the mark and the header for the adjacency list.

```
enum MARKTYPE {VISITED, UNVISITED};
typedef struct {
    enum MARKTYPE mark;
    LIST successors;
} GRAPH[MAX];
```

where `LIST` is an adjacency list, defined in the customary manner:

```
typedef struct CELL *LIST;
struct CELL {
    NODE nodeName
    LIST next;
};
```

We begin by marking all the nodes `UNVISITED`. Recursive function `dfs(u, G)` of Fig. 9.27 works on a node u of some externally defined graph G of type `GRAPH`.

At line (1) we mark u `VISITED`, so we don't call `dfs` on it again. Line (2) initializes p to point to the first cell on the adjacency list for node u. The loop of lines (3) through (7) takes p down the adjacency list, considering each successor, v, of u, in turn.

```
    void dfs(NODE u, GRAPH G)
    {
        LIST p; /* runs down the adjacency list of u */
        NODE v; /* the node in the cell pointed to by p */

(1)     G[u].mark = VISITED;
(2)     p = G[u].successors;
(3)     while (p != NULL) {
(4)         v = p->nodeName;
(5)         if (G[v].mark == UNVISITED)
(6)             dfs(v, G);
(7)         p = p->next;
        }
    }
```

Fig. 9.27. The recursive depth-first search function.

Line (4) sets v to be the "current" successor of u. At line (5) we test whether v has ever been visited before. If so, we skip the recursive call at line (6) and we move p to the next cell of the adjacency list at line (7). However, if v has never been

visited, we start a depth-first search from node v, at line (6). Eventually, we finish the call to `dfs(v, G)`. Then, we execute line (7) to move p down u's adjacency list and go around the loop.

✦ **Example 9.18.** Suppose G is the graph of Fig. 9.26, and, for specificity, assume the nodes on each adjacency list are ordered alphabetically. Initially, all nodes are marked `UNVISITED`. Let us call `dfs(a)`.[7] Node a is marked `VISITED` at line (1), and at line (2) we initialize p to point to the first cell on a's adjacency list. At line (4) v is set to b, since b is the node in the first cell. Since b is currently unvisited, the test of line (5) succeeds, and at line (6) we call `dfs(b)`.

Now, we start a new call to `dfs`, with $u = b$, while the old call with $u = a$ is dormant but still alive. We begin at line (1), marking b `VISITED`. Since c is the first node on b's adjacency list, c becomes the value of v at line (4). Node c is unvisited, so that we succeed at line (5) and at line (6) we call `dfs(c)`.

A third call to `dfs` is now alive, and to begin `dfs(c)`, we mark c `VISITED` and set v to b at line (4), since b is the first, and only, node on c's adjacency list. However, b was already marked `VISITED` at line (1) of the call to `dfs(b)`, so that we skip line (6) and move p down c's adjacency list at line (7). Since c has no more successors, p becomes `NULL`, so that the test of line (3) fails, and `dfs(c)` is finished.

We now return to the call `dfs(b)`. Pointer p is advanced at line (7), and it now points to the second cell of b's adjacency list, which holds node d. We set v to d at line (4), and since d is unvisited, we call `dfs(d)` at line (6).

For the execution of `dfs(d)`, we mark d `VISITED`. Then v is first set to c. But c is visited, and so next time around the loop, $v = e$. That leads to the call `dfs(e)`. Node e has only c as a successor, and so after marking e `VISITED`, `dfs(e)` returns to `dfs(d)`. We next set $v = f$ at line (4) of `dfs(d)`, and call `dfs(f)`. After marking f `VISITED`, we find that f also has only c as a successor, and c is visited.

We are now finished with `dfs(f)`. Since f is the last successor of d, we are also finished with `dfs(d)`, and since d is the last successor of b, we are done with `dfs(b)` as well. That takes us back to `dfs(a)`. Node a has another successor, d, but that node is visited, and so we are done with `dfs(a)` as well.

Figure 9.28 summarizes the action of `dfs` on the graph of Fig. 9.26. We show the stack of calls to `dfs`, with the currently active call at the right. We also indicate the action taken at each step, and we show the value of the local variable v associated with each currently live call, or show that `p = NULL`, indicating that there is no active value for v. ✦

Constructing a Depth-First Search Tree

Because we mark nodes to avoid visiting them twice, the graph behaves like a tree as we explore it. In fact, we can draw a tree whose parent-child edges are some of the arcs of the graph G being searched. If we are in `dfs(u)`, and a call to `dfs(v)` results, then we make v a child of u in the tree. The children of u appear, from left to right, in the order in which *dfs* was called on these children. The node upon which the initial call to *dfs* was made is the root. No node can have *dfs* called on it twice, since it is marked `VISITED` at the first call. Thus, the structure defined is truly a tree. We call the tree a *depth-first search tree* for the given graph.

[7] In what follows, we shall omit the second argument of `dfs`, which is always the graph G.

dfs(a) $v = b$				Call **dfs(b)**
dfs(a) $v = b$	**dfs(b)** $v = c$			Call **dfs(c)**
dfs(a) $v = b$	**dfs(b)** $v = c$	**dfs(c)** $v = b$		Skip; b already visited
dfs(a) $v = b$	**dfs(b)** $v = c$	**dfs(c)** $p =$**NULL**		Return
dfs(a) $v = b$	**dfs(b)** $v = d$			Call **dfs(d)**
dfs(a) $v = b$	**dfs(b)** $v = d$	**dfs(d)** $v = c$		Skip; c already visited
dfs(a) $v = b$	**dfs(b)** $v = d$	**dfs(d)** $v = e$		Call **dfs(e)**
dfs(a) $v = b$	**dfs(b)** $v = d$	**dfs(d)** $v = c$	**dfs(e)** $v = c$	Skip; c already visited
dfs(a) $v = b$	**dfs(b)** $v = d$	**dfs(d)** $v = e$	**dfs(e)** $p =$**NULL**	Return
dfs(a) $v = b$	**dfs(b)** $v = d$	**dfs(d)** $v = f$		Call **dfs(f)**
dfs(a) $v = b$	**dfs(b)** $v = d$	**dfs(d)** $v = f$	**dfs(f)** $v = c$	Skip; c already visited
dfs(a) $v = b$	**dfs(b)** $v = d$	**dfs(d)** $v = f$	**dfs(f)** $p =$**NULL**	Return
dfs(a) $v = b$	**dfs(b)** $v = d$	**dfs(d)** $p =$**NULL**		Return
dfs(a) $v = b$	**dfs(b)** $p =$**NULL**			Return
dfs(a) $v = d$				Skip; d already visited
dfs(a) $p =$**NULL**				Return

Fig. 9.28. Trace of calls made during depth-first search.

✦ **Example 9.19.** The tree for the exploration of the graph in Fig. 9.26 that was summarized in Fig. 9.28 is seen in Fig. 9.29. We show the *tree arcs*, representing the parent-child relationship, as solid lines. Other arcs of the graph are shown as dotted arrows. For the moment, we should ignore the numbers labeling the nodes. ✦

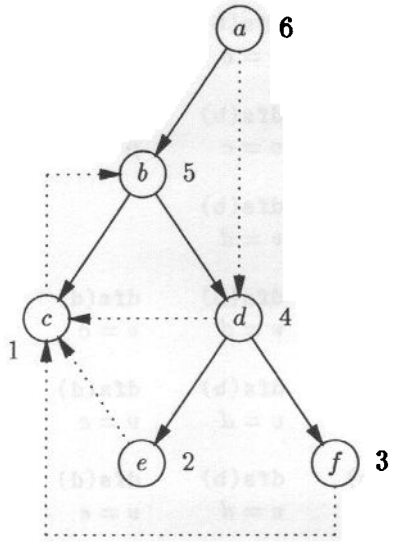


Fig. 9.29. One possible depth-first search tree for the graph of Fig. 9.26.

Classification of Arcs for a Depth-First Search Tree

When we build a depth-first search tree for a graph G, we can classify the arcs of G into four groups. It should be understood that this classification is with respect to a particular depth-first search tree, or equivalently, with respect to the particular order for the nodes in each adjacency list that led to a particular exploration of G. The four kinds of arcs are

1. *Tree arcs*, which are the arcs $u \to v$ such that `dfs(v)` is called by `dfs(u)`.
2. *Forward arcs*, which are arcs $u \to v$ such that v is a proper descendant of u, but not a child of u. For instance, in Fig. 9.29, the arc $a \to d$ is the only forward arc. No tree arc is a forward arc.
3. *Backward arcs*, which are arcs $u \to v$ such that v is an ancestor of u in the tree ($u = v$ is permitted). Arc $c \to b$ is the only example of a backward arc in Fig. 9.29. Any loop, an arc from a node to itself, is classified as backward.
4. *Cross arcs*, which are arcs $u \to v$ such that v is neither an ancestor nor descendant of u. There are three cross arcs in Fig. 9.29: $d \to c$, $e \to c$, and $f \to c$.

Cross arcs go from right to left

In Fig. 9.29, each of the cross arcs go from right to left. It is no coincidence that they do so. Suppose we had in some depth-first search tree a cross arc $u \to v$ such that u was to the left of v. Consider what happens during the call to `dfs(u)`. By the time we finish `dfs(u)`, we shall have considered the arc from u to v. If v has not yet been placed in the tree, then it becomes a child of u in the tree. Since

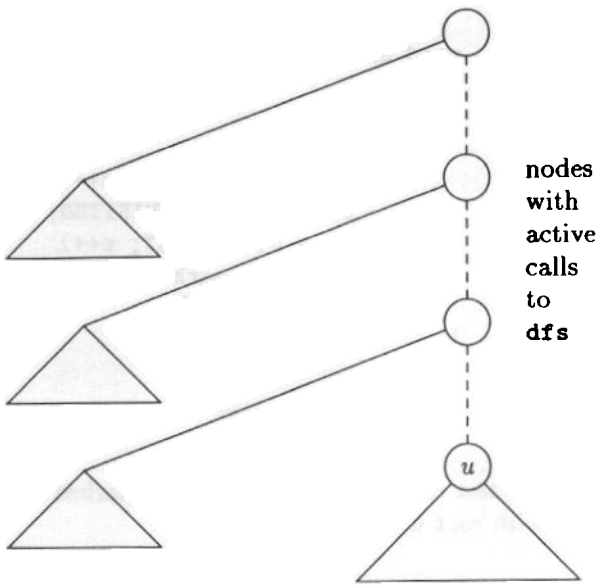


Fig. 9.30. Part of the tree that is built when arc $u \rightarrow v$ is considered.

that evidently did not happen (for then v would not be to the right of u), it must be that v is already in the tree when the arc $u \rightarrow v$ is considered.

However, Fig. 9.30 shows the parts of the tree that exist while `dfs(u)` is active. Since children are added in left-to-right order, no proper ancestor of node u as yet has a child to the right of u. Thus, v can only be an ancestor of u, a descendant of u, or somewhere to the left of u. Thus, if $u \rightarrow v$ is a cross edge, v must be to the left of u, not the right of u as we initially supposed.

The Depth-First Search Forest

We were quite fortunate in Example 9.19 that when we started at node a, we were able to reach all the nodes of the graph of Fig. 9.26. Had we started at any other node, we would not have reached a, and a would not appear in the tree. Thus, the general method of exploring a graph is to construct a sequence of trees. We start at some node u and call `dfs(u)`. If there are nodes not yet visited, we pick one, say v, and call `dfs(v)`. We repeat this process as long as there are nodes not yet assigned to any tree.

When all nodes have been assigned a tree, we list the trees, from left to right, in the order of their construction. This list of trees is called the *depth-first search forest*. In terms of the data types `NODE` and `GRAPH` defined earlier, we can explore an entire externally defined graph G, starting the search on as many roots as necessary by the function of Fig. 9.31. There, we assume that the type `NODE` is `int`, and MAX is the number of nodes in G.

In lines (1) and (2) we initialize all nodes to be `UNVISITED`. Then, in the loop of lines (3) to (5), we consider each node u in turn. When we consider u, if that node has not yet been added to any tree, it will still be marked unvisited when we make the test of line (4). In that case, we call `dfs(u, G)` at line (5) and explore the depth-first search tree with root u. In particular, the first node always becomes the root of a tree. However, if u has already been added to a tree when we perform

```
    void dfsForest(GRAPH G);
    {
        NODE u;

(1)     for (u = 0; u < MAX; u++)
(2)         G[u].mark = UNVISITED;
(3)     for (u = 0; u < MAX; u++)
(4)         if (G[u].mark == UNVISITED)
(5)             dfs(u, G);
    }
```

Fig. 9.31. Exploring a graph by exploring as many trees as necessary.

the test of line (4), then u will be marked `VISITED`, and so we do not create a tree with root u.

✦ **Example 9.20.** Suppose we apply the above algorithm to the graph of Fig. 9.26, but let d be the node whose name is 0; that is, d is the first root of a tree for the depth-first spanning forest. We call `dfs(d)`, which constructs the first tree of Fig. 9.32. Now, all nodes but a are visited. As u becomes each of the various nodes in the loop of lines (3) to (5) of Fig. 9.31, the test of line (4) fails except when $u = a$. Then, we create the one-node second tree of Fig. 9.32. Note that both successors of a are marked `VISITED` when we call `dfs(a)`, and so we do not make any recursive calls from `dfs(a)`. ✦

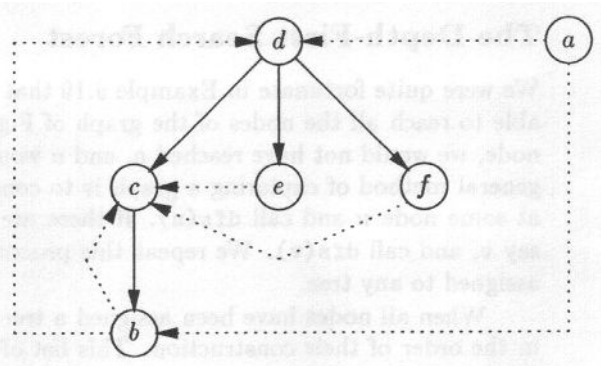


Fig. 9.32. A depth-first search forest.

When we present the nodes of a graph as a depth-first search forest, the notions of forward, backward, and tree arcs apply as before. However, the notion of a cross arc must be extended to include arcs that run from one tree to a tree to its left. Examples of such cross arcs are $a \to b$ and $a \to d$ in Fig. 9.32.

The rule that cross arcs always go from right to left continues to hold. The reason is also the same. If there were a cross arc $u \to v$ that went from one tree to a tree to the right, then consider what happens when we call `dfs(u)`. Since v

The Perfection of Depth-First Search

Regardless of the relationship between the numbers of nodes and arcs, the running time of the depth-first exploration of a graph takes time proportional to the "size" of the graph, that is, the sum of the numbers of nodes and arcs. Thus, depth-first search is, to within a constant factor, as fast as any algorithm that "looks at" the graph.

was not added to the tree being formed at the moment, it must already have been in some tree. But the trees to the right of u have not yet been created, and so v cannot be part of one of them.

Running Time of the Depth-First Search Algorithm

Let G be a graph with n nodes and let m be the larger of the number of nodes and the number of arcs. Then `dfsForest` of Fig. 9.31 takes $O(m)$ time. The proof of this fact requires a trick. When calculating the time taken by a call `dfs(u)`, we shall not count the time taken by any recursive calls to `dfs` at line (6) in Fig. 9.27, as Section 3.9 suggested we should. Rather, observe that we call `dfs(u)` once for each value of u. Thus, if we sum the cost of each call, exclusive of its recursive calls, we get the total time spent in all the calls as a group.

Notice that the while-loop of lines (3) to (7) in Fig. 9.27 can take a variable amount of time, even excluding the time spent in recursive calls to `dfs`, because the number of successors of node u could be any number from 0 to n. Suppose we let m_u be the out-degree of node u, that is, the number of successors of u. Then the number of times around the while-loop during the execution of `dfs(u)` is surely m_u. We do not count the execution of `dfs(v, G)` at line (6) when assessing the running time of `dfs(u)`, and the body of the loop, exclusive of this call, takes $O(1)$ time. Thus, the total time spent in the loop of lines (3) to (7), exclusive of time spent in recursive calls is $O(1 + m_u)$; the additional 1 is needed because m_u might be 0, in which case we still take $O(1)$ time for the test of (3). Since lines (1) and (2) of `dfs` take $O(1)$ time, we conclude that, neglecting recursive calls, `dfs(u)` takes time $O(1 + m_u)$ to complete.

Now we observe that during the running of `dfsForest`, we call `dfs(u)` exactly once for each value of u. Thus, the total time spent in all these calls is big-oh of the sum of the times spent in each, that is, $O(\sum_u (1 + m_u))$. But $\sum_u m_u$ is just the number of arcs in the graph, that is, at most m,[8] since each arc emanates from some one node. The number of nodes is n, so that $\sum_u 1$ is just n. As $n \leq m$, the time taken by all the calls to `dfs` is thus $O(m)$.

Finally, we must consider the time taken by `dfsForest`. This program, in Fig. 9.31, consists of two loops, each iterated n times. The bodies of the loops are easily seen to take $O(1)$ time, exclusive of the calls to `dfs`, and so the cost of the loops is $O(n)$. This time is dominated by the $O(m)$ time of the calls to `dfs`. Since the time for the `dfs` calls is already accounted for, we conclude that `dfsForest`, together with all its calls to `dfs`, takes $O(m)$ time.

[8] In fact, the sum of the m_u's will be exactly m, except in the case that the number of nodes exceeds the number of arcs; recall that m is the larger of the numbers of nodes and arcs.

Postorder Traversals of Directed Graphs

Once we have a depth-first search tree, we could number its nodes in postorder. However, there is an easy way to do the numbering during the search itself. We simply attach the number to a node u as the last thing we do before `dfs(u)` completes. Then, a node is numbered right after all its children are numbered, just as in a postorder numbering.

```
    int k; /* counts visited nodes */

    void dfs(NODE u, GRAPH G)
    {
        LIST p; /* points to cells of adjacency list of u */
        NODE v; /* the node in the cell pointed to by p */
(1)     G[u].mark = VISITED;
(2)     p = G[u].successors;
(3)     while (p != NULL) {
(4)         v = p->nodeName;
(5)         if (G[v].mark == UNVISITED)
(6)             dfs(v, G);
(7)         p = p->next;
        }
(8)     ++k;
(9)     G[u].postorder = k;
    }

    void dfsForest(GRAPH G)
    {
        NODE u;
(10)    k = 0;
(11)    for (u = 0; u < MAX; u++)
(12)        G[u].mark = UNVISITED;
(13)    for (u = 0; u < MAX; u++)
(14)        if (G[u].mark == UNVISITED)
(15)            dfs(u, G);
    }
```

Fig. 9.33. Procedure to number the nodes of a directed graph in postorder.

✦ **Example 9.21.** The tree of Fig. 9.29, which we constructed by depth-first search of the graph in Fig. 9.26, has the postorder numbers labeling the nodes. If we examine the trace of Fig. 9.28, we see that the first call to return is `dfs(c)`, and node c is given the number 1. Then, we visit d, then e, and return from the call to e. Therefore, e's number is 2. Similarly, we visit and return from f, which is numbered 3. At that point, we have completed the call on d, which gets number 4. That completes the call to `dfs(b)`, and the number of b is 5. Finally, the original

call to a returns, giving a the number 6. Notice that this order is exactly the one we would get if we simply walked the tree in postorder. ✦

We can assign the postorder numbers to the nodes with a few simple modifications to the depth-first search algorithm we have written so far; these changes are summarized in Fig. 9.33.

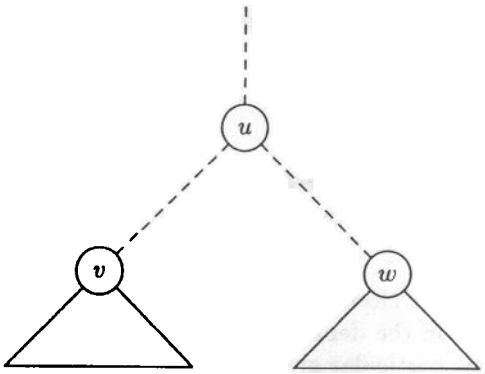
1. In the `GRAPH` type, we need an additional field for each node, called `postorder`. For the graph G, we place the postorder number of node u in `G[u].postorder`. This assignment is accomplished at line (9) of Fig. 9.33.

2. We use a global variable `k` to count nodes in postorder. This variable is defined externally to `dfs` and `dfsForest`. As seen in Fig. 9.33, we initialize k to 0 in line (10) of `dfsForest`, and just before assigning a postorder number, we increment k by 1 at line (8) in `dfs`.

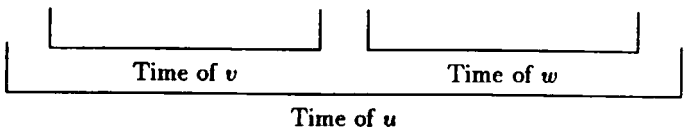
Notice that as a result, when there is more than one tree in the depth-first search forest, the first tree gets the lowest numbers, the next tree gets the next numbers in order, and so on. For example, in Fig. 9.32, a would get the postorder number 6.

Special Properties of Postorder Numbers

The impossibility of cross arcs that go left to right tells us something interesting and useful about the postorder numbers and the four types of arcs in a depth-first presentation of a graph. In Fig. 9.34(a) we see three nodes, u, v, and w, in a depth-first presentation of a graph. Nodes v and w are descendants of u, and w is to the right of v. Figure 9.34(b) shows the duration of activity for the calls to `dfs` for each of these nodes.



(a) Three nodes in a depth-first tree.



(b) Active intervals for their calls to `dfs`.

Fig. 9.34. Relationship between position in tree and duration of calls.

We can make several observations. First, the call to `dfs` on a descendant like v is active for only a subinterval of the time during which the call on an ancestor, like u, is active. In particular, the call to `dfs(v)` terminates before the call to `dfs(u)` does. Thus, the postorder number of v must be less than the postorder number of u whenever v is a proper descendant of u.

Second, if w is to the right of v, then the call to `dfs(w)` cannot begin until after the call to `dfs(v)` terminates. Thus, whenever v is to the left of w, the postorder number of v is less than that of w. Although not shown in Fig. 9.34, the same is true even if v and w are in different trees of the depth-first search forest, with v's tree to the left of w's tree.

We can now consider the relationship between the postorder numbers of u and v for each arc $u \to v$.

1. If $u \to v$ is a tree arc or forward arc, then v is a descendant of u, and so v precedes u in postorder.

2. If $u \to v$ is a cross arc, then we know v is to the left of u, and again v precedes u in postorder.

3. If $u \to v$ is a backward arc and $v \neq u$, then v is a proper ancestor of u, and so v follows u in postorder. However, $v = u$ is possible for a backward arc, since a loop is a backward arc. Thus, in general, for backward arc $u \to v$, we know that the postorder number of v is at least as high as the postorder number of u.

In summary, we see that in postorder, the head of an arc precedes the tail, unless the arc is a backward arc; in which case the tail precedes or equals the head. Thus, we can identify the backward arcs simply by finding those arcs whose tails are equal to or less than their heads in postorder. We shall see a number of applications of this idea in the next section.

EXERCISES

9.6.1: For the tree of Fig. 9.5 (see the exercises for Section 9.2), give two depth-first search trees starting with node a. Give a depth-first search tree starting with node d.

9.6.2*: No matter which node we start with in Fig. 9.5, we wind up with only one tree in the depth-first search forest. Explain briefly why that must be the case for this particular graph.

9.6.3: For each of your trees of Exercise 9.6.1, indicate which of the arcs are tree, forward, backward, and cross arcs.

9.6.4: For each of your trees of Exercise 9.6.1, give the postorder numbers for the nodes.

9.6.5*: Consider the graph with three nodes, a, b, and c, and the two arcs $a \to b$ and $b \to c$. Give all the possible depth-first search forests for this graph, considering all possible starting nodes for each tree. What is the postorder numbering of the nodes for each forest? Are the postorder numbers always the same for this graph?

9.6.6*: Consider the generalization of the graph of Exercise 9.6.5 to a graph with n nodes, $a_1, a_2, \ldots, a_n$, and arcs $a_1 \rightarrow a_2$, $a_2 \rightarrow a_3, \ldots, a_{n-1} \rightarrow a_n$. Prove by complete induction on n that this graph has 2^{n-1} different depth-first search forests. *Hint*: It helps to remember that $1 + 1 + 2 + 4 + 8 + \cdots + 2^i = 2^{i+1}$, for $i \geq 0$.

9.6.7*: Suppose we start with a graph G and add a new node x that is a predecessor of all other nodes in G. If we run `dfsForest` of Fig. 9.31 on the new graph, starting at node x, then a single tree results. If we then delete x from this tree, several trees may result. How do these trees relate to the depth-first search forest of the original graph G?

9.6.8**: Suppose we have a directed graph G, from whose representation we have just constructed a depth-first spanning forest F by the algorithm of Fig. 9.31. Let us now add the arc $u \rightarrow v$ to G to form a new graph H, whose representation is exactly that of G, except that node v now appears somewhere on the adjacency list for node u. If we now run Fig. 9.31 on this representation of H, under what circumstances will the same depth-first forest F be constructed? That is, when will the tree arcs for H are exactly the same as the tree arcs for G?

9.7 Some Uses of Depth-First Search

In this section, we see how depth-first search can be used to solve some problems quickly. As previously, we shall here use n to represent the number of nodes of a graph, and we shall use m for the larger of the number of nodes and the number of arcs; in particular, we assume that $n \leq m$ is always true. Each of the algorithms presented takes $O(m)$ time, on a graph represented by adjacency lists. The first algorithm determines whether a directed graph is acyclic. Then for those graphs that are acyclic, we see how to find a topological sort of the nodes (topological sorting was discussed in Section 7.10; we shall review the definitions at the appropriate time). We also show how to compute the transitive closure of a graph (see Section 7.10 again), and how to find connected components of an undirected graph faster than the algorithm given in Section 9.4.

Finding Cycles in a Directed Graph

During a depth-first search of a directed graph G, we can assign a postorder number to all the nodes in $O(m)$ time. Recall from the last section that we discovered the only arcs whose tails are equal to or less than their heads in postorder are the backward arcs. Whenever there is a backward arc, $u \rightarrow v$, in which the postorder number of v is at least as large as the postorder number of u, there must be a cycle in the graph, as suggested by Fig. 9.35. The cycle consists of the arc from u to v, and the path in the tree from v to its descendant u.

The converse is also true; that is, if there is a cycle, then there must be a backward arc. To see why, suppose there is a cycle, say $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k \rightarrow v_1$, and let the postorder number of node v_i be p_i, for $i = 1, 2, \ldots, k$. If $k = 1$, that is, the cycle is a single arc, then surely $v_1 \rightarrow v_1$ is a backward arc in any depth-first presentation of G.

If $k > 1$, suppose that none of the arcs $v_1 \rightarrow v_2$, $v_2 \rightarrow v_3$, and so on, up to $v_{k-1} \rightarrow v_k$ are backward. Then each head precedes each tail in postorder, and so the postorder numbers $p_1, p_2, \ldots, p_k$ form a decreasing sequence. In particular,

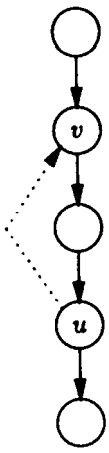


Fig. 9.35. Every backward arc forms a cycle with tree arcs.

$p_k < p_1$. Then consider the arc $v_k \rightarrow v_1$ that completes the cycle. The postorder number of its tail, which is p_k, is less than the postorder number of its head, p_1, and so this arc is a backward arc. That proves there must be some backward arc in any cycle.

As a result, after computing the postorder numbers of all nodes, we simply examine all the arcs, to see if any has a tail less than or equal to its head, in postorder. If so, we have found a backward arc, and the graph is cyclic. If there is no such arc, the graph is acyclic. Figure 9.36 shows a function that tests whether an externally defined graph G is acyclic, using the data structure for graphs described in the previous section. It also makes use of the function **dfsForest** defined in Fig. 9.33 to compute the postorder numbers of the nodes of G.

```
    BOOLEAN testAcyclic(GRAPH G)
    {
        NODE u, v; /* u runs through all the nodes */
        LIST p; /* p points to each cell on the adjacency list
                   for u; v is a node on the adjacency list */

(1)     dfsForest(G);
(2)     for (u = 0; u < MAX; u++) {
(3)         p = G[u].successors;
(4)         while (p != NULL) {
(5)             v = p->nodeName;
(6)             if (G[u].postorder <= G[v].postorder)
(7)                 return FALSE;
(8)             p = p->next;
            }
        }
(9)     return TRUE;
    }
```

Fig. 9.36. Function to determine whether a graph G is acyclic.

After calling **dfsForest** to compute postorder numbers at line (1), we examine each node u in the loop of lines (2) through (8). Pointer p goes down the adjacency list for u, and at line (5), v in turn becomes each successor of u. If at line (6) we find that u equals or precedes v in postorder, then we have found a backward arc $u \rightarrow v$, and we return **FALSE** at line (7). If we find no such backward arc, we return **TRUE** at line (9).

Running Time of the Acyclicity Test

As before, let n be the number of nodes of graph G and let m be the larger of the number of nodes and the number of arcs. We already know that the call to **dfsForest** at line (1) of Fig. 9.36 takes $O(m)$ time. Lines (5) to (8), the body of the while-loop, evidently take $O(1)$ time. To get a good bound on the time for the while-loop itself, we must use the trick that was used in the previous section to bound the time of depth-first search. Let m_u be the out-degree of node u. Then we go around the loop of lines (4) to (8) m_u times. Thus, the time spent in lines (4) to (8) is $O(1 + m_u)$.

Line (3) only takes $O(1)$ time, and so the time spent in the for-loop of lines (2) to (8) is $O(\sum_u (1 + m_u))$. As observed in the previous section, the sum of 1 is $O(n)$, and the sum of m_u is m. Since $n \leq m$, the time for the loop of lines (2) to (8) is $O(m)$. That is the same as the time for line (1), and line (9) takes $O(1)$ time. Thus, the entire acyclicity test takes $O(m)$ time. As for depth-first search itself, the time to detect cycles is, to within a constant factor, just the time it takes to look at the entire graph.

Topological Sorting

Suppose we know that a directed graph G is acyclic. As for any graph, we may find a depth-first search forest for G and thereby determine a postorder for the nodes of G. Suppose $(v_1, v_2, \ldots, v_n)$ is a list of the nodes of G in the reverse of postorder; that is, v_1 is the node numbered n in postorder, v_2 is numbered $n-1$, and in general, v_i is the node numbered $n - i + 1$ in postorder.

The order of the nodes on this list has the property that all arcs of G go forward in the order. To see why, suppose $v_i \rightarrow v_j$ is an arc of G. Since G is acyclic, there are no backward arcs. Thus, for every arc, the head precedes the tail. That is, v_j precedes v_i in postorder. But the list is the reverse of postorder, and so v_i precedes v_j on the list. That is, every tail precedes the corresponding head in the list order.

Topological order

An order for the nodes of a graph G with the property that for every arc of G the tail precedes the head is called a *topological order*, and the process of finding such an order for the nodes is called *topological sorting*. Only acyclic graphs have a topological order, and as we have just seen, we can produce a topological order for an acyclic graph $O(m)$ time, where m is the larger of the number of nodes and arcs, by performing a depth-first search. As we are about to give a node its postorder number, that is, as we complete the call to **dfs** on that node, we push the node onto a stack. When we are done, the stack is a list in which the nodes appear in postorder, with the highest at the top (front). That is the reverse postorder we desire. Since the depth-first search takes $O(m)$ time, and pushing the nodes onto a stack takes only $O(n)$ time, the whole process takes $O(m)$ time.

Applications of Topological Order and Cycle Finding

There are a number of situations in which the algorithms discussed in this section will prove useful. Topological ordering comes in handy when there are constraints on the order in which we do certain tasks, which we represent by nodes. If we draw an arc from u to v whenever we must do task u before v, then a topological order is an order in which we can perform all the tasks. An example in Section 7.10 about putting on shoes and socks illustrated this type of problem.

A similar example is the calling graph of a nonrecursive collection of functions, in which we wish to analyze each function after we have analyzed the functions it calls. As the arcs go from caller to called function, the reverse of a topological order, that is, the postorder itself, is an order in which we can analyze the function, making sure that we only work on a function after we have worked on all the functions it calls.

In other situations, it is sufficient to run the cycle test. For example, a cycle in the graph of task priorities tells us there is no order in which all the tasks can be done, and a cycle in a calling graph tells us there is recursion.

✦ **Example 9.22.** In Fig. 9.37(a) is an acyclic graph, and in Fig. 9.37(b) is the depth-first search forest we get by considering the nodes in alphabetic order. We also show in Fig. 9.37(b) the postorder numbers that we get from this depth-first search. If we list the nodes highest postorder number first, we get the topological order (d, e, c, f, b, a). The reader should check that each of the eight arcs in Fig. 9.37(a) has a tail that precedes its head according to this list. There are, incidentally, three other topological orders for this graph, such as (d, c, e, b, f, a). ✦

The Reachability Problem

Reachable set

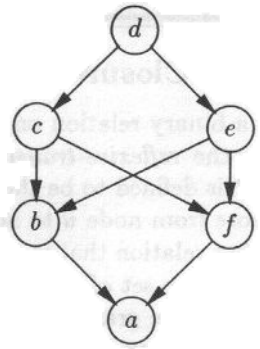
A natural question to ask about a directed graph is, given a node u, which nodes can we reach from u by following arcs? We call this set of nodes the *reachable set* for node u. In fact, if we ask this *reachability* question for each node u, then we know for which pairs of nodes (u, v) there is a path from u to v.

The algorithm for solving reachability is simple. If we are interested in node u, we mark all nodes `UNVISITED` and call `dfs(u)`. We then examine all the nodes again. Those marked `VISITED` are reachable from u, and the others are not. If we then wish to find the nodes reachable from another node v, we set all the nodes to `UNVISITED` again and call `dfs(v)`. This process may be repeated for as many nodes as we like.

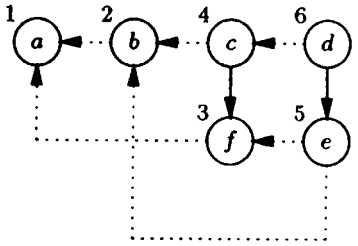
✦ **Example 9.23.** Consider the graph of Fig. 9.37(a). If we start our depth-first search from node a, we can go nowhere, since there are no arcs out of a. Thus, `dfs(a)` terminates immediately. Since only a is visited, we conclude that a is the only node reachable from a.

If we start with b, we can reach a, but that is all; the reachable set for b is $\{a, b\}$. Similarly, from c we reach $\{a, b, c, f\}$, from d we reach all the nodes, from e we reach $\{a, b, e, f\}$, and from f we can reach only $\{a, f\}$.

For another example, consider the graph of Fig. 9.26. From a we can reach all the nodes. From any node but a, we can reach all the nodes except a. ✦



(a) A directed acyclic graph.



(b) A depth-first search forest.

Fig. 9.37. Topologically sorting an acyclic graph.

Running Time of the Reachability Test

Let us assume we have a directed graph G with n nodes and m arcs. We shall also assume G is represented by the data type `GRAPH` of the previous section. First, suppose we want to find the reachable set for a node u. Initializing the nodes to be `UNVISITED` takes $O(n)$ time. The call to `dfs(u, G)` takes $O(m)$ time, and examining the nodes again to see which are visited takes $O(n)$ time. While we examine the nodes, we could also create a list of those nodes that are reachable from u, still taking only $O(n)$ time. Thus, finding the reachable set for one node takes $O(m)$ time.

Now suppose we want the reachable sets for all n nodes. We may repeat the algorithm n times, once for each node. Thus, the total time is $O(nm)$.

Finding Connected Components by Depth-First Search

In Section 9.4, we gave an algorithm for finding the connected components of an undirected graph with n nodes, and with m equal to the larger of the number of nodes and edges, in $O(m \log n)$ time. The tree structure we used to merge components is of interest in its own right; for example, we used it to help implement Kruskal's minimal spanning tree algorithm. However, we can find connected components more efficiently if we use depth-first search.
suffices.

The idea is to treat the undirected graph as if it were a directed graph with each edge replaced by arcs in both directions. If we represent the graph by adjacency lists, then we do not even have to make any change to the representation. Now

Transitive Closure and Reflexive-Transitive Closure

Let R be a binary relation on a set S. The reachability problem can be viewed as computing the *reflexive-transitive closure* of R, which is usually denoted R^*. The relation R^* is defined to be the set of pairs (u, v) such that there is a path of length zero or more from node u to node v in the graph represented by R.

Another relation that is very similar is R^+, the *transitive closure* of R, which is defined to be the set of pairs (u, v) such that there is a path of length one or more from u to v in the graph represented by R. The distinction between R^* and R^+ is that (u, u) is always in R^* for every u in S, whereas (u, u) is in R^+ if and only if there is a cycle of length one or more from u to u. To compute R^+ from R^*, we just have to check whether or not each node u has some entering arc from one of its reachable nodes, including itself; if it does not, we remove u from its own reachable set.

construct the depth-first search forest for the directed graph. Each tree in the forest is one connected component of the undirected graph.

To see why, first note that the presence of an arc $u \to v$ in the directed graph indicates that there is an edge $\{u, v\}$. Thus, all the nodes of a tree are connected.

Now we must show the converse, that if two nodes are connected, then they are in the same tree. Suppose there were a path in the undirected graph between two nodes u and v that are in different trees. Say the tree of u was constructed first. Then there is a path in the directed graph from u to v, which tells us that v, and all the nodes on this path, should have been added to the tree with u. Thus, nodes are connected in the undirected graph if and only if they are in the same tree; that is, the trees are the connected components.

✦ **Example 9.24.** Consider the undirected graph of Fig. 9.4 again. One possible depth-first search forest we might construct for this graph is shown in Fig. 9.38. Notice how the three depth-first search trees correspond to the three connected components. ✦

EXERCISES

9.7.1: Find all the topological orders for the graph of Fig. 9.37.

9.7.2*: Suppose R is a partial order on domain D. We can represent R by its graph, where the nodes are the elements of D and there is an arc $u \to v$ whenever uRv and $u \neq v$. Let $(v_1, v_2, \ldots, v_n)$ be a topological ordering of the graph of R. Let T be the relation defined by $v_i T v_j$ whenever $i \leq j$. Show that

a) T is a total order, and

b) The pairs in R are a subset of the pairs in T; that is, T is a total order containing the partial order R.

9.7.3: Apply depth-first search to the graph of Fig. 9.21 (after converting it to a symmetric directed graph), to find the connected components.

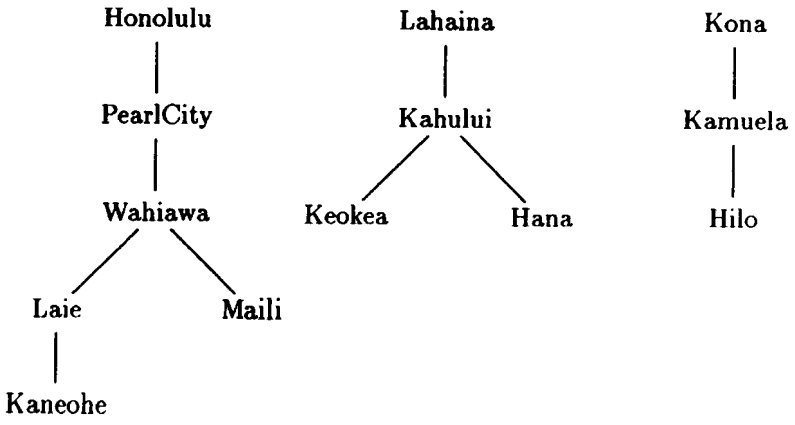


Fig. 9.38. The depth-first search forest divides an undirected graph into connected components.

9.7.4: Consider the graph with arcs $a \to c$, $b \to a$, $b \to c$, $d \to a$, and $e \to c$.

a) Test the graph for cycles.
b) Find all the topological orders for the graph.
c) Find the reachable set of each node.

9.7.5*: In the next section we shall consider the general problem of finding shortest paths from a source node s. That is, we want for each node u the length of the shortest path from s to u if one exists. When we have a directed, acyclic graph, the problem is easier. Give an algorithm that will compute the length of the shortest path from node s to each node u (infinity if no such path exists) in a directed, acyclic graph G. Your algorithm should take $O(m)$ time, where m is the larger of the number of nodes and arcs of G. Prove that your algorithm has this running time. *Hint*: Start with a topological sort of G, and visit each node in turn. On visiting a node u, calculate the shortest distance from s to u in terms of the already calculated shortest distances to the predecessors of u.

9.7.6*: Give algorithms to compute the following for a directed, acyclic graph G. Your algorithms should run in time $O(m)$, where m is the larger of the number of nodes and arcs of G, and you should prove that this running time is all that your algorithm requires. *Hint*: Adapt the idea of Exercise 9.7.5.

a) For each node u, find the length of the longest path from u to anywhere.
b) For each node u, find the length of the longest path to u from anywhere.
c) For a given source node s and for all nodes u of G, find the length of the *longest* path from s to u.
d) For a given source node s and for all nodes u of G, find the length of the longest path from u to s.
e) For each node u, find the length of the longest path through u.

9.8 Dijkstra's Algorithm for Finding Shortest Paths

Suppose we have a graph, which could be either directed or undirected, with labels on the arcs (or edges) to represent the "length" of that arc. An example is Fig. 9.4, which showed the distance along certain roads of the Hawaiian Islands. It is quite common to want to know the minimum distance between two nodes; for example, maps often include tables of driving distance as a guide to how far one can travel in a day, or to help determine which of two routes (that go through different intermediate cities) is shorter. A similar kind of problem would associate with each arc the time it takes to travel along that arc, or perhaps the cost of traveling that arc. Then the minimum "distance" between two nodes would correspond to the traveling time or the fare, respectively.

Minimum distance

In general, the *distance* along a path is the sum of the labels of that path. The *minimum distance from node u to node v* is the minimum of the distance of any path from u to v.

✦ **Example 9.25.** Consider the map of Oahu in Fig. 9.10. Suppose we want to find the minimum distance from Maili to Kaneohe. There are several paths we could choose. One useful observation is that, as long as the labels of the arcs are nonnegative, the minimum-distance path need never have a cycle. For we could skip that cycle and find a path between the same two nodes, but with a distance no greater than that of the path with the cycle. Thus, we need only consider

1. The path through Pearl City and Honolulu.
2. The path through Wahiawa, Pearl City, and Honolulu.
3. The path through Wahiawa and Laie.
4. The path through Pearl City, Wahiawa, and Laie.

The distances of these paths are 44, 51, 67, and 84, respectively. Thus, the minimum distance from Maili to Kaneohe is 44. ✦

Source node

If we wish to find the minimum distance from one given node, called the *source* node, to all the nodes of the graph, one of the most efficient techniques to use is a method called *Dijkstra's algorithm*, the subject of this section. It turns out that if all we want is the distance from one node u to another node v, the best way is to run Dijkstra's algorithm with u as the source node and stop when we deduce the distance to v. If we want to find the minimum distance between every pair of nodes, there is an algorithm that we shall cover in the next section, called Floyd's algorithm, that sometimes is preferable to running Dijkstra's algorithm with every node as a source.

The essence of Dijkstra's algorithm is that we discover the minimum distance from the source to other nodes in the order of those minimum distances, that is, closest nodes first. As Dijkstra's algorithm proceeds, we have a situation like that suggested in Fig. 9.39. In the graph G there are certain nodes that are *settled*, that is, their minimum distance is known; this set always includes s, the source node. For the unsettled nodes v, we record the length of the shortest *special path*, which is a path that starts at the source node, travels only through settled nodes, then at the last step jumps out of the settled region to v.

Settled node

Special path

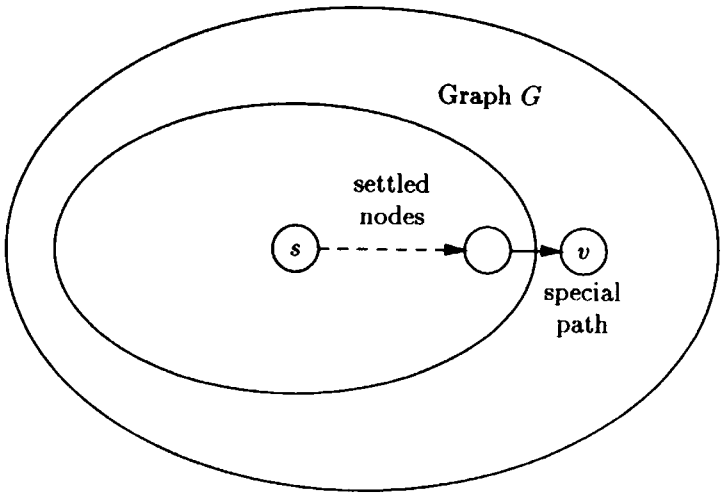


Fig. 9.39. Intermediate stage during the execution of Dijkstra's algorithm.

We maintain a value $dist(u)$ for every node u. If u is a settled node, then $dist(u)$ is the length of the shortest path from the source to u. If u is not settled, then $dist(u)$ is the length of the shortest special path from the source to u. Initially, only the source node s is settled, and $dist(s) = 0$, since the path consisting of s alone surely has distance 0. If there is an arc from s to u, then $dist(u)$ is the label of that arc. Notice that when only s is settled, the only special paths are the arcs out of s, so that $dist(u)$ should be the label of the arc $s \rightarrow u$ if there is one. We shall use a defined constant `INFTY`, that is intended to be larger than the distance along any path in the graph G. `INFTY` serves as an "infinite" value and indicates that no special paths have yet been discovered. That is, initially, if there is no arc $s \rightarrow u$, then $dist(u) =$ `INFTY`.

Now suppose we have some settled and some unsettled nodes, as suggested by Fig. 9.39. We find the node v that is unsettled, but has the smallest $dist$ value of any unsettled node. We "settle" v by

1. Accepting $dist(v)$ as the minimum distance from s to v.
2. Adjusting the value of $dist(u)$, for all nodes u that remain unsettled, to account for the fact that v is now settled.

The adjustment required by step (2) is the following. We compare the old value of $dist(u)$ with the sum of $dist(v)$ and label of the arc $v \rightarrow u$, and if the latter sum is smaller, we replace $dist(u)$ by that sum. If there is no arc $v \rightarrow u$, then we do not adjust $dist(u)$.

✦ **Example 9.26.** Consider the map of Oahu in Fig. 9.10. That graph is undirected, but we shall assume edges are arcs in both directions. Let the source be Honolulu. Then initially, only Honolulu is settled and its distance is 0. We can set $dist$(PearlCity) to 13 and $dist$(Kaneohe) to 11, but other cities, having no arc from Honolulu, are given distance `INFTY`. The situation is shown in the first column of Fig. 9.40. The star on distances indicates that the node is settled.

Among the unsettled nodes, the one with the smallest distance is now Kaneohe,

	ROUND				
CITY	(1)	(2)	(3)	(4)	(5)
Honolulu	0*	0*	0*	0*	0*
PearlCity	13	13	13*	13*	13*
Maili	INFTY	INFTY	33	33	33*
Wahiawa	INFTY	INFTY	25	25*	25*
Laie	INFTY	35	35	35	35
Kaneohe	11	11*	11*	11*	11*

VALUES OF *dist*

Fig. 9.40. Stages in the execution of Dijkstra's algorithm.

and so this node is settled. There are arcs from Kaneohe to Honolulu and Laie. The arc to Honolulu does not help, but the value of *dist*(Kaneohe), which is 11, plus the label of the arc from Kaneohe to Laie, which is 24, totals 35, which is less than "infinity," the current value of *dist*(Laie). Thus, in the second column, we have reduced the distance to Laie to 35. Kaneohe is now settled.

In the next round, the unsettled node with the smallest distance is Pearl City, with a distance of 13. When we make Pearl City settled, we must consider the neighbors of Pearl City, which are Maili and Wahiawa. We reduce the distance to Maili to 33 (the sum of 13 and 20), and we reduce the distance to Wahiawa to 25 (the sum of 13 and 12). The situation is now as in column (3).

Next to be settled is Wahiawa, with a distance of 25, least among the currently unsettled nodes. However, that node does not allow us to reduce the distance to any other node, and so column (4) has the same distances as column (3). Similarly, we next settle Maili, with a distance of 33, but that does not reduce any distances, leaving column (5) the same as column (4). Technically, we have to settle the last node, Laie, but the last node cannot affect any other distances, and so column (5) gives the shortest distances from Honolulu to all six cities. ✦

Why Dijkstra's Algorithm Works

In order to show that Dijkstra's algorithm works, we must assume that the labels of arcs are nonnegative.[9] We shall prove by induction on k that when there are k settled nodes,

a) For each settled node u, *dist*(u) is the minimum distance from s to u, and the shortest path to u consists only of settled nodes.

b) For each unsettled node u, *dist*(u) is the minimum distance of any special path from s to u (**INFTY** if no such path exists).

BASIS. For $k = 1$, s is the only settled node. We initialize *dist*(s) to 0, which satisfies (a). For every other node u, we initialize *dist*(u) to be the label of the arc $s \rightarrow u$ if it exists, and **INFTY** if not. Thus, (b) is satisfied.

[9] When labels are allowed to be negative, we can find graphs for which Dijkstra's algorithm gives incorrect answers.

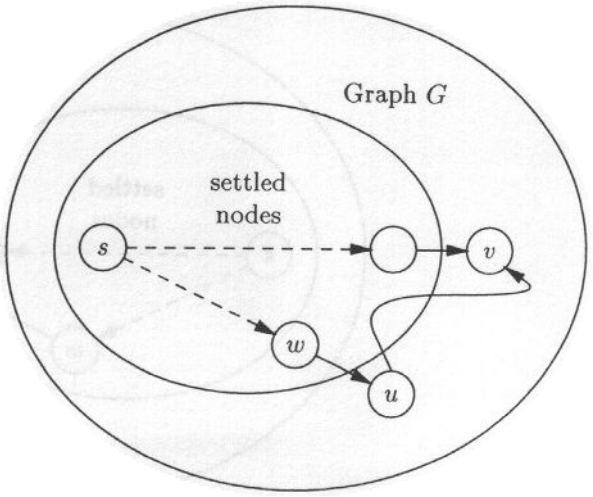


Fig. 9.41. Hypothetical shorter path to v, through w and u.

INDUCTION. Now assume (a) and (b) hold after k nodes have been settled, and let v be the $(k+1)$st node settled. We claim that (a) still holds, because $dist(v)$ is the least distance of any path from s to v. Suppose not. By part (b) of the inductive hypothesis, when k nodes are settled, $dist(v)$ is the minimum distance of any special path to v, and so there must be some shorter nonspecial path to v. As suggested in Fig. 9.41, this path must leave the settled nodes at some node w (which could be s), and go to some unsettled node u. From there, the path could meander in and out of the settled nodes, until it finally arrives at v.

However, v was chosen to be the $(k+1)$st node settled, which means that at this time, $dist(u)$ could not be less than $dist(v)$, or else we would have selected u as the $(k+1)$st node. By (b) of the inductive hypothesis, $dist(u)$ is the minimum length of any special path to u. But the path from s to w to u in Fig. 9.41 is a special path, so that its distance is at least $dist(u)$. Thus, the supposed shorter path from s to v through w and u has a distance that is at least $dist(v)$, because the initial part from s to u already has distance $dist(u)$, and $dist(u) \geq dist(v)$.[10] Thus, (a) holds for $k+1$ nodes, that is, (a) continues to hold when we include v among the settled nodes.

Now we must show that (b) holds when we add v to the settled nodes. Consider some node u that remains unsettled when we add v to the settled nodes. On the shortest special path to u, there must be some penultimate (next-to-last) node; this node could either be v or some other node w. The two possibilities are suggested by Fig. 9.42.

First, suppose the penultimate node is v. Then the length of the path from s to v to u suggested in Fig. 9.42 is $dist(v)$ plus the label of the arc $v \to u$.

Alternatively, suppose the penultimate node is some other node w. By inductive hypothesis (a), the shortest path from s to w consists only of nodes that were settled prior to v, and therefore, v does not appear on the path. Thus, the length of the shortest special path to u does not change when we add v to the settled nodes.

Now recall that when we settle v, we adjust each $dist(u)$ to be the smaller of

[10] Note that the fact that the labels are nonnegative is vital; if not, the portion of the path from u to v could have a negative distance, resulting in a shorter path to v.

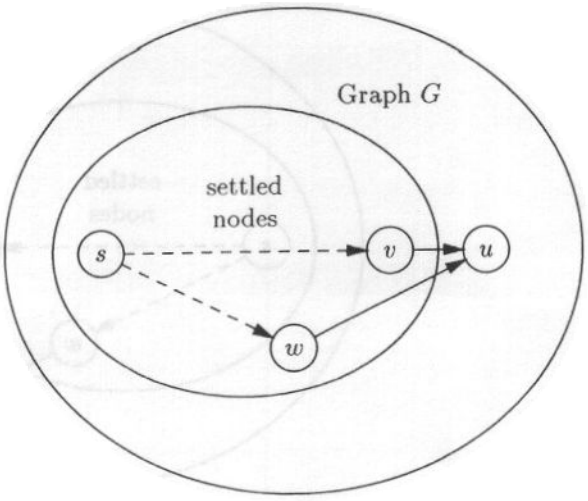


Fig. 9.42. What is the penultimate node on the shortest special path to u?

the old value of $dist(u)$ and $dist(v)$ plus the label of arc $v \to u$. The former covers the case that some w other than v is the penultimate node, and the latter covers the case that v is the penultimate node. Thus, part (b) also holds, and we have completed the inductive step.

Data Structures for Dijkstra's Algorithm

We shall now present an efficient implementation of Dijkstra's algorithm making use of the balanced partially ordered tree structure of Section 5.9.[11] We use two arrays, one called **graph** to represent the graph, and the other called **potNodes** to represent the partially ordered tree. The intent is that to each graph node u there corresponds a partially ordered tree node a that has priority equal to $dist(u)$. However, unlike Section 5.9, we shall organize the partially ordered tree by least priority rather than greatest. (Alternatively, we could take the priority of a to be $-dist(u)$.) Figure 9.43 illustrates the data structure.

We use **NODE** for the type of graph nodes. As usual, we shall name nodes with integers starting at 0. We shall use the type **POTNODE** for the type of nodes in the partially ordered tree. As in Section 5.9, we shall assume that the nodes of the partially ordered tree are numbered starting at 1 for convenience. Thus, both **NODE** and **POTNODE** are synonyms for **int**.

The data type **GRAPH** is defined to be

```
typedef struct {
    float dist;
    LIST successors;
    POTNODE toPOT;
} GRAPH[MAX];
```

[11] Actually, this implementation is only best when the number of arcs is somewhat less than the square of the number of nodes, which is the maximum number of arcs there can be. A simple implementation for the dense case is discussed in the exercises.

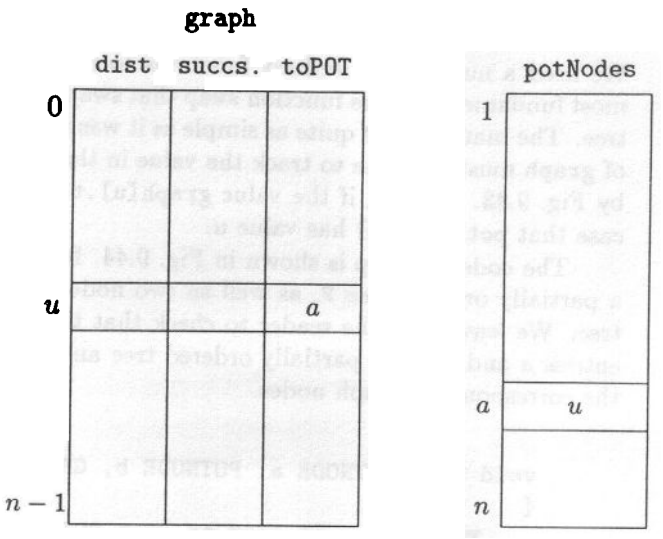


Fig. 9.43. Data structure to represent a graph for Dijkstra's algorithm.

Here, `MAX` is the number of nodes in the graph, and `LIST` is the type of adjacency lists consisting of cells of type `CELL`. Since we need to include labels, which we take to be floating-point numbers, we shall declare as the type `CELL`

```
typedef struct CELL *LIST;
struct CELL {
    NODE nodeName;
    float nodeLabel;
    LIST next;
};
```

We declare the data type `POT` to be an array of graph nodes

```
typedef NODE POT[MAX+1];
```

We can now define the principal data structures:

```
GRAPH graph;
POT potNodes;
POTNODE last;
```

The array of structures `graph` contains the nodes of the graph, the array `potNodes` contains the nodes of the partially ordered tree, and the variable *last* indicates the current end of the partially ordered tree, which resides in `potNodes[1..last]`.

Intuitively, the structure of the partially ordered tree is represented by the positions in the array `potNodes`, as usual for a partially ordered tree. The elements of this array let us tell the priority of a node by referring back to the graph itself. In particular, we place in `potNodes[a]` the index u of the graph node represented. The `dist` field, `graph[u].dist`, gives the priority of node `a` in the partially ordered tree.

Auxiliary Functions for Dijkstra's Algorithm

We need a number of auxiliary functions to make our implementation work. The most fundamental is the function **swap** that swaps two nodes of the partially ordered tree. The matter is not quite as simple as it was in Section 5.9. Here, the field **toPOT** of **graph** must continue to track the value in the array **potNodes**, as was suggested by Fig. 9.43. That is, if the value **graph[u].toPOT** is a, then it must also be the case that **potNodes[a]** has value u.

The code for **swap** is shown in Fig. 9.44. It takes as arguments a graph **G** and a partially ordered tree **P**, as well as two nodes **a** and **b** of that partially ordered tree. We leave it to the reader to check that the function exchanges the values in entries a and b of the partially ordered tree and also exchanges the **toPOT** fields of the corresponding graph nodes.

```
void swap(POTNODE a, POTNODE b, GRAPH G, POT P)
{
    NODE temp; /* used to swap POT nodes */

    temp = P[b];
    P[b] = P[a];
    P[a] = temp;
    G[P[a]].toPOT = a;
    G[P[b]].toPOT = b;
}
```

Fig. 9.44. Function to swap two nodes of the partially ordered tree.

We shall need to bubble nodes up and down the partially ordered tree, as we did in Section 5.9. The major difference is that here, the value in an element of the array **potNodes** is not the priority. Rather, that value takes us to a node of **graph**, and in the structure for that node we find the field **dist**, which gives us the priority. We therefore need an auxiliary function **priority** that returns **dist** for the appropriate graph node. We shall also assume for this section that smaller priorities rise to the top of the partially ordered tree, rather than larger priorities as in Section 5.9.

Figure 9.45 shows the function **priority** and functions **bubbleUp** and **bubbleDown** that are simple modifications of the functions of the same name in Section 5.9. Each takes a graph **G** and a partially ordered tree **P** as arguments. Function **bubbleDown** also needs an integer **last** that indicates the end of the current partially ordered tree in the array **P**.

Initialization

We shall assume that the adjacency list for each graph node has already been created and that a pointer to the adjacency list for graph node u appears in **graph[u].successors**. We shall also assume that node 0 is the source node. If we take the graph node i to correspond to node $i+1$ of the partially ordered tree, then the array **potNodes** is appropriately initialized as a partially ordered tree. That is, the root of the partially ordered tree represents the source node of the graph, to

```
float priority(POTNODE a, GRAPH G, POT P)
{
    return G[P[a]].dist;
}

void bubbleUp(POTNODE a, GRAPH G, POT P)
{
    if ((a > 1) &&
            (priority(a, G, P) < priority(a/2, G, P))) {
        swap(a, a/2, G, P);
        bubbleUp(a/2, G, P);

}

void bubbleDown(POTNODE a, GRAPH G, POT P, int last)
{
    POTNODE child;

    child = 2*a;
    if (child < last &&
           priority(child+1, G, P) < priority(child, G, P))
        ++child;
    if (child <= last &&
            priority(a, G, P) > priority(child, G, P)) {
        swap(a, child, G, P);
        bubbleDown(child, G, P, last);

}
```

Fig. 9.45. Bubbling nodes up and down the partially ordered tree.

which we give priority 0, and to all other nodes we give priority `INFTY`, our "infinite" defined constant.

As we shall see, on the first round of Dijkstra's algorithm, we select the source node to "settle," which will create the condition we regard as our starting point in the informal introduction, where the source node is settled and `dist[u]` is noninfinite only when there is an arc from the source to u. The initialization function is shown in Fig. 9.46. As with previous functions in this section, `initialize` takes as arguments the graph and the partially ordered tree. It also takes a pointer `pLast` to the integer `last`, so it can initialize it to MAX, the number of nodes in the graph. Recall that `last` will indicate the last position in the array for the partially ordered tree that is currently in use.

Note that the indexes of the partially ordered tree are 1 through MAX, while for the graph, they are 0 through $MAX - 1$. Thus, in lines (3) and (4) of Fig. 9.46, we have to make node i of the graph correspond initially to node $i+1$ of the partially ordered tree.

Implementation of Dijkstra's Algorithm

Figure 9.47 shows the code for Dijkstra's algorithm, using all the functions we

```
    void initialize(GRAPH G, POT P, int *pLast);
    {
        int i; /* we use i as both a graph and a tree node */

(1)     for (i = 0; i < MAX; i++) {
(2)         G[i].dist = INFTY;
(3)         G[i].toPOT = i+1;
(4)         P[i+1] = i;
        }
(5)     G[0].dist = 0;
(6)     (*pLast) = MAX;
    }
```

Fig. 9.46. Initialization for Dijkstra's algorithm

Initializing with an Exception

Notice that at line (2) of Fig 9.46, we set `dist[1]` to `INFTY`, along with all the other distances. Then at line (5), we correct this distance to 0. That is more efficient than testing each value of i to see if it is the exceptional case. True, we could eliminate line (5) if we replaced line (2) by

```
if (i == 0)
    G[i].dist = 0;
else
    G[i].dist = INFTY;
```

but that would not only add to the code, it would increase the running time, since we would have to do n tests and n assignments, instead of $n + 1$ assignments and no tests, as we did in lines (2) and (5) of Fig. 9.46.

have previously written. To execute Dijkstra's algorithm on the graph `graph` with partially ordered tree `potNodes` and with integer `last` to indicate the end of the partially ordered tree, we initialize these variables and then call

```
Dijkstra(graph, potNodes, &last)
```

The function `Dijkstra` works as follows. At line (1) we call `initialize`. The remainder of the code, lines (2) through (13), is a loop, each iteration of which corresponds to one round of Dijkstra's algorithm, where we pick one node v and settle it. The node v picked at line (3) is always the one whose corresponding tree node is at the root of the partially ordered tree. At line (4), we take v out of the partially ordered tree, by swapping it with the current last node of that tree. Line (5) actually removes v by decrementing `last`. Then line (6) restores the partially ordered tree property by calling `bubbleDown` on the node we just placed at the root. In effect, unsettled nodes appear below *last* and settled nodes are at *last* and above.

At line (7) we begin updating distances to reflect the fact that v is now settled. Pointer p is initialized to the beginning of the adjacency list for node v. Then in the loop of lines (8) to (13), we consider each successor u of v. After setting variable

```
    void Dijkstra(GRAPH G, POT P, int *pLast)
    {
        NODE u, v; /* v is the node we select to settle */
        LIST ps; /* ps runs down the list of successors of v;
                u is the successor pointed to by ps */

(1)     initialize(G, P, pLast);
(2)     while ((*pLast) > 1) {
(3)         v = P[1];
(4)         swap(1, *pLast, G, P);
(5)         --(*pLast);
(6)         bubbleDown(1, G, P, *pLast);
(7)         ps = G[v].successors;
(8)         while (ps != NULL) {
(9)             u = ps->nodeName;
(10)            if (G[u].dist > G[v].dist + ps->nodeLabel) {
(11)                G[u].dist = G[v].dist + ps->nodeLabel;
(12)                bubbleUp(G[u].toPOT, G, P);
                }
(13)            ps = ps->next;
            }
        }
    }
```

Fig. 9.47. The main function for Dijkstra's algorithm.

u to one of the successors of v at line (9), we test at line (10) whether the shortest special path to u goes through v. That is the case whenever the old value of $dist(u)$, represented in this data structure by `G[u].dist`, is greater than the sum of $dist(v)$ plus the label of the arc $v \rightarrow u$. If so, then at line (11), we set $dist(u)$ to its new, smaller value, and at line (12) we call `bubbleUp`, so, if necessary, u can rise in the partially ordered tree to reflect its new priority. The loop completes when at line (13) we move p down the adjacency list of v.

Running Time of Dijkstra's Algorithm

As in previous sections, we shall assume that our graph has n nodes and that m is the larger of the number of arcs and the number of nodes. We shall analyze the running time of each of the functions, in the order they were described. First, `swap` clearly takes $O(1)$ time, since it consists only of assignment statements. Likewise, `priority` takes $O(1)$ time.

Function `bubbleUp` is recursive, but its running time is $O(1)$ plus the time of a recursive call on a node that is half the distance to the root. As we argued in Section 5.9, there are at most $\log n$ calls, each taking $O(1)$ time, for a total of $O(\log n)$ time for `bubbleUp`. Similarly, `bubbleDown` takes $O(\log n)$ time.

Function `initialize` takes $O(n)$ time. That is, the loop of lines (1) to (4) is iterated n times, and its body takes $O(1)$ time per iteration. That gives $O(n)$ time for the loop. Lines (5) and (6) each contribute $O(1)$, which we may neglect.

Now let us turn our attention to function `Dijkstra` in Fig. 9.47. Let m_v be the out-degree of node v, or equivalently, the length of v's adjacency list. Begin by

analyzing the inner loop of lines (8) to (13). Each of lines (9) to (13) take $O(1)$ time, except for line (12), the call to **bubbleUp**, which we argued takes $O(\log n)$ time. Thus, the body of the loop takes $O(\log n)$ time. The number of times around the loop equals the length of the adjacency list for v, which we referred to as m_v. Thus the running time of the loop of lines (8) through (13) may be taken as $O(1 + m_v \log n)$; the term 1 covers the case where v has no successors, that is, $m_v = 0$, yet we still do the test of line (8).

Now consider the outer loop of lines (2) through (13). We already accounted for lines (8) to (13). Line (6) takes $O(\log n)$ for a call to **bubbleDown**. The other lines of the body take $O(1)$ each. The body thus takes time $O((1 + m_v) \log n)$.

The outer loop is iterated exactly $n - 1$ times, as **last** ranges from n down to 2. The term 1 in $1 + m_v$ thus contributes $n - 1$, or $O(n)$. However, the m_v term must be summed over each node v, since all nodes (but the last) are chosen once to be v. Thus, the contribution of m_v summed over all iterations of the outer loop is $O(m)$, since $\sum_v m_v \leq m$. We conclude that the outer loop takes time $O(m \log n)$. The additional time for line (1), the call to **initialize**, is only $O(n)$, which we may neglect. Our conclusion is that Dijkstra's algorithm takes time $O(m \log n)$, that is, at most a factor of $\log n$ more than the time taken just to look at the nodes and arcs of the graph.

EXERCISES

9.8.1: Find the shortest distance from Detroit to the other cities, according to the graph of Fig. 9.21 (see the exercises for Section 9.4). If a city is unreachable from Detroit, the minimum distance is "infinity."

9.8.2: Sometimes, we wish to count the number of arcs traversed getting from one node to another. For example, we might wish to minimize the number of transfers needed in a plane or bus trip. If we label each arc 1, then a minimum-distance calculation will count arcs. For the graph in Fig. 9.5 (see the exercises for Section 9.2), find the minimum number of arcs needed to reach each node from node a.

9.8.3: In Fig. 9.48(a) are seven species of hominids and their convenient abbreviations. Certain of these species are known to have preceded others because remains have been found in the same place separated by layers indicating that time had elapsed. The table in Fig. 9.48(b) gives triples (x, y, t) that mean species x has been found in the same place as species y, but x appeared t millions of years before y.

a) Draw a directed graph representing the data of Fig. 9.48, with arcs from the earlier species to the later, labeled by the time difference.
b) Run Dijkstra's algorithm on the graph from (a), with AF as the source, to find the shortest time by which each of the other species could have followed AF.

9.8.4*: The implementation of Dijkstra's algorithm that we gave takes $O(m \log n)$ time, which is less than $O(n^2)$ time, except in the case that the number of arcs is close to n^2, its maximum possible number. If m is large, we can devise another implementation, without a priority queue, where we take $O(n)$ time to select the winner at each round, but only $O(m_u)$ time, that is, time proportional to the number of arcs out of the settled node u, to update *dist*. The result is an $O(n^2)$ time algorithm. Develop the ideas suggested here, and write a C program for this implementation of Dijkstra's algorithm.

Australopithecus Afarensis	AF
Australopithecus Africanus	AA
Homo Habilis	HH
Australopithecus Robustus	AR
Homo Erectus	HE
Australopithecus Boisei	AB
Homo Sapiens	HS

(a) Species and abbreviations.

SPECIES 1	SPECIES 2	TIME
AF	HH	1.0
AF	AA	0.8
HH	HE	1.2
HH	AB	0.5
HH	AR	0.3
AA	AB	0.4
AA	AR	0.6
AB	HS	1.7
HE	HS	0.8

(b) Species 1 precedes species 2 by time.

Fig. 9.48. Relationships between hominid species.

9.8.5**: Dijkstra's algorithm does not always work if there are negative labels on some arcs. Give an example of a graph with some negative labels for which Dijkstra's algorithm gives the wrong answer for some minimum distance.

9.8.6**: Let G be a graph for which we have run Dijkstra's algorithm and settled the nodes in some order. Suppose we add to G an arc $u \to v$ with a weight of 0, to form a new graph G'. Under what conditions will Dijkstra's algorithm run on G' settle the nodes in the same order as for G?

9.8.7*: In this section we took the approach of linking the arrays representing the graph G and the partially ordered tree by storing integers that were indices into the other array. Another approach is to use pointers to array elements. Reimplement Dijkstra's algorithm using pointers instead of integer indices.

9.9 Floyd's Algorithm for Shortest Paths

If we want the minimum distances between all pairs of nodes in a graph with n nodes, with nonnegative labels, we can run Dijkstra's algorithm with each of the n nodes as source. Since one run of Dijkstra's algorithm takes $O(m \log n)$ time, where m is the larger of the number of nodes and number of arcs, finding the minimum distances between all pairs of nodes this way takes $O(mn \log n)$ time. Moreover, if m is close to its maximum, n^2, we can use an $O(n^2)$-time implementation of

Dijkstra's algorithm discussed in Exercise 9.8.4, which when run n times gives us an $O(n^3)$-time algorithm to find the minimum distances between each pair of nodes.

There is another algorithm for finding the minimum distances between all pairs of nodes, called *Floyd's algorithm*. This algorithm takes $O(n^3)$ time, and thus is in principle no better than Dijkstra's algorithm, and worse than Dijkstra's algorithm when the number of arcs is much less than n^2. However, Floyd's algorithm works on an adjacency matrix, rather than adjacency lists, and it is conceptually much simpler than Dijkstra's algorithm.

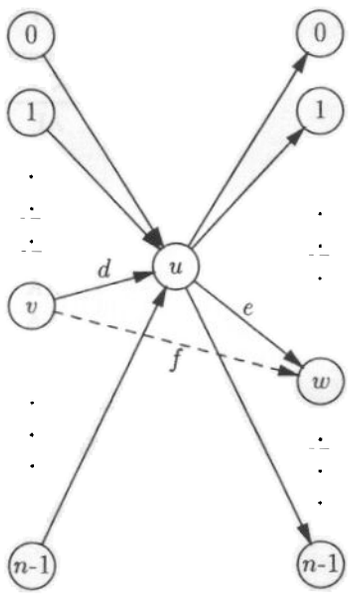


Fig. 9.49. Using node u as a pivot to improve the distances between some pairs of nodes.

Pivot

The essence of Floyd's algorithm is that we consider in turn each node u of the graph as a *pivot*. When u is the pivot, we try to take advantage of u as an intermediate node between all pairs of nodes, as suggested in Fig. 9.49. For each pair of nodes, say v and w, if the sum of the labels of arcs $v \to u$ and $u \to w$, which is $d + e$ in Fig. 9.49, is less than the current label, f, of the arc from v to w, then we replace f by $d + e$.

A fragment of code implementing Floyd's algorithm is shown in Fig. 9.50. As before, we assume nodes are named by integers starting at 0. We use **NODE** as the type of nodes, but we assume this type is integers or an equivalent enumerated type. We assume there is an $n \times n$ array **arc**, such that **arc[v][w]** is the label of the arc $v \to w$ in the given graph. However, on the diagonal we have **arc[v][v]** $= 0$ for all nodes v, even if there is an arc $v \to v$. The reason is that the shortest distance from a node to itself is always 0, and we do not wish to follow any arcs at all. If there is no arc from v to w, then we let **arc[v][w]** be **INFTY**, a special value that is much greater than any other label. There is a similar array **dist** that, at the end, holds the minimum distances; **dist[v][w]** will become the minimum distance from node v to node w.

Lines (1) to (3) initialize **dist** to be **arc**. Lines (4) to (8) form a loop in which

```
        NODE u, v, w;

(1)     for (v = 0; v < MAX; v++)
(2)         for (w = 0; w < MAX; w++)
(3)             dist[v][w] = arc[v][w];
(4)     for (u = 0; u < MAX; u++)
(5)         for (v = 0; v < MAX; v++)
(6)             for (w = 0; w < MAX; w++)
(7)                 if (dist[v][u] + dist[u][w] < dist[v][w])
(8)                     dist[v][w] = dist[v][u] + dist[u][w];
```

Fig. 9.50. Floyd's algorithm.

Warshall's Algorithm

Sometimes, we are only interested in telling whether there exists a path between two nodes, rather than what the minimum distance is. If so, we can use an adjacency matrix where the type of elements is **BOOLEAN** (**int**), with **TRUE** (1) indicating the presence of an arc and **FALSE** (0) its absence. Similarly, the elements of the **dist** matrix are of type **BOOLEAN**, with **TRUE** indicating the existence of a path and **FALSE** indicating that no path between the two nodes in question is known. The only modification we need to make to Floyd's algorithm is to replace lines (7) and (8) of Fig. 9.50 by

```
(7)   if (dist[v][w] == FALSE)
(8)       dist[v][w] = dist[v][u] && dist[u][w];
```

These lines will set **dist[v][w]** to **TRUE**, if it is not already **TRUE**, whenever both **dist[v][u]** and **dist[u][w]** are **TRUE**.

The resulting algorithm, called *Warshall's algorithm*, computes the reflexive and transitive closure of a graph of n nodes in $O(n^3)$ time. That is never better than the $O(nm)$ time that the method of Section 9.7 takes, where we used depth-first search from each node. However, Warshall's algorithm uses an adjacency matrix rather than lists, and if m is near n^2, it may actually be more efficient than multiple depth-first searches because of the simplicity of Warshall's algorithm.

each node u is taken in turn to be the pivot. For each pivot u, in a double loop on v and w, we consider each pair of nodes. Line (7) tests whether it is shorter to go from v to w through u than directly, and if so, line (8) lowers **dist[v][w]** to the sum of the distances from v to u and from u to w.

✦ **Example 9.27.** Let us work with the graph of Fig. 9.10 from Section 9.3, using the numbers 0 through 5 for the nodes; 0 is Laie, 1 is Kaneohe, and so on. Figure 9.51 shows the **arc** matrix, with label **INFTY** for any pair of nodes that do not have a connecting edge. The **arc** matrix is also the initial value of the **dist** matrix.

Note that the graph of Fig. 9.10 is undirected, so the matrix is symmetric; that is, **arc[v][w]** = **arc[v][w]**. If the graph were directed, this symmetry might not

be present, but Floyd's algorithm takes no advantage of symmetry, and thus works for directed or undirected graphs.

	0	1	2	3	4	5
0	0	24	INFTY	INFTY	INFTY	28
1	24	0	11	INFTY	INFTY	INFTY
2	INFTY	11	0	13	INFTY	INFTY
3	INFTY	INFTY	13	0	20	12
4	INFTY	INFTY	INFTY	20	0	15
5	28	INFTY	INFTY	12	15	0

Fig. 9.51. The arc matrix, which is the initial value of the `dist` matrix.

The first pivot is $u = 0$. Since the sum of `INFTY` and anything is `INFTY`, the only pair of nodes v and w, neither of which is u, for which `dist[v][u] + dist[u][w]` is less than `INFTY` is $v = 1$ and $w = 5$, or vice versa.[12] Since `dist[1][5]` is `INFTY` at this time, we replace `dist[1][5]` by the sum of `dist[1][0] + dist[0][5]` which is 52. Similarly, we replace `dist[5][1]` by 52. No other distances can be improved with pivot 0, which leaves the `dist` matrix of Fig. 9.52.

	0	1	2	3	4	5
0	0	24	INFTY	INFTY	INFTY	28
1	24	0	11	INFTY	INFTY	52
2	INFTY	11	0	13	INFTY	INFTY
3	INFTY	INFTY	13	0	20	12
4	INFTY	INFTY	INFTY	20	0	15
5	28	52	INFTY	12	15	0

Fig. 9.52. The matrix `dist` after using 0 as the pivot.

Now we make node 1 the pivot. In the current `dist`, shown in Fig. 9.52, node 1 has noninfinite connections to 0 (distance 24), 2 (distance 11), and 5 (distance 52). We can combine these edges to reduce the distance between nodes 0 and 2 from `INFTY` to $24+11 = 35$. Also, the distance between 2 and 5 is reduced to $11+52 = 63$. Note that 63 is the distance along the path from Honolulu, to Kaneohe, to Laie, to Wahiawa, not the shortest way to get to Wahiawa, but the shortest way that only goes through nodes that have been the pivot so far. Eventually, we shall find the shorter route through Pearl City. The current `dist` matrix is shown in Fig. 9.53.

Now we make 2 be the pivot. Node 2 currently has noninfinite connections to 0 (distance 35), 1 (distance 11), 3 (distance 13), and 5 (distance 63). Among these nodes, the distance between 0 and 3 can be improved to $35 + 13 = 48$, and the

[12] If one of v and w is the u, it is easy to see `dist[v][w]` can never be improved by going through u. Thus, we can ignore pairs of the form (v, u) or (u, w) when searching for pairs whose distance is improved by going through the pivot u.

	0	1	2	3	4	5
0	0	24	35	INFTY	INFTY	28
1	24	0	11	INFTY	INFTY	52
2	35	11	0	13	INFTY	63
3	INFTY	INFTY	13	0	20	12
4	INFTY	INFTY	INFTY	20	0	15
5	28	52	63	12	15	0

Fig. 9.53. The matrix `dist` after using 1 as the pivot.

	0	1	2	3	4	5
0	0	24	35	48	INFTY	28
1	24	0	11	24	INFTY	52
2	35	11	0	13	INFTY	63
3	48	24	13	0	20	12
4	INFTY	INFTY	INFTY	20	0	15
5	28	52	63	12	15	0

Fig. 9.54. The matrix `dist` after using 2 as the pivot.

distance between 1 and 3 can be improved to $11 + 13 = 24$. Thus, the current `dist` matrix is shown in Fig. 9.54.

Next, node 3 becomes the pivot. Figure 9.55 shows the current best distance between 3 and each of the other nodes.[13] By traveling through node 3, we can make the following improvements in distances.

1. Between 1 and 5, the distance is reduced to 36.
2. Between 2 and 5, the distance is reduced to 25.
3. Between 0 and 4, the distance is reduced to 68.
4. Between 1 and 4, the distance is reduced to 44.
5. Between 2 and 4, the distance is reduced to 33.

The current `dist` matrix is shown in Fig. 9.56.

The use of 4 as a pivot does not improve any distances. When 5 is the pivot, we can improve the distance between 0 and 3, since in Fig. 9.56,

```
dist[0][5] + dist[5][3] = 40
```

[13] The reader should compare Fig. 9.55 with Fig. 9.49. The latter shows how to use a pivot node in the general case of a directed graph, where the arcs in and out of the pivot may have different labels. Fig. 9.55 takes advantage of the symmetry in the example graph, letting us use edges between node 3 and the other nodes to represent both arcs into node 3, as on the left of Fig. 9.49, and arcs out of 3, as on the right of Fig. 9.49.

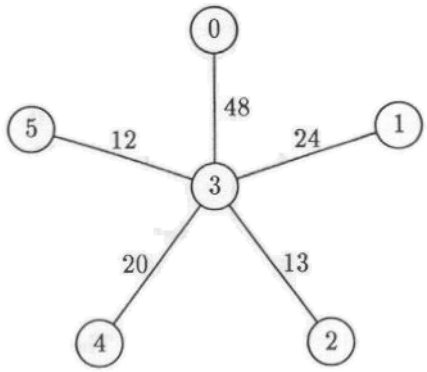


Fig. 9.55. Current best distances to node 4.

	0	1	2	3	4	5
0	0	24	35	48	68	28
1	24	0	11	24	44	36
2	35	11	0	13	33	25
3	48	24	13	0	20	12
4	68	44	33	20	0	15
5	28	36	25	12	15	0

Fig. 9.56. The matrix `dist` after using 3 as the pivot.

	0	1	2	3	4	5
0	0	24	35	40	43	28
1	24	0	11	24	44	36
2	35	11	0	13	33	25
3	40	24	13	0	20	12
4	43	44	33	20	0	15
5	28	36	25	12	15	0

Fig. 9.57. The final `dist` matrix.

which is less than `dist[0][3]`, or 48. In terms of cities, that corresponds to discovering that it is shorter to go from Laie to Pearl City via Wahiawa than via Kaneohe and Honolulu. Similarly, we can improve the distance between 0 and 4 to 43, from 68. The final `dist` matrix is shown in Fig. 9.57. ✦

Why Floyd's Algorithm Works

As we have seen, at any stage during Floyd's algorithm the distance from node v to node w will be the distance of the shortest of those paths that go through only nodes that have been the pivot. Eventually, all nodes get to be the pivot, and `dist[v][w]` holds the minimum distance of all possible paths.

k-path

We define a *k-path* from a node v to a node w to be a path from v to w such that no intermediate node is numbered higher than k. Note that there is no constraint that v or w be k or less.

An important special case is when $k = -1$. Since nodes are assumed numbered starting at 0, a (-1)-path can have no intermediate nodes at all. It can only be either an arc or a single node that is both the beginning and end of a path of length 0.

Figure 9.58 suggests what a k-path looks like, although the end points, v and w, can be above or below k. In that figure, the height of the line represents the numbers of the nodes along the path from v to w.

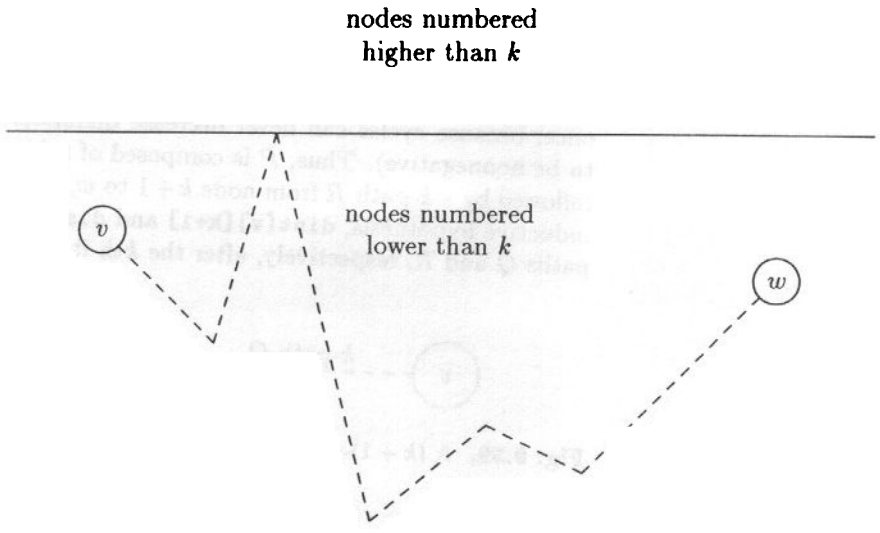


Fig. 9.58. A k-path cannot have nodes higher than k, except (possibly) at the ends.

✦ **Example 9.28.** In Fig. 9.10, the path 0, 1, 2, 3 is a 2-path. The intermediate nodes, 1 and 2, are each 2 or less. This path is also a 3-path, a 4-path, and a 5-path. It is not a 1-path, however, because the intermediate node 2 is greater than 1. Similarly, it is not a 0-path or a (-1)-path. ✦

As we assume nodes are numbered 0 to $n-1$, a (-1)-path cannot have any intermediate nodes at all, and thus must be an arc or a single node. An $(n-1)$-path is any path at all, since there can be no intermediate node numbered higher than $n-1$ in any path of a graph with nodes numbered 0 through $n-1$. We shall prove by induction on k the statement

STATEMENT $S(k)$: If labels of arcs are nonnegative, then just before we set u to $k+1$ in the loop of lines (4) to (8) of Fig. 9.50, `dist[v][w]` is the length of the shortest k-path from v to w, or `INFTY` if there is no such path.

BASIS. The basis is $k = -1$. We set u to 0 just before we execute the body of the loop for the first time. We have just initialized `dist` to be `arc` in lines (1) to (3). Since the arcs and the paths consisting of a node by itself are the only (-1)-paths, the basis holds.

INDUCTION. Assume $S(k)$, and consider what happens to `dist[v][w]` during the iteration of the loop with $u = k + 1$. Suppose there is a shortest $(k+1)$-path from v to w. There are two cases, depending on whether P goes through node $k + 1$.

1. If P is a k-path, that is, P does not actually go through node $k + 1$, then by the inductive hypothesis, `dist[v][w]` already equals the length of P after the kth iteration. We cannot change `dist[u][v]` during the round with $k + 1$ as pivot, because there are no shorter $(k + 1)$-paths.

2. If P is a $(k + 1)$-path, we can assume that P only goes through node $k + 1$ once, because cycles can never decrease distances (recall we require all labels to be nonnegative). Thus, P is composed of a k-path Q from v to node $k + 1$, followed by a k-path R from node $k + 1$ to w, as suggested in Fig. 9.59. By the inductive hypothesis, `dist[v][k+1]` and `dist[k+1][w]` will be the lengths of paths Q and R, respectively, after the kth iteration.

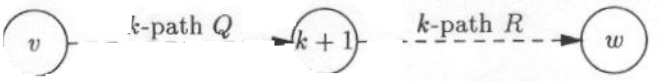


Fig. 9.59. A $(k + 1)$-path P can be broken into two k-paths, Q followed by R.

Let us begin by observing that `dist[v][k+1]` and `dist[k+1][w]` cannot be changed in the $(k+1)$st iteration. The reason is that all arc labels are nonnegative, and so all lengths of paths are nonnegative; thus the test of line (7) in Fig. 9.50 must fail when u (i.e., node $k + 1$) is one of v or w.

Thus, when we apply the test of line (7) for arbitrary v and w, with $u = k+1$, the values of `dist[v][k+1]` and `dist[k+1][w]` have not changed since the end of the kth iteration. That is to say, the test of line (7) compares the length of the shortest k-path, with the sum of the lengths of the shortest k-paths from v to $k+1$ and from $k + 1$ to w. In case (1), where path P does not go through $k + 1$, the former will be the shorter, and in case (2), where P does go through $k + 1$, the latter will be the sum of the lengths of the paths Q and R in Fig. 9.59, and will be the shorter.

We conclude that the $(k + 1)$st iteration sets `dist[v][w]` to the length of the shortest $(k + 1)$-path, for all nodes v and w. That is the statement $S(k + 1)$, and so we conclude the induction.

To finish our proof, we let $k = n - 1$. That is, we know that after finishing all n iterations, `dist[v][w]` is the minimum distance of any $(n - 1)$-path from v to w. But since any path is an $(n - 1)$-path, we have shown that `dist[v][w]` is the minimum distance along any path from v to w.

EXERCISES

9.9.1: Assuming all arcs in Fig. 9.5 (see the exercises for Section 9.2) have label 1, use Floyd's algorithm to find the length of the shortest path between each pair of nodes. Show the distance matrix after pivoting with each node.

9.9.2: Apply Warshall's algorithm to the graph of Fig. 9.5 to compute its reflexive and transitive closure. Show the reachability matrix after pivoting with each node.

9.9.3: Use Floyd's algorithm to find the shortest distances between each pair of cities in the graph of Michigan in Fig. 9.21 (see the exercises for Section 9.4).

9.9.4: Use Floyd's algorithm to find the shortest possible time between each of the hominid species in Fig. 9.48 (see the exercises for Section 9.8).

9.9.5: Sometimes we want to consider only paths of one or more arcs, and exclude single nodes as paths. How can we modify the initialization of the `arc` matrix so that only paths of length 1 or more will be considered when finding the shortest path from a node to itself?

9.9.6*: Find all the acyclic 2-paths in Fig. 9.10.

9.9.7*: Why does Floyd's algorithm not work when there are both positive and negative costs on the arcs?

9.9.8**: Give an algorithm to find the longest acyclic path between two given nodes.

9.9.8**: Suppose we run Floyd's algorithm on a graph G. Then, we lower the label of the arc $u \rightarrow v$ to 0, to construct the new graph G'. For what pairs of nodes s and t will `dist[s][t]` be the same at each round when Floyd's algorithm is applied to G and G'?

9.10 An Introduction to Graph Theory

Graph theory is the branch of mathematics concerned with properties of graphs. In the previous sections, we have presented the basic definitions of graph theory, along with some fundamental algorithms that computer scientists have developed to calculate key properties of graphs efficiently. We have seen algorithms for computing shortest paths, spanning trees, and depth-first-search trees. In this section, we shall present a few more important concepts from graph theory.

Complete Graphs

An undirected graph that has an edge between every pair of distinct nodes is called a *complete* graph. The complete graph with n nodes is called K_n. Figure 9.60 shows the complete graphs K_1 through K_4.

The number of edges in K_n is $n(n-1)/2$, or $\binom{n}{2}$. To see why, consider an edge $\{u, v\}$ of K_n. For u we can pick any of the n nodes; for v we can pick any of the remaining $n-1$ nodes. The total number of choices is therefore $n(n-1)$. However, we count each edge twice that way, once as $\{u, v\}$ and a second time as $\{v, u\}$, so that we must divide the total number of choices by 2 to get the correct number of edges.

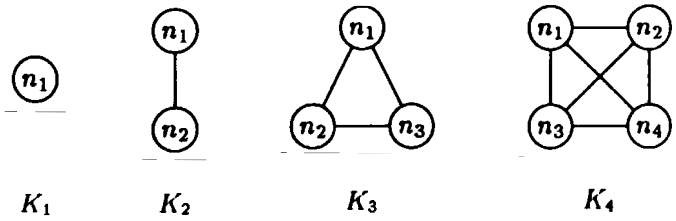


Fig. 9.60. The first four complete graphs.

Complete directed graph

There is also a notion of a complete directed graph. This graph has an arc from every node to every other node, including itself. A complete directed graph with n nodes has n^2 arcs. Figure 9.61 shows the complete directed graph with 3 nodes and 9 arcs.

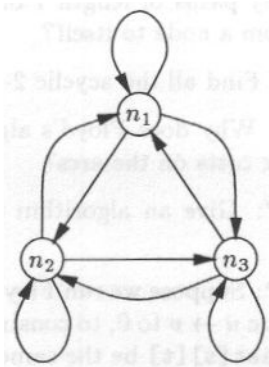


Fig. 9.61. The complete directed graph with three nodes.

Planar Graphs

An undirected graph is said to be *planar* if it is possible to place its nodes on a plane and then draw its edges as continuous lines so that no two edges cross.

✦ **Example 9.29.** The graph K_4 was drawn in Fig. 9.60 in such a way that its two diagonal edges crossed. However, K_4 is a planar graph, as we can see by the drawing in Fig. 9.62. There, by redrawing one of the diagonals on the outside, we avoid having any two edges cross. We say that Fig. 9.62 is a *plane presentation* of the graph K_4, while the drawing in Fig. 9.60 is a nonplane presentation of K_4. Note that it is permissible to have edges that are not straight lines in a plane presentation. ✦

Plane presentation

Nonplanar graph

In Figure 9.63 we see what are in a sense the two simplest *nonplanar* graphs, that is, graphs that do not have any plane presentation. One is K_5, the complete graph with five nodes. The other is sometimes called $K_{3,3}$; it is formed by taking two groups of three nodes and connecting each node of one group to each node of the other group, but not to nodes of the same group. The reader should try to

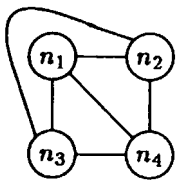
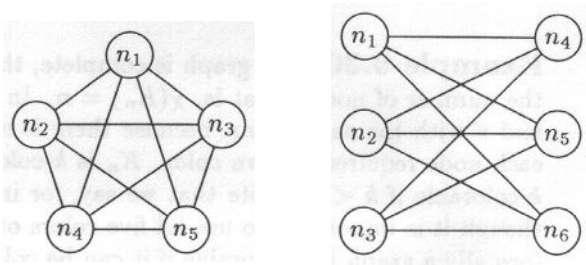


Fig. 9.62. A plane presentation of K_4.



K_5 $K_{3,3}$

Fig. 9.63. The two simplest nonplanar graphs.

redraw each of these graphs so that no two edges cross, just to get a feel for why they are not planar.

Kuratowski's theorem

A famous theorem by Kuratowski states every nonplanar graph contains a "copy" of at least one of these two graphs. We must be a little careful in interpreting the notion of a copy, however, since to see a copy of K_5 or $K_{3,3}$ in an arbitrary nonplanar graph G, we may have to associate some edges in the graphs of Fig. 9.63 with paths in the graph G.

Applications of Planarity

Planarity has considerable importance in computer science. For example, many graphs or similar diagrams need to be presented on a computer screen or on paper. For clarity, it is desirable to make a plane presentation of the graph, or if the graph is not planar, to make as few crossings of edges as possible.

The reader may observe that in Chapter 13 we draw some fairly complex diagrams of circuits, which are really graphs whose nodes are gates and junction points of wires, and whose edges are the wires. Since these circuits are not planar in general, we had to adopt a convention in which wires were allowed to cross without connecting, and a dot signals a connection of wires.

A related application concerns the design of integrated circuits. Integrated circuits, or "chips," embody logical circuits such as those discussed in Chapter 13. They do not require that the logical circuit be inscribed in a plane presentation, but there is a similar limitation that allows us to assign edges to several "levels," often three or four levels. On one level, the graph of the circuit must have a plane presentation; edges are not allowed to cross. However, edges on different levels may cross.

Graph Coloring

Chromatic number

The problem of *graph coloring* for a graph G is to assign a "color" to each node so that no two nodes that are connected by an edge are assigned the same color. We may then ask how many distinct colors are required to *color* a graph in this sense. The minimum number of colors needed for a graph G is called the *chromatic number* of G, often denoted $\chi(G)$. A graph that can be colored with no more than k colors is called *k-colorable*.

k-colorability

✦ **Example 9.30.** If a graph is complete, then its chromatic number is equal to the number of nodes; that is, $\chi(K_n) = n$. In proof, we cannot color two nodes u and v with the same color, because there is surely an edge between them. Thus, each node requires its own color. K_n is k-colorable for each $k \geq n$, but K_n is not k-colorable if $k < n$. Note that we say, for instance, that K_4 is 5-colorable, even though it is impossible to use all five colors on the four-node graph K_4. However, formally a graph is k-colorable if it can be colored with k or fewer colors, not only if it is colorable with exactly k colors.

As another example, the graph $K_{3,3}$ shown in Fig. 9.63 has chromatic number 2. For example, we can color the three nodes in the group on the left *red* and color the three nodes on the right *blue*. Then all edges go between a *red* and a *blue* node.

Bipartite graph

$K_{3,3}$ is an example of a *bipartite graph*, which is another name for a graph that can be colored with two colors. All such graphs can have their nodes divided into two groups such that no edge runs between members of the same group.

As a final example, the chromatic number for the six-node graph of Fig. 9.64 is 4. To see why, note that the node in the center cannot have the same color as any other node, since it is connected to all. Thus, we reserve a color for it, say, *red*. We need at least two other colors for the ring of nodes, since neighbors around the ring cannot get the same color. However, if we try alternating colors — say, *blue* and *green* — as we did in Fig. 9.64, then we run into a problem that the fifth node has both *blue* and *green* neighbors, and therefore needs a fourth color, *yellow*, in our example. ✦

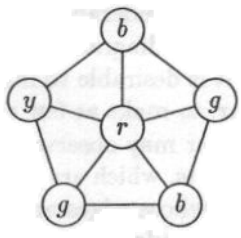


Fig. 9.64. A graph with chromatic number 4.

Applications of Graph Coloring

Finding a good graph coloring is another problem that has many uses in computer science. For example, in our introduction to the first chapter, we considered assigning courses to time slots so that no pair of courses in the same time slot had a student taking both courses. The motivation was to schedule final exams so that no student had to take two exams at the same time. We drew a graph whose

nodes were the courses, with an edge between two courses if they had a student in common.

The question of how many time slots we need in which to schedule exams can thus be posed as the question of what is the chromatic number of this graph. All nodes of the same color can be scheduled at the same time since they have no edges between any two of them. Conversely, if we have a schedule that does not cause conflicts for any student, then we can color all the courses scheduled at the same time with the same color, and thus produce a graph coloring with as many colors as there are exam periods.

In Chapter 1 we discussed a heuristic based on finding maximal independent sets to schedule the exams. That is a reasonable heuristic for finding a good coloring of a graph as well. One might expect that one could try all possible colorings for a graph as small as the five-node graph in Fig. 1.1, and indeed that is true. However, the number of possible colorings of a graph grows exponentially with the number of nodes, and it is not feasible to consider all possible colorings for significantly larger graphs, in our search for the least possible number of colors.

Cliques

k-clique

Clique number

A *clique* in an undirected graph G is a set of nodes such that there is in G an edge between every pair of nodes in the set. A clique of k nodes is called a *k-clique*. The size of the largest clique in a graph is called the *clique number* of that graph.

✦ **Example 9.31.** As a simple example, every complete graph K_n is a clique consisting of all n nodes. In fact, K_n has a k-clique for all $k \leq n$, but no k-clique if $k > n$.

The graph of Fig. 9.64 has cliques of size three, but no greater. The 3-cliques are each shown as triangles. There cannot be a 4-clique in this graph, because it would have to include some of the nodes in the ring. Each ring node is connected to only three other nodes, so the 4-clique would have to include some node v on the ring, its neighbors on the ring, and the central node. However, the neighbors of v on the ring do not have an edge between them, so we do not have a 4-clique. ✦

As an example application of cliques, suppose we represented conflicts among courses not as in Fig. 1.1, but rather by putting an edge between two nodes if they *did not have* a student enrolled in both courses. Thus, two courses connected by an edge could have their exams scheduled at the same time. We could then look for

Maximal clique

maximal cliques, that is, cliques that were not subsets of larger cliques, and schedule the exams for a maximal clique of courses at the same period.

EXERCISES

9.10.1: For the graph of Fig. 9.4,

a) What is the chromatic number?
b) What is the clique number?
c) Give an example of one largest clique.

9.10.2: What are the chromatic numbers of the undirected versions of the graphs shown in (a) Fig. 9.5 and (b) Fig. 9.26? (Treat arcs as edges.)

9.10.3: Figure 9.5 is not presented in a plane manner. Is the graph planar? That is, can you redraw it so there are no crossing edges?

9.10.4*: Three quantities associated with an undirected graph are its degree (maximum number of neighbors of any node), its chromatic number, and its clique number. Derive inequalities that must hold between these quantities. Explain why they must hold.

9.10.5**: Design an algorithm that will take any graph of n nodes, with m the larger of the number of nodes and edges, and in $O(m)$ time will tell whether the graph is bipartite (2-colorable).

9.10.6*: We can generalize the graph of Fig. 9.64 to have a central node and k nodes in a ring, each node connected only to its neighbors around the ring and to the central node. As a function of k, what is the chromatic number of this graph?

9.10.7*: What can you say about the chromatic number of unordered, unrooted trees (as discussed in Section 9.5)?

9.10.8**: Let $K_{i,j}$ be the graph formed by taking a group of i nodes and a group of j nodes and placing an edge from every member of one group to every member of the other group. We observed that if $i = j = 3$, then the resulting graph is not planar. For what values of i and j is the graph $K_{i,j}$ planar?

9.11 Summary of Chapter 9

The table of Fig. 9.65 summarizes the various problems we have addressed in this chapter, the algorithms for solving them, and the running time of the algorithms. In this table, n is the number of nodes in the graph and m is the larger of the number of nodes and the number of arcs/edges. Unless otherwise noted, we assume graphs are represented by adjacency lists.

In addition, we have introduced the reader to most of the key concepts of graph theory. These include

- Paths and shortest paths
- Spanning trees
- Depth-first search trees and forests
- Graph coloring and the chromatic number
- Cliques and clique numbers
- Planar graphs.

PROBLEM	ALGORITHM(S)	RUNNING TIME
Minimal spanning tree	Kruskal's	$O(m \log n)$
Detecting cycles	Depth-first search	$O(m)$
Topological order	Depth-first search	$O(m)$
Single-source reachability	Depth-first search	$O(m)$
Connected components	Depth-first search	$O(m)$
Transitive closure	n depth-first searches	$O(mn)$
Single-source shortest path	Dijkstra's with POT implementation	$O(m \log n)$
	Dijkstra's with implementation of Exercise 9.8.4	$O(n^2)$
All-pairs shortest path	n uses of Dijkstra with POT implementation	$O(mn \log n)$
	n uses of Dijkstra with implementation of Exercise 9.8.4	$O(n^3)$
	Floyd's, with adjacency matrix	$O(n^3)$

Fig. 9.65. A summary of graph algorithms.

9.12 Bibliographic Notes for Chapter 9

For additional material on graph algorithms, see Aho, Hopcroft, and Ullman [1974, 1983]. Depth-first search was first used to create efficient graph algorithms by Hopcroft and Tarjan [1973]. Dijkstra's algorithm is from Dijkstra [1959], Floyd's algorithm from Floyd [1962], Kruskal's algorithm from Kruskal [1956], and Warshall's algorithm from Warshall [1962].

Berge [1962] covers the mathematical theory of graphs. Lawler [1976], Papadimitriou and Steiglitz [1982], and Tarjan [1983] present advanced graph optimization techniques.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman [1974]. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman [1983]. *Data Structures and Algorithms*, Addison-Wesley, Reading, Mass.

Berge, C. [1962]. *The Theory of Graphs and its Applications*, Wiley, New York.

Dijkstra, E. W. [1959]. "A note on two problems in connexion with graphs," *Numberische Mathematik* **1**, pp. 269–271.

Floyd, R. W. [1962]. "Algorithm 97: shortest path," *Comm. ACM* **5**:6, pp. 345.

Hopcroft, J. E., and R. E. Tarjan [1973]. "Efficient algorithms for graph manipulation," *Comm. ACM* **16**:6, pp. 372-378.

Kruskal, J. B., Jr. [1956]. "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proc. AMS* **7**:1, pp. 48–50.

Lawler, E. [1976]. *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York.

Papadimitriou, C. H., and K. Steiglitz [1982]. *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, New Jersey.

Tarjan, R. E. [1983]. *Data Structures and Network Algorithms*, SIAM, Philadelphia.

Warshall, S. [1962]. "A theorem on Boolean matrices," *J. ACM* 9:1, pp. 11-12.