Chapter 4

Arrays, Records and Pointers

4.1 INTRODUCTION

Data structures are classified as either linear or nonlinear. A data structure is said to be linear if its elements form a sequence, or, in other words, a linear list. There are two basic ways of representing such linear structures in memory. One way is to have the linear relationship between the elements represented by means of sequential memory locations. These linear structures are called *arrays* and form the main subject matter of this chapter. The other way is to have the linear relationship between the elements represented by means of pointers or links. These linear structures are called *linked lists*; they form the main content of Chap. 5. Nonlinear structures such as trees and graphs are treated in later chapters.

The operations one normally performs on any linear structure, whether it be an array or a linked list, include the following:

(*a*) *Traversal*. Processing each element in the list.

(*b*) *Search*. Finding the location of the element with a given value or the record with a given key.

(*c*) *Insertion*. Adding a new element to the list.

(*d*) *Deletion*. Removing an element from the list.

(*e*) *Sorting*. Arranging the elements in some type of order.

(*f*) *Merging*. Combining two lists into a single list.

The particular linear structure that one chooses for a given situation depends on the relative frequency with which one performs these different operations on the structure.

This chapter discusses a very common linear structure called an array. Since arrays are usually easy to traverse, search and sort, they are frequently used to store relatively permanent collections of data. On the other hand, if the size of the structure and the data in the structure are constantly changing, then the array may not be as useful a structure as the linked list, discussed in Chap. 5.

4.2 LINEAR ARRAYS

A *linear array* is a list of a finite number n of *homogeneous* data elements (i.e., data elements of the same type) such that:

(*a*) The elements of the array are referenced respectively by an *index set* consisting of n consecutive numbers.

(*b*) The elements of the array are stored respectively in successive memory locations.

The number n of elements is called the *length* or *size* of the array. If not explicitly stated, we will assume the index set consists of the integers $1, 2, \ldots, n$. In general, the length or the number of data elements of the array can be obtained from the index set by the formula

$$\text{Length} = \text{UB} - \text{LB} + 1 \tag{4.1}$$

where UB is the largest index, called the *upper bound*, and LB is the smallest index, called the *lower bound*, of the array. Note that length = UB when LB = 1.

The elements of an array A may be denoted by the subscript notation

$$A_1, A_2, A_3, \ldots, A_n$$

or by the parentheses notation (used in FORTRAN, PL/1 and BASIC)

$$\text{A(1)}, \text{A(2)}, \ldots, \text{A(N)}$$

or by the bracket notation (used in Pascal)

$$A[1], A[2], A[3], \ldots, A[N]$$

We will usually use the subscript notation or the bracket notation. Regardless of the notation, the number K in A[K] is called a *subscript* or an *index* and A[K] is called a *subscripted variable*. Note that subscripts allow any element of A to be referenced by its relative position in A.

EXAMPLE 4.1

(*a*) Let DATA be a 6-element linear array of integers such that

DATA[1] = 247 DATA[2] = 56 DATA[3] = 429 DATA[4] = 135 DATA[5] = 87 DATA[6] = 156

Sometimes we will denote such an array by simply writing

DATA: 247, 56, 429, 135, 87, 156

The array DATA is frequently pictured as in Fig. 4-1(*a*) or Fig. 4-1(*b*).

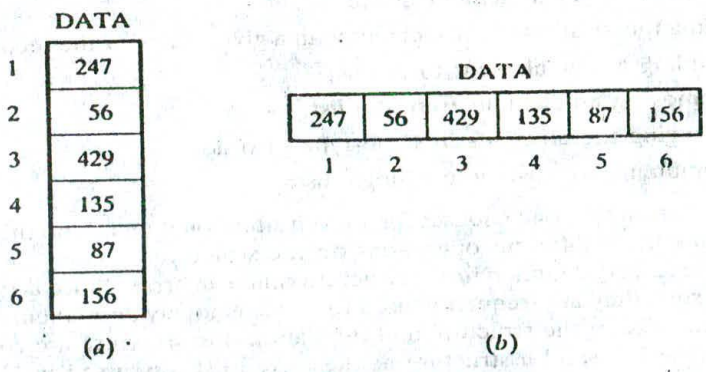


Fig. 4-1

(*b*) An automobile company uses an array AUTO to record the number of automobiles sold each year from 1932 through 1984. Rather than beginning the index set with 1, it is more useful to begin the index set with 1932 so that

AUTO[K] = number of automobiles sold in the year K

Then LB = 1932 is the lower bound and UB = 1984 is the upper bound of AUTO. By Eq. (4.1),

$$\text{Length} = UB - LB + 1 = 1984 - 1930 + 1 = 55$$

That is, AUTO contains 55 elements and its index set consists of all integers from 1932 through 1984.

Each programming language has its own rules for declaring arrays. Each such declaration must give, implicitly or explicitly, three items of information: (1) the name of the array, (2) the data type of the array and (3) the index set of the array.

EXAMPLE 4.2

(*a*) Suppose DATA is a 6-element linear array containing real values. Various programming languages declare such an array as follows:

```
FORTRAN:   REAL DATA(6)
PL/1:      DECLARE DATA(6) FLOAT;
Pascal:    VAR DATA: ARRAY[1 . . 6] OF REAL
```

We will declare such an array, when necessary, by writing DATA(6). (The context will usually indicate the data type, so it will not be explicitly declared.)

(*b*) Consider the integer array AUTO with lower bound LB = 1932 and upper bound UB = 1984. Various programming languages declare such an array as follows:

FORTRAN 77	INTEGER AUTO(1932:1984)
PL/1:	DECLARE AUTO(1932:1984) FIXED;
Pascal	VAR AUTO: ARRAY[1932 . . 1984] of INTEGER

We will declare such an array by writing AUTO(1932:1984).

Some programming languages (e.g., FORTRAN and Pascal) allocate memory space for arrays *statically*, i.e., during program compilation; hence the size of the array is fixed during program execution. On the other hand, some programming languages allow one to read an integer n and then declare an array with n elements; such programming languages are said to allocate memory *dynamically*.

4.3 REPRESENTATION OF LINEAR ARRAYS IN MEMORY

Let LA be a linear array in the memory of the computer. Recall that the memory of the computer is simply a sequence of addressed locations as pictured in Fig. 4-2. Let us use the notation

LOC(LA[K]) = address of the element LA[K] of the array LA

As previously noted, the elements of LA are stored in successive memory cells. Accordingly, the computer does not need to keep track of the address of every element of LA, but needs to keep track only of the address of the first element of LA, denoted by

Base(LA)

and called the *base address* of LA. Using this address *Base*(LA), the computer calculates the address of any element of LA by the following formula:

$$\text{LOC(LA[K])} = \textit{Base}(\text{LA}) + w(\text{K} - \text{lower bound}) \tag{4.2}$$

where w is the number of words per memory cell for the array LA. Observe that the time to calculate LOC(LA[K]) is essentially the same for any value of K. Furthermore, given any subscript K, one can locate and access the content of LA[K] without scanning any other element of LA.

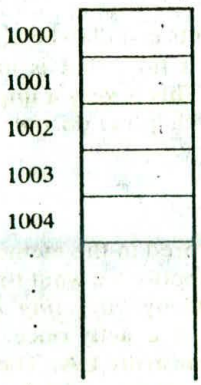


Fig. 4-2 Computer memory.

EXAMPLE 4.3

Consider the array AUTO in Example 4.1(*b*), which records the number of automobiles sold each year from 1932 through 1984. Suppose AUTO appears in memory as pictured in Fig. 4-3. That is, *Base*(AUTO) = 200, and $w = 4$ words per memory cell for AUTO. Then

LOC(AUTO[1932]) = 200, LOC(AUTO[1933]) = 204, LOC(AUTO[1934]) = 208, . . .

The address of the array element for the year K = 1965 can be obtained by using Eq. (4.2):

LOC(AUTO[1965]) = *Base*(AUTO) + w(1965 − lower bound) = 200 + 4(1965 − 1932) = 332

Again we emphasize that the contents of this element can be obtained without scanning any other element in array AUTO.

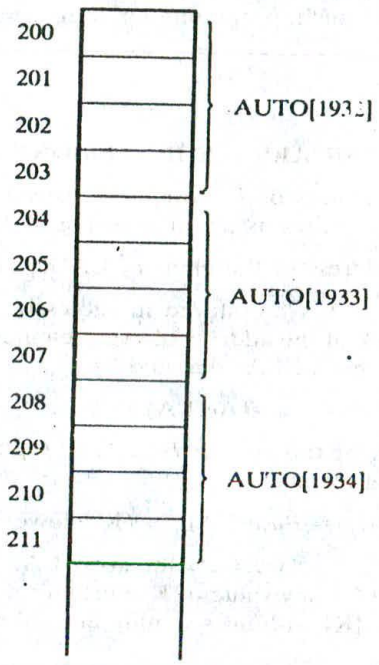


Fig. 4-3

Remark: A collection A of data elements is said to be *indexed* if any element of A, which we shall call A_K, can be located and processed in a time that is independent of K. The above discussion indicates that linear arrays can be indexed. This is very a important property of linear arrays. In fact, linked lists, which are covered in the next chapter, do not have this property

4.4 TRAVERSING LINEAR ARRAYS

Let A be a collection of data elements stored in the memory of the computer. Suppose we want to print the contents of each element of A or suppose we want to count the number of elements of A with a given property. This can be accomplished by *traversing* A, that is, by accessing and processing (frequently called *visiting*) each element of A exactly once.

The following algorithm traverses a linear array LA. The simplicity of the algorithm comes from the fact that LA is a linear structure. Other linear structures, such as linked lists, can also be easily traversed. On the other hand, the traversal of nonlinear structures, such as trees and graphs, is considerably more complicated.

Algorithm 4.1: (Traversing a Linear Array) Here LA is a linear array with lower bound LB and upper bound UB. This algorithm traverses LA applying an operation PROCESS to each element of LA.

1. [Initialize counter.] Set K := LB.
2. Repeat Steps 3 and 4 while K ≤ UB.
3. [Visit element.] Apply PROCESS to LA[K].
4. [Increase counter.] Set K := K + 1.
   [End of Step 2 loop.]
5. Exit.

We also state an alternative form of the algorithm which uses a repeat-for loop instead of the repeat-while loop.

Algorithm 4.1': (Traversing a Linear Array) This algorithm traverses a linear array LA with lower bound LB and upper bound UB.

1. Repeat for K = LB to UB:
   Apply PROCESS to LA[K].
   [End of loop.]
2. Exit.

Caution: The operation PROCESS in the traversal algorithm may use certain variables which must be initialized before PROCESS is applied to any of the elements in the array. Accordingly, the algorithm may need to be preceded by such an initialization step.

EXAMPLE 4.4

Consider the array AUTO in Example 4.1(*b*), which records the number of automobiles sold each year from 1932 through 1984. Each of the following modules, which carry out the given operation, involves traversing AUTO.

(*a*) Find the number NUM of years during which more than 300 automobiles were sold.

1. [Initialization step.] Set NUM := 0.
2. Repeat for K = 1932 to 1984:
   If AUTO[K] > 300, then: Set NUM := NUM + 1.
   [End of loop.]
3. Return.

(*b*) Print each year and the number of automobiles sold in that year.

1. Repeat for K = 1932 to 1984:
   Write: K, AUTO[K].
   [End of loop.]
2. Return.

(Observe that (*a*) requires an initialization step for the variable NUM before traversing the array AUTO.)

4.5 INSERTING AND DELETING

Let A be a collection of data elements in the memory of the computer. "Inserting" refers to the operation of adding another element to the collection A, and "deleting" refers to the operation of removing one of the elements from A. This section discusses inserting and deleting when A is a linear array.

Inserting an element at the "end" of a linear array can be easily done provided the memory space allocated for the array is large enough to accommodate the additional element. On the other hand, suppose we need to insert an element in the middle of the array. Then, on the average, half of the

elements must be moved downward to new locations to accommodate the new element and keep the order of the other elements.

Similarly, deleting an element at the "end" of an array presents no difficulties, but deleting an element somewhere in the middle of the array would require that each subsequent element be moved one location upward in order to "fill up" the array.

Remark: Since linear arrays are usually pictured extending downward, as in Fig. 4-1, the term "downward" refers to locations with larger subscripts, and the term "upward" refers to locations with smaller subscripts.

EXAMPLE 4.5

Suppose TEST has been declared to be a 5-element array but data have been recorded only for TEST[1], TEST[2] and TEST[3]. If X is the value of the next test, then one simply assigns

$$\text{TEST}[4] := \text{X}$$

to add X to the list. Similarly, if Y is the value of the subsequent test, then we simply assign

$$\text{TEST}[5] := \text{Y}$$

to add Y to the list. Now, however, we cannot add any new test scores to the list.

EXAMPLE 4.6

Suppose NAME is an 8-element linear array, and suppose five names are in the array, as in Fig. 4-4(*a*). Observe that the names are listed alphabetically, and suppose we want to keep the array names alphabetical at all times. Suppose Ford is added to the array. Then Johnson, Smith and Wagner must each be moved downward one location, as in Fig. 4-4(*b*). Next suppose Taylor is added to the array; then Wagner must be moved, as in Fig. 4-4(*c*). Last, suppose Davis is removed from the array. Then the five names Ford, Johnson, Smith, Taylor and Wagner must each be moved upward one location, as in Fig. 4-4(*d*). Clearly such movement of data would be very expensive if thousands of names were in the array.

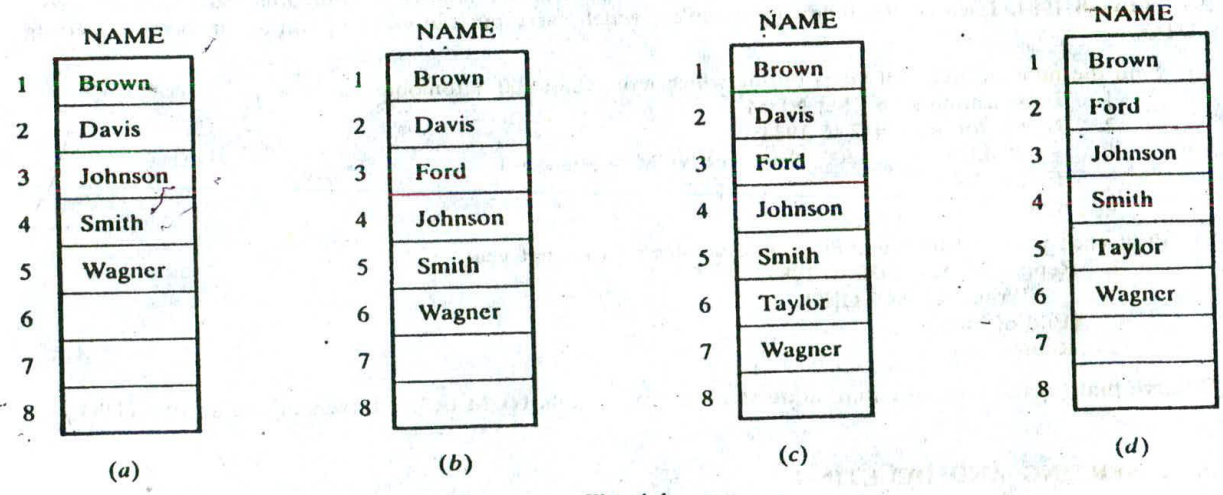


Fig. 4-4

The following algorithm inserts a data element ITEM into the Kth position in a linear array LA with N elements. The first four steps create space in LA by moving downward one location each element from the Kth position on. We emphasize that these elements are moved in reverse order—i.e., first LA[N], then LA[N − 1], . . . , and last LA[K]; otherwise data might be erased. (See Prob. 4.3.) In more detail, we first set J := N and then, using J as a counter, decrease J each time the loop is

executed until J reaches K. The next step, Step 5, inserts ITEM into the array in the space just created. Before the exit from the algorithm, the number N of elements in LA is increased by 1 to account for the new element.

Algorithm 4.2: (Inserting into a Linear Array) INSERT(LA, N, K, ITEM)
Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$. This algorithm inserts an element ITEM into the Kth position in LA.

1. [Initialize counter.] Set J := N.
2. Repeat Steps 3 and 4 while $J \geq K$.
3. [Move Jth element downward.] Set LA[J + 1] := LA[J].
4. [Decrease counter.] Set J := J − 1.
   [End of Step 2 loop.]
5. [Insert element.] Set LA[K] := ITEM.
6. [Reset N.] Set N := N + 1.
7. Exit.

The following algorithm deletes the Kth element from a linear array LA and assigns it to a variable ITEM.

Algorithm 4.3: (Deleting from a Linear Array) DELETE(LA, N, K, ITEM)
Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$. This algorithm deletes the Kth element from LA.

1. Set ITEM := LA[K].
2. Repeat for J = K to N − 1:
   [Move J + 1st element upward.] Set LA[J] := LA[J + 1].
   [End of loop.]
3. [Reset the number N of elements in LA.] Set N := N − 1.
4. Exit.

Remark: We emphasize that if many deletions and insertions are to be made in a collection of data elements, then a linear array may not be the most efficient way of storing the data.

4.6 SORTING; BUBBLE SORT

Let A be a list of n numbers. *Sorting* A refers to the operation of rearranging the elements of A so they are in increasing order, i.e., so that

$$A[1] \leq A[2] < A[3] < \cdots < A[N]$$

For example, suppose A originally is the list

8, 4, 19, 2, 7, 13, 5, 16

After sorting, A is the list

2, 4, 5, 7, 8, 13, 16, 19

Sorting may seem to be a trivial task. Actually, sorting efficiently may be quite complicated. In fact, there are many, many different sorting algorithms; some of these algorithms are discussed in Chap. 9. Here we present and discuss a very simple sorting algorithm known as the *bubble sort*.

Remark: The above definition of sorting refers to arranging numerical data in increasing order; this restriction is only for notational convenience. Clearly, sorting may also mean arranging numerical

data in decreasing order or arranging nonnumerical data in alphabetical order. Actually, A is frequently a file of records, and sorting A refers to rearranging the records of A so that the values of a given key are ordered.

Bubble Sort

Suppose the list of numbers A[1], A[2], . . . , A[N] is in memory. The bubble sort algorithm works as follows:

Step 1. Compare A[1] and A[2] and arrange them in the desired order, so that A[1] < A[2]. Then compare A[2] and A[3] and arrange them so that A[2] < A[3]. Then compare A[3] and A[4] and arrange them so that A[3] < A[4]. Continue until we compare A[N − 1] with A[N] and arrange them so that A[N − 1] < A[N].

Observe that Step 1 involves $n - 1$ comparisons. (During Step 1, the largest element is "bubbled up" to the nth position or "sinks" to the nth position.) When Step 1 is completed, A[N] will contain the largest element.

Step 2. Repeat Step 1 with one less comparison; that is, now we stop after we compare and possibly rearrange A[N − 2] and A[N − 1]. (Step 2 involves N − 2 comparisons and, when Step 2 is completed, the second largest element will occupy A[N − 1].)

Step 3. Repeat Step 1 with two fewer comparisons; that is, we stop after we compare and possibly rearrange A[N − 3] and A[N − 2].

. . . . . .

Step N − 1. Compare A[1] with A[2] and arrange them so that A[1] < A[2].

After $n - 1$ steps, the list will be sorted in increasing order.

The process of sequentially traversing through all or part of a list is frequently called a "pass," so each of the above steps is called a pass. Accordingly, the bubble sort algorithm requires $n - 1$ passes, where n is the number of input items.

EXAMPLE 4.7

Suppose the following numbers are stored in an array A:

32, 51, 27, 85, 66, 23, 13, 57

We apply the bubble sort to the array A. We discuss each pass separately.

Pass 1. We have the following comparisons:

(*a*) Compare A_1 and A_2. Since 32 < 51, the list is not altered.

(*b*) Compare A_2 and A_3. Since 51 > 27, interchange 51 and 27 as follows:

32, (27,) (51,) 85, 66, 23, 13, 57

(*c*) Compare A_3 and A_4. Since 51 < 85, the list is not altered.

(*d*) Compare A_4 and A_5. Since 85 > 66, interchange 85 and 86 as follows:

32, 27, 51, (66,) (85,) 23, 13, 57

(*e*) Compare A_5 and A_6. Since 85 > 23, interchange 85 and 23 as follows:

32, 27, 51, 66, (23,) (85,) 13, 57

(*f*) Compare A_6 and A_7. Since 85 > 13, interchange 85 and 13 to yield:

32, 27, 51, 66, 23, (13,) (85,) 57

(*g*) Compare A_7 and A_8. Since 85 > 57, interchange 85 and 57 to yield:

32, 27, 51, 66, 23, 13, (57,) (85)

At the end of this first pass, the largest number, 85, has moved to the last position. However, the rest of the numbers are not sorted, even though some of them have changed their positions.

For the remainder of the passes, we show only the interchanges.

Pass 2. (27, 33,) 51, 66, 23, 13, 57, 85
27, 33, 51, (23, 66,) 13, 57, 85
27, 33, 51, 23, (13, 66,) 57, 85
27, 33, 51, 23, 13, (57, 66,) 85

At the end of Pass 2, the second largest number, 66, has moved its way down to the next-to-last position.

Pass 3. 27, 33, (23, 51,) 13, 57, 66, 85
27, 33, 23, (13, 51,) 57, 66, 85

Pass 4. 27, (23, 33,) 13, 51, 57, 66, 85
27, 23, (13, 33,) 51, 57, 66, 85

Pass 5. (23, 27,) 13, 33, 51, 57, 66, 85
23, (13, 27,) 33, 51, 57, 66, 85

Pass 6. (13, 23,) 27, 33, 51, 57, 66, 85

Pass 6 actually has two comparisons, A_1 with A_2 and A_2 and A_3. The second comparison does not involve an interchange.

Pass 7. Finally, A_1 is compared with A_2. Since $13 < 23$, no interchange takes place.

Since the list has 8 elements; it is sorted after the seventh pass. (Observe that in this example, the list was actually sorted after the sixth pass. This condition is discussed at the end of the section.)

We now formally state the bubble sort algorithm.

Algorithm 4.4: (Bubble Sort) BUBBLE(DATA, N)
Here DATA is an array with N elements. This algorithm sorts the elements in DATA.

```
1. Repeat Steps 2 and 3 for K = 1 to N − 1.
2.     Set PTR := 1. [Initializes pass pointer PTR.]
3.     Repeat while PTR ≤ N − K: [Executes pass.]
           (a) If DATA[PTR] > DATA[PTR + 1], then:
                   Interchange DATA[PTR] and DATA[PTR + 1].
               [End of If structure.]
           (b) Set PTR := PTR + 1.
       [End of inner loop.]
   [End of Step 1 outer loop.]
4. Exit.
```

Observe that there is an inner loop which is controlled by the variable PTR, and the loop is contained in an outer loop which is controlled by an index K. Also observe that PTR is used as a subscript but K is not used as a subscript, but rather as a counter.

Complexity of the Bubble Sort Algorithm

Traditionally, the time for a sorting algorithm is measured in terms of the number of comparisons. The number $f(n)$ of comparisons in the bubble sort is easily computed. Specifically, there are $n - 1$ comparisons during the first pass, which places the largest element in the last position; there are $n - 2$ comparisons in the second step, which places the second largest element in the next-to-last position; and so on. Thus

$$f(n) = (n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n-1)}{2} = \frac{n^2}{2} + O(n) = O(n^2)$$

In other words, the time required to execute the bubble sort algorithm is proportional to n^2, where n is the number of input items.

Remark: Some programmers use a bubble sort algorithm that contains a 1-bit variable FLAG (or a *logical* variable FLAG) to signal when no interchange takes place during a pass. If FLAG = 0 after any pass, then the list is already sorted and there is no need to continue. This may cut down on the number of passes. However, when using such a flag, one must initialize, change and test the variable FLAG during each pass. Hence the use of the flag is efficient only when the list originally is "almost" in sorted order.

4.7 SEARCHING; LINEAR SEARCH

Let DATA be a collection of data elements in memory, and suppose a specific ITEM of information is given. *Searching* refers to the operation of finding the location LOC of ITEM in DATA, or printing some message that ITEM does not appear there. The search is said to be *successful* if ITEM does appear in DATA and *unsuccessful* otherwise.

Frequently, one may want to add the element ITEM to DATA after an unsuccessful search for ITEM in DATA. One then uses a *search and insertion* algorithm, rather than simply a *search* algorithm; such search and insertion algorithms are discussed in the problem sections.

There are many different searching algorithms. The algorithm that one chooses generally depends on the way the information in DATA is organized. Searching is discussed in detail in Chap. 9. This section discusses a simple algorithm called *linear search*, and the next section discusses the well-known algorithm called *binary search*.

The complexity of searching algorithms is measured in terms of the number $f(n)$ of comparisons required to find ITEM in DATA where DATA contains n elements. We shall show that linear search is a linear time algorithm, but that binary search is a much more efficient algorithm, proportional in time to $\log_2 n$. On the other hand, we also discuss the drawback of relying only on the binary search algorithm.

Linear Search

Suppose DATA is a linear array with n elements. Given no other information about DATA, the most intuitive way to search for a given ITEM in DATA is to compare ITEM with each element of DATA one by one. That is, first we test whether DATA[1] = ITEM, and then we test whether DATA[2] = ITEM, and so on. This method, which traverses DATA sequentially to locate ITEM, is called *linear search* or *sequential search*.

To simplify the matter, we first assign ITEM to DATA[N + 1], the position following the last element of DATA. Then the outcome

$$\text{LOC} = \text{N} + 1$$

where LOC denotes the location where ITEM first occurs in DATA, signifies the search is unsuccessful. The purpose of this initial assignment is to avoid repeatedly testing whether or not we have reached the end of the array DATA. This way, the search must eventually "succeed."

A formal presentation of linear search is shown in Algorithm 4.5.

Observe that Step 1 guarantees that the loop in Step 3 must terminate. Without Step 1 (see Algorithm 2.4), the Repeat statement in Step 3 must be replaced by the following statement, which involves two comparisons, not one:

$$\text{Repeat while LOC} \leq \text{N and DATA[LOC]} \neq \text{ITEM:}$$

On the other hand, in order to use Step 1, one must guarantee that there is an unused memory location

Algorithm 4.5: (Linear Search) LINEAR(DATA, N, ITEM, LOC)
Here DATA is a linear array with N elements, and ITEM is a given item of information. This algorithm finds the location LOC of ITEM in DATA, or sets LOC := 0 if the search is unsuccessful.

1. [Insert ITEM at the end of DATA.] Set DATA[N + 1] := ITEM.
2. [Initialize counter.] Set LOC := 1.
3. [Search for ITEM.]
   Repeat while DATA[LOC] ≠ ITEM:
      Set LOC := LOC + 1.
   [End of loop.]
4. [Successful?] If LOC = N + 1, then: Set LOC := 0.
5. Exit.

at the end of the array DATA; otherwise, one must use the linear search algorithm discussed in Algorithm 2.4.

EXAMPLE 4.8

Consider the array NAME in Fig. 4-5(*a*), where $n = 6$.

(*a*) Suppose we want to know whether Paula appears in the array and, if so, where. Our algorithm temporarily places Paula at the end of the array, as pictured in Fig. 4-5(*b*), by setting NAME[7] = Paula. Then the algorithm searches the array from top to bottom. Since Paula first appears in NAME[N + 1], Paula is not in the original array.

(*b*) Suppose we want to know whether Susan appears in the array and, if so, where. Our algorithm temporarily places Susan at the end of the array, as pictured in Fig. 4-5(*c*), by setting NAME[7] = Susan. Then the algorithm searches the array from top to bottom. Since Susan first appears in NAME[4] (where $4 \leq n$), we know that Susan is in the original array.

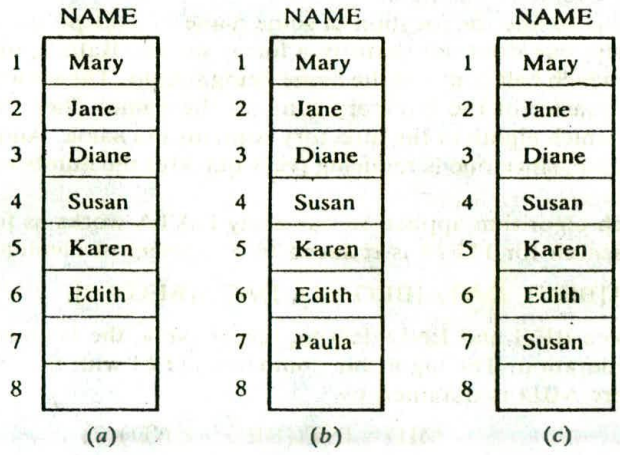


Fig. 4-5

Complexity of the Linear Search Algorithm

As noted above, the complexity of our search algorithm is measured by the number $f(n)$ of comparisons required to find ITEM in DATA where DATA contains n elements. Two important cases to consider are the average case and the worst case.

Clearly, the worst case occurs when one must search through the entire array DATA, i.e., when ITEM does not appear in DATA. In this case, the algorithm requires

$$f(n) = n + 1$$

comparisons. Thus, in the worst case, the running time is proportional to n.

The running time of the average case uses the probabilistic notion of expectation. (See Sec. 2.5.) Suppose p_k is the probability that ITEM appears in DATA[K], and suppose q is the probability that ITEM does not appear in DATA. (Then $p_1 + p_2 + \cdots + p_n + q = 1$.) Since the algorithm uses k comparisons when ITEM appears in DATA[K], the average number of comparisons is given by

$$f(n) = 1 \cdot p_1 + 2 \cdot p_2 + \cdots + n \cdot p_n + (n+1) \cdot q$$

In particular, suppose q is very small and ITEM appears with equal probability in each element of DATA. Then $q \approx 0$ and each $p_i = 1/n$. Accordingly,

$$f(n) = 1 \cdot \frac{1}{n} + 2 \cdot \frac{1}{n} + \cdots + n \cdot \frac{1}{n} + (n+1) \cdot 0 = (1 + 2 + \cdots + n) \cdot \frac{1}{n}$$

$$= \frac{n(n+1)}{2} \cdot \frac{1}{n} = \frac{n+1}{2}$$

That is, in this special case, the average number of comparisons required to find the location of ITEM is approximately equal to half the number of elements in the array.

4.8 BINARY SEARCH

Suppose DATA is an array which is sorted in increasing numerical order or, equivalently, alphabetically. Then there is an extremely efficient searching algorithm, called *binary search*, which can be used to find the location LOC of a given ITEM of information in DATA. Before formally discussing the algorithm, we indicate the general idea of this algorithm by means of an idealized version of a familiar everyday example.

Suppose one wants to find the location of some name in a telephone directory (or some word in a dictionary). Obviously, one does not perform a linear search. Rather, one opens the directory in the middle to determine which half contains the name being sought. Then one opens that half in the middle to determine which quarter of the directory contains the name. Then one opens that quarter in the middle to determine which eighth of the directory contains the name. And so on. Eventually, one finds the location of the name, since one is reducing (very quickly) the number of possible locations for it in the directory.

The binary search algorithm applied to our array DATA works as follows. During each stage of our algorithm, our search for ITEM is reduced to a *segment* of elements of DATA:

$$\text{DATA[BEG], DATA[BEG + 1], DATA[BEG + 2], . . . , DATA[END]}$$

Note that the variables BEG and END denote, respectively, the beginning and end locations of the segment under consideration. The algorithm compares ITEM with the middle element DATA[MID] of the segment, where MID is obtained by

$$\text{MID} = \text{INT}((\text{BEG} + \text{END})/2)$$

(We use INT(A) for the integer value of A.) If DATA[MID] = ITEM, then the search is successful and we set LOC := MID. Otherwise a new segment of DATA is obtained as follows:

(*a*) If ITEM < DATA[MID], then ITEM can appear only in the left half of the segment:

$$\text{DATA[BEG], DATA[BEG + 1], . . . , DATA[MID - 1]}$$

So we reset END := MID − 1 and begin searching again.

(*b*) If ITEM > DATA[MID], then ITEM can appear only in the right half of the segment:

$$DATA[MID + 1], DATA[MID + 2], \ldots, DATA[END]$$

So we reset BEG := MID + 1 and begin searching again.

Initially, we begin with the entire array DATA; i.e., we begin with BEG = 1 and END = n, or, more generally, with BEG = LB and END = UB.

If ITEM is not in DATA, then eventually we obtain

$$END < BEG$$

This condition signals that the search is unsuccessful, and in such a case we assign LOC := NULL. Here NULL is a value that lies outside the set of indices of DATA. (In most cases, we can choose NULL = 0.)

We state the binary search algorithm formally.

Algorithm 4.6: (Binary Search) BINARY(DATA, LB, UB, ITEM, LOC)
Here DATA is a sorted array with lower bound LB and upper bound UB, and ITEM is a given item of information. The variables BEG, END and MID denote, respectively, the beginning, end and middle locations of a segment of elements of DATA. This algorithm finds the location LOC of ITEM in DATA or sets LOC = NULL.

```
1. [Initialize segment variables.]
   Set BEG := LB, END := UB and MID = INT((BEG + END)/2).
2. Repeat Steps 3 and 4 while BEG ≤ END and DATA[MID] ≠ ITEM.
3.     If ITEM < DATA[MID], then:
           Set END := MID − 1.
       Else:
           Set BEG := MID + 1.
       [End of If structure.]
4.     Set MID := INT((BEG + END)/2).
   [End of Step 2 loop.]
5. If DATA[MID] = ITEM, then:
       Set LOC := MID.
   Else:
       Set LOC := NULL.
   [End of If structure.]
6. Exit.
```

Remark: Whenever ITEM does not appear in DATA, the algorithm eventually arrives at the stage that BEG = END = MID. Then the next step yields END < BEG, and control transfers to Step 5 of the algorithm. This occurs in part (*b*) of the next example.

EXAMPLE 4.9

Let DATA be the following sorted 13-element array:

DATA: 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99

We apply the binary search to DATA for different values of ITEM.

(*a*) Suppose ITEM = 40. The search for ITEM in the array DATA is pictured in Fig. 4-6, where the values of DATA[BEG] and DATA[END] in each stage of the algorithm are indicated by circles and the value of

DATA[MID] by a square. Specifically, BEG, END and MID will have the following successive values:

(1) Initially, BEG = 1 and END = 13. Hence

$$MID = INT[(1 + 13)/2] = 7 \quad \text{and so} \quad DATA[MID] = 55$$

(2) Since $40 < 55$, END has its value changed by END = MID − 1 = 6. Hence

$$MID = INT[(1 + 6)/2] = 3 \quad \text{and so} \quad DATA[MID] = 30$$

(3) Since $40 > 30$, BEG has its value changed by BEG = MID + 1 = 4. Hence

$$MID = INT[(4 + 6)/2] = 5 \quad \text{and so} \quad DATA[MID] = 40$$

We have found ITEM in location LOC = MID = 5.

(1) (11, 22, 30, 33, 40, 44, [55,] 60, 66, 77, 80, 88, (99)
(2) (11, 22, [30,] 33, 40, (44,) 55, 60, 66, 77, 80, 88, 99
(3) 11, 22, 30, (33,) [40,] (44,) 55, 60, 66, 77, 80, 88, 99 [Successful]

Fig. 4-6 Binary search for ITEM = 40.

(*b*) Suppose ITEM = 85. The binary search for ITEM is pictured in Fig. 4-7. Here BEG, END and MID will have the following successive values:

(1) Again initially, BEG = 1, END = 13, MID = 7 and DATA[MID] = 55.
(2) Since $85 > 55$, BEG has its value changed by BEG = MID + 1 = 8. Hence

$$MID = INT[(8 + 13)/2] = 10 \quad \text{and so} \quad DATA[MID] = 77$$

(3) Since $85 > 77$, BEG has its value changed by BEG = MID + 1 = 11. Hence

$$MID = INT[(11 + 13)/2] = 12 \quad \text{and so} \quad DATA[MID] = 88$$

(4) Since $85 < 88$, END has its value changed by END = MID − 1 = 11. Hence

$$MID = INT[(11 + 11)/2] = 11 \quad \text{and so} \quad DATA[MID] = 80$$

(Observe that now BEG = END = MID = 11.)

Since $85 > 80$, BEG has its value changed by BEG = MID + 1 = 12. But now BEG > END. Hence ITEM does not belong to DATA.

(1) (11,) 22, 30, 33, 40, 44, [55,] 60, 66, 77, 80, 88, (99)
(2) 11, 22, 30, 33, 40, 44, 55, (60,) 66, [77,] 80, 88, (99)
(3) 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, (80,) [88,] (99)
(4) 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, (80,) 88, 99 [Unsuccessful]

Fig. 4-7 Binary search for ITEM = 85.

Complexity of the Binary Search Algorithm

The complexity is measured by the number $f(n)$ of comparisons to locate ITEM in DATA where DATA contains n elements. Observe that each comparison reduces the sample size in half. Hence we require at most $f(n)$ comparisons to locate ITEM where

$$2^{f(n)} > n \quad \text{or equivalently} \quad f(n) = \lfloor \log_2 n \rfloor + 1$$

That is, the running time for the worst case is approximately equal to $\log_2 n$. One can also show that the running time for the average case is approximately equal to the running time for the worst case.

EXAMPLE 4.10

Suppose DATA contains 1 000 000 elements. Observe that

$$2^{10} = 1024 > 1000 \quad \text{and hence} \quad 2^{20} > 1000^2 = 1\,000\,000$$

Accordingly, using the binary search algorithm, one requires only about 20 comparisons to find the location of an item in a data array with 1 000 000 elements.

Limitations of the Binary Search Algorithm

Since the binary search algorithm is very efficient (e.g., it requires only about 20 comparisons with an initial list of 1 000 000 elements), why would one want to use any other search algorithm? Observe that the algorithm requires two conditions: (1) the list must be sorted and (2) one must have direct access to the middle element in any sublist. This means that one must essentially use a sorted array to hold the data. But keeping data in a sorted array is normally very expensive when there are many insertions and deletions. Accordingly, in such situations, one may use a different data structure, such as a linked list or a binary search tree, to store the data.

4.9 MULTIDIMENSIONAL ARRAYS

The linear arrays discussed so far are also called *one-dimensional arrays*, since each element in the array is referenced by a single subscript. Most programming languages allow two-dimensional and three-dimensional arrays, i.e., arrays where elements are referenced, respectively, by two and three subscripts. In fact, some programming languages allow the number of dimensions for an array to be as high as 7. This section discusses these multidimensional arrays.

Two-Dimensional Arrays

A two-dimensional $m \times n$ array A is a collection of $m \cdot n$ data elements such that each element is specified by a pair of integers (such as J, K), called *subscripts*, with the property that

$$1 \le J \le m \quad \text{and} \quad 1 \le K \le n$$

The element of A with first subscript j and second subscript k will be denoted by

$$A_{J,K} \quad \text{or} \quad A[J, K]$$

Two-dimensional arrays are called *matrices* in mathematics and *tables* in business applications; hence two-dimensional arrays are sometimes called *matrix arrays*.

There is a standard way of drawing a two-dimensional $m \times n$ array A where the elements of A form a rectangular array with m rows and n columns and where the element A[J, K] appears in row J and column K. (A *row* is a horizontal list of elements, and a *column* is a vertical list of elements.) Figure 4-8 shows the case where A has 3 rows and 4 columns. We emphasize that each row contains those elements with the same first subscript, and each column contains those elements with the same second subscript.

		Columns			
		1	2	3	4
Rows	1	A[1, 1]	A[1, 2]	A[1, 3]	A[1, 4]
	2	A[2, 1]	A[2, 2]	A[2, 3]	A[2, 4]
	3	A[3, 1]	A[3, 2]	A[3, 3]	A[3, 4]

Fig. 4-8 Two-dimensional 3×4 array A.

EXAMPLE 4.11

Suppose each student in a class of 25 students is given 4 tests. Assuming the students are numbered from 1 to 25, the test scores can be assigned to a 25×4 matrix array SCORE as pictured in Fig. 4-9. Thus SCORE[K, L] contains the Kth student's score on the Lth test. In particular, the second row of the array,

SCORE[2, 1], SCORE[2, 2], SCORE[2, 3], SCORE[2, 4]

contains the four test scores of the second student.

Student	Test 1	Test 2	Test 3	Test 4
1	84	73	88	81
2	95	100	88	96
3	72	66	77	72
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
25	78	82	70	85

Fig. 4-9 Array SCORE.

Suppose A is a two-dimensional $m \times n$ array. The first dimension of A contains the *index set* $1, \ldots, m$, with *lower bound* 1 and *upper bound* m; and the second dimension of A contains the *index set* $1, 2, \ldots, n$, with *lower bound* 1 and *upper bound* n. The *length* of a dimension is the number of integers in its index set. The pair of lengths $m \times n$ (read "m by n") is called the *size* of the array.

Some programming languages allow one to define multidimensional arrays in which the lower bounds are not 1. (Such arrays are sometimes called *nonregular*.) However, the index set for each dimension still consists of the consecutive integers from the lower bound to the upper bound of the dimension. The length of a given dimension (i.e., the number of integers in its index set) can be obtained from the formula

$$\text{Length} = \text{upper bound} - \text{lower bound} + 1 \tag{4.3}$$

(Note that this formula is the same as Eq. (4.1), which was used for linear arrays.) Generally speaking, unless otherwise stated, we will always assume that our arrays are *regular*, that is, that the lower bound of any dimension of an array is equal to 1.

Each programming language has its own rules for declaring multidimensional arrays. (As is the case with linear arrays, all element in such arrays must be of the same data type.) Suppose, for example, that DATA is a two-dimensional 4×8 array with elements of the *real* type. FORTRAN, PL/1 and Pascal would declare such an array as follows:

```
FORTRAN:   REAL DATA(4, 8)
PL/1:      DECLARE DATA(4, 8) FLOAT;
Pascal:    VAR DATA: ARRAY[1 . . 4, 1 . . 8] OF REAL;
```

Observe that Pascal includes the lower bounds even though they are 1.

Remark: Programming languages which are able to declare nonregular arrays usually use a colon to separate the lower bound from the upper bound in each dimension, while using a comma to separate the dimensions. For example, in FORTRAN,

```
INTEGER NUMB(2:5, -3:1)
```

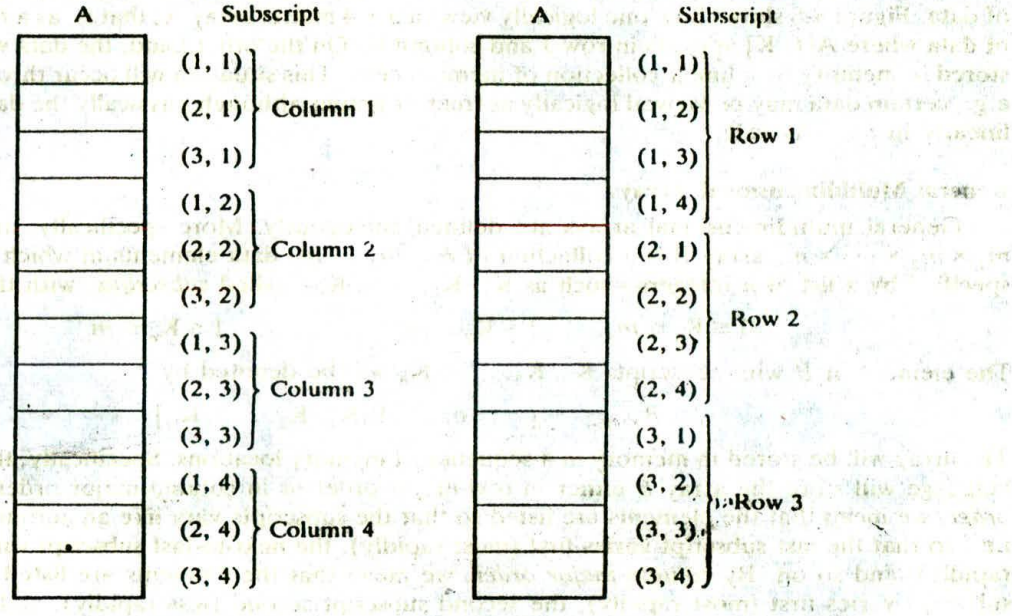
declares NUMB to be a two-dimensional array of the integer type. Here the index sets of the dimensions consist, respectively, of the integers

$$2, 3, 4, 5 \quad \text{and} \quad -3, -2, -1, 0, 1$$

By Eq. (4.3), the length of the first dimension is equal to $5 - 2 + 1 = 4$, and the length of the second dimension is equal to $1 - (-3) + 1 = 5$. Thus NUMB contains $4 \cdot 5 = 20$ elements.

Representation of Two-Dimensional Arrays in Memory

Let A be a two-dimensional $m \times n$ array. Although A is pictured as a rectangular array of elements with m rows and n columns, the array will be represented in memory by a block of $m \cdot n$ sequential memory locations. Specifically, the programming language will store the array A either (1) column by column, is what is called *column-major order*, or (2) row by row, in *row-major order*. Figure 4-10 shows these two ways when A is a two-dimensional 3×4 array. We emphasize that the particular representation used depends upon the programming language, not the user.



(*a*) Column-major order. (*b*) Row-major order.

Fig. 4-10

Recall that, for a linear array LA, the computer does not keep track of the address LOC(LA[K]) of every element LA[K] of LA, but does keep track of *Base*(LA), the address of the first element of LA. The computer uses the formula

$$\text{LOC(LA[K])} = \textit{Base}(\text{LA}) + w(\text{K} - 1)$$

to find the address of LA[K] in time independent of K. (Here w is the number of words per memory cell for the array LA, and 1 is the lower bound of the index set of LA.)

A similar situation also holds for any two-dimensional $m \times n$ array A. That is, the computer keeps track of *Base*(A)—the address of the first element A[1, 1] of A—and computes the address LOC(A[J, K]) of A[J, K] using the formula

$$\text{(Column-major order)} \qquad \text{LOC(A[J, K])} = \textit{Base}(\text{A}) + w[\text{M}(\text{K} - 1) + (\text{J} - 1)] \tag{4.4}$$

or the formula

$$\text{(Row-major order)} \qquad \text{LOC(A[J, K])} = \textit{Base}(\text{A}) + w[\text{N}(\text{J} - 1) + (\text{K} - 1)] \tag{4.5}$$

Again, w denotes the number of words per memory location for the array A. Note that the formulas are linear in J and K, and that one can find the address LOC(A[J, K]) in time independent of J and K.

EXAMPLE 4.12

Consider the 25×4 matrix array SCORE in Example 4.11. Suppose *Base*(SCORE) = 200 and there are $w = 4$ words per memory cell. Furthermore, suppose the programming language stores two-dimensional arrays using row-major order. Then the address of SCORE[12, 3], the third test of the twelfth student, follows:

$$\text{LOC(SCORE[12, 3])} = 200 + 4[4(12 - 1) + (3 - 1)] = 200 + 4[46] = 384$$

Observe that we have simply used Eq. (4.5).

Multidimensional arrays clearly illustrate the difference between the logical and the physical views of data. Figure 4-8 shows how one logically views a 3×4 matrix array A, that is, as a rectangular array of data where A[J, K] appears in row J and column K. On the other hand, the data will be physically stored in memory by a linear collection of memory cells. This situation will occur throughout the text; e.g., certain data may be viewed logically as trees or graphs although physically the data will be stored linearly in memory cells.

General Multidimensional Arrays

General multidimensional arrays are defined analogously. More specifically, an n-dimensional $m_1 \times m_2 \times \cdots \times m_n$ array B is a collection of $m_1 \cdot m_2 \cdots m_n$ data elements in which each element is specified by a list of n integers—such as $K_1, K_2, \ldots, K_n$—called *subscripts*, with the property that

$$1 \le K_1 \le m_1, \qquad 1 \le K_2 \le m_2, \qquad \ldots, \qquad 1 \le K_n \le m_n$$

The element of B with subscripts $K_1, K_2, \ldots, K_n$ will be denoted by

$$B_{K_1, K_2, \ldots K_n} \qquad \text{or} \qquad B[K_1, K_2, \ldots, K_N]$$

The array will be stored in memory in a sequence of memory locations. Specifically, the programming language will store the array B either in row-major order or in column-major order. By *row-major order*, we mean that the elements are listed so that the subscripts vary like an automobile odometer, i.e., so that the last subscript varies first (most rapidly), the next-to-last subscript varies second (less rapidly), and so on. By *column-major order*, we mean that the elements are listed so that the first subscript varies first (most rapidly), the second subscript second (less rapidly), and so on.

EXAMPLE 4.13

Suppose B is a three-dimensional $2 \times 4 \times 3$ array. Then B contains $2 \cdot 4 \cdot 3 = 24$ elements. These 24 elements of B are usually pictured as in Fig. 4-11; i.e., they appear in three layers, called *pages*, where each page consists of the 2×4 rectangular array of elements with the same third subscript. (Thus the three subscripts of an element in a three-dimensional array are called, respectively, the *row*, *column* and *page* of the element.) The two ways of storing B in memory appear in Fig. 4-12. Observe that the arrows in Fig. 4-11 indicate the column-major order of the elements.

The definition of general multidimensional arrays also permits lower bounds other than 1. Let C be such an n-dimensional array. As before, the index set for each dimension of C consists of the consecutive integers from the lower bound to the upper bound of the dimension. The length L_i of dimension i of C is the number of elements in the index set, and L_i can be calculated, as before, from

$$L_i = \text{upper bound} - \text{lower bound} + 1 \tag{4.6}$$

For a given subscript K_i, the effective index E_i of L_i is the number of indices preceding K_i in the index set, and E_i can be calculated from

$$E_i = K_i - \text{lower bound} \tag{4.7}$$

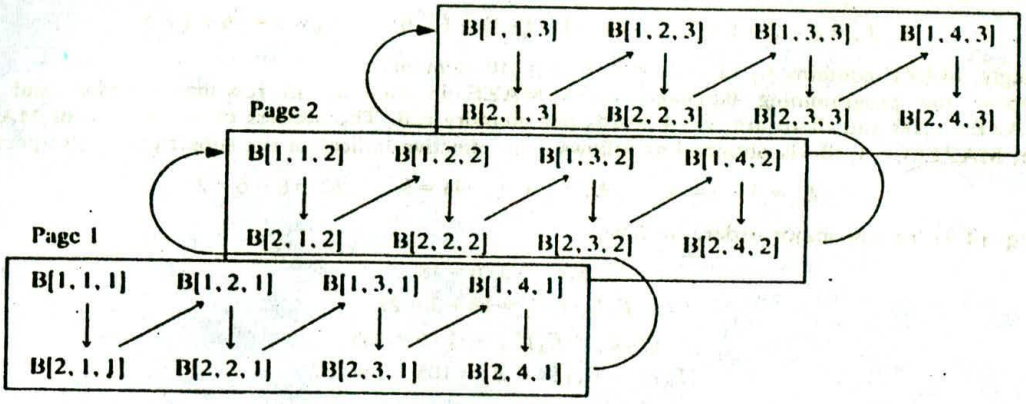
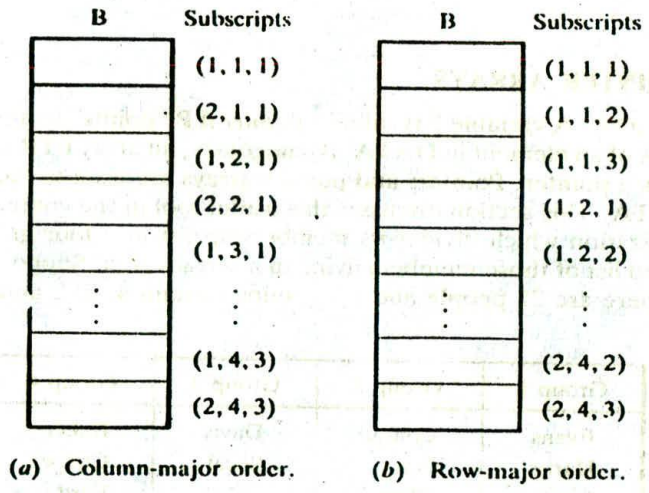


Fig. 4-11



(*a*) Column-major order. (*b*) Row-major order.

Fig. 4-12

Then the address LOC(C[K_1, K_2, . . . , K_N] of an arbitrary element of C can be obtained from the formula

$$Base(C) + w[(((\cdots(E_N L_{N-1} + E_{N-1})L_{N-2}) + \cdots + E_3)L_2 + E_2)L_1 + E_1] \qquad (4.8)$$

or from the formula

$$Base(C) + w[(\cdots((E_1 L_2 + E_2)L_3 + E_3)L_4 + \cdots + E_{N-1})L_N + E_N] \qquad (4.9)$$

according to whether C is stored in column-major or row-major order. Once again, *Base*(C) denotes the address of the first element of C, and w denotes the number of words per memory location.

EXAMPLE 4.14

Suppose a three-dimensional array MAZE is declared using

MAZE(2:8, −4:1, 6:10)

Then the lengths of the three dimensions of MAZE are, respectively,

$$L_1 = 8 - 2 + 1 = 7, \qquad L_2 = 1 - (-4) + 1 = 6, \qquad L_3 = 10 - 6 + 1 = 5$$

Accordingly, MAZE contains $L_1 \cdot L_2 \cdot L_3 = 7 \cdot 6 \cdot 5 = 210$ elements.

Suppose the programming language stores MAZE in memory in row-major order, and suppose $Base(\text{MAZE}) = 200$ and there are $w = 4$ words per memory cell. The address of an element of MAZE—for example, MAZE[5, −1, 8]—is obtained as follows. The effective indices of the subscripts are, respectively,

$$E_1 = 5 - 2 = 3, \qquad E_2 = -1 - (-4) = 3, \qquad E_3 = 8 - 6 = 2$$

Using Eq. (4.9) for row-major order, we have:

$$E_1L_2 = 3 \cdot 6 = 18$$
$$E_1L_2 + E_2 = 18 + 3 = 21$$
$$(E_1L_2 + E_2)L_3 = 21 \cdot 5 = 105$$
$$(E_1L_2 + E_3)L_3 + E_3 = 105 + 2 = 107$$

Therefore,

$$\text{LOC(MAZE[5, −1, 8])} = 200 + 4(107) = 200 + 428 = 628$$

4.10 POINTERS; POINTER ARRAYS

Let DATA be any array. A variable P is called a *pointer* if P "points" to an element in DATA, i.e., if P contains the address of an element in DATA. Analogously, an array PTR is called a *pointer array* if each element of PTR is a pointer. Pointers and pointer arrays are used to facilitate the processing of the information in DATA. This section discusses this useful tool in the context of a specific example.

Consider an organization which divides its membership list into four groups, where each group contains an alphabetized list of those members living in a certain area. Suppose Fig. 4-13 shows such a listing. Observe that there are 21 people and the groups contain 4, 9, 2 and 6 people, respectively.

Group 1	Group 2	Group 3	Group 4
Evans	Conrad	Davis	Baker
Harris	Felt	Segal	Cooper
Lewis	Glass		Ford
Shaw	Hill		Gray
	King		Jones
	Penn		Reed
	Silver		
	Troy		
	Wagner		

Fig. 4-13

Suppose the membership list is to be stored in memory keeping track of the different groups. One way to do this is to use a two-dimensional $4 \times n$ array where each row contains a group, or to use a two-dimensional $n \times 4$ array where each column contains a group. Although this data structure does allow us to access each individual group, much space will be wasted when the groups vary greatly in size. Specifically, the data in Fig. 4-13 will require at least a 36-element 4×9 or 9×4 array to store the 21 names, which is almost twice the space that is necessary. Figure 4-14 shows the representation of the 4×9 array; the asterisks denote data elements and the zeros denote unused storage locations. (Arrays

whose rows—or columns—begin with different numbers of data elements and end with unused storage locations are said to be *jagged*.)

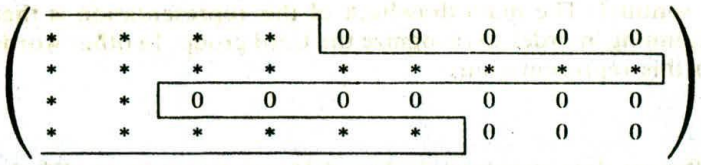


Fig. 4-14 Jagged array.

Another way the membership list can be stored in memory is pictured in Fig. 4-15(*a*). That is, the list is placed in a linear array, one group after another. Clearly, this method is space-efficient. Also, the entire list can easily be processed—one can easily print all the names on the list, for example. On the other hand, there is no way to access any particular group; e.g., there is no way to find and print only the names in the third group.

A modified version of the above method is pictured in Fig. 4-15(*b*). That is, the names are listed in a linear array, group by group, except now some sentinel or marker, such as the three dollar signs used

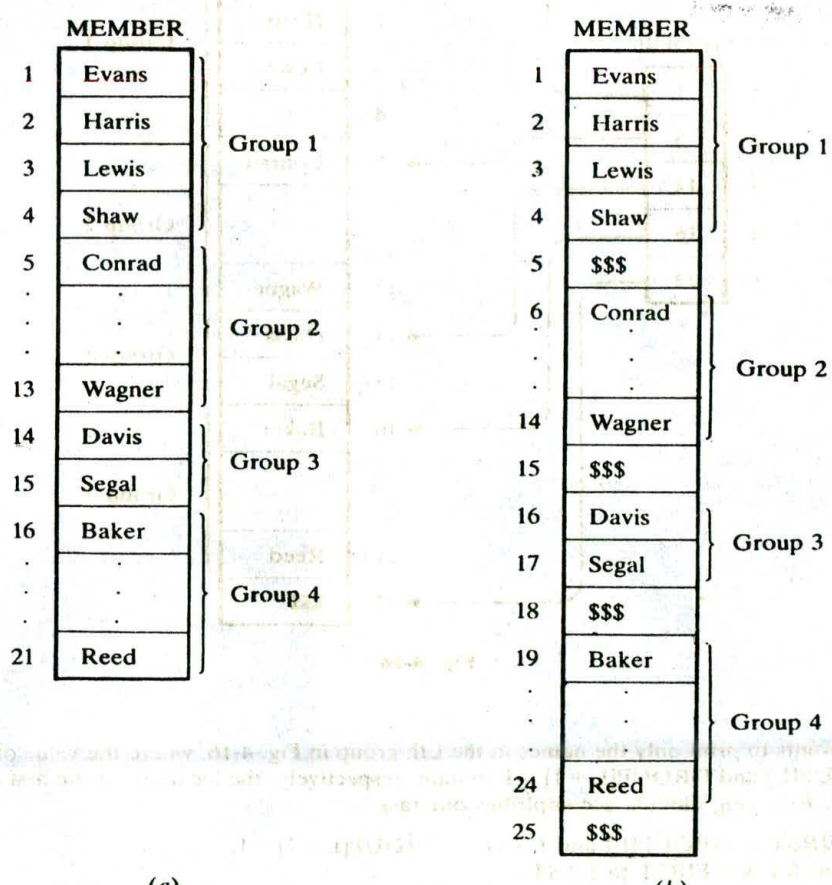


Fig. 4-15

here, will indicate the end of a group. This method uses only a few extra memory cells—one for each group—but now one can access any particular group. For example, a programmer can now find and print those names in the third group by locating those names which appear after the second sentinel and before the third sentinel. The main drawback of this representation is that the list still must be traversed from the beginning in order to recognize the third group. In other words, the different groups are not indexed with this representation.

Pointer Arrays

The two space-efficient data structures in Fig. 4-15 can be easily modified so that the individual groups can be indexed. This is accomplished by using a pointer array (here, GROUP) which contains the locations of the different groups or, more specifically, the locations of the first elements in the different groups. Figure 4-16 shows how Fig. 4-15(*a*) is modified. Observe that GROUP[L] and GROUP[L + 1] − 1 contain, respectively, the first and last elements in group L. (Observe that GROUP[5] points to the sentinel of the list and that GROUP[5] − 1 gives us the location of the last element in Group 4.)

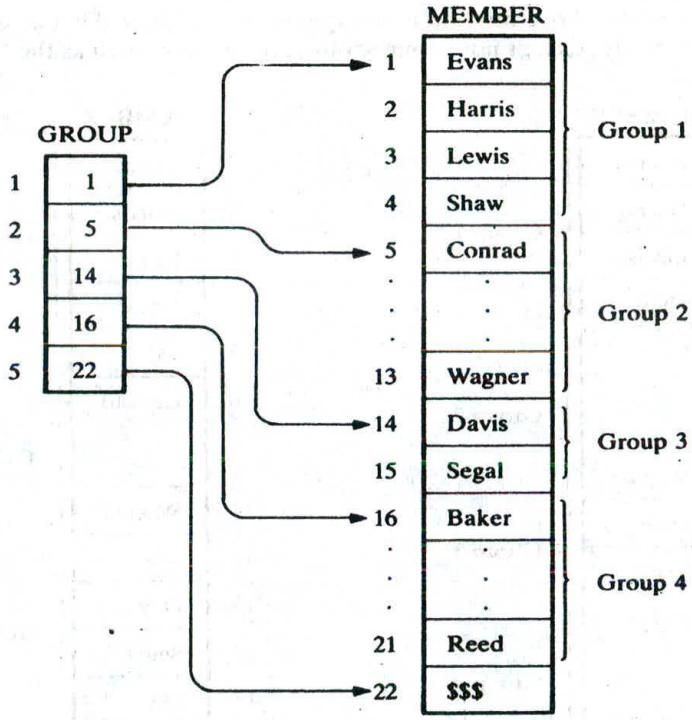


Fig. 4-16

EXAMPLE 4.15

Suppose one wants to print only the names in the Lth group in Fig. 4-16, where the value of L is part of the input. Since GROUP[L] and GROUP[L + 1] − 1 contain, respectively, the locations of the first and last name in the Lth group, the following module accomplishes our task:

1. Set FIRST := GROUP[L] and LAST := GROUP[L + 1] − 1.
2. Repeat for K = FIRST to LAST:
   Write: MEMBER[K].
   [End of loop.]
3. Return.

The simplicity of the module comes from the fact that the pointer array GROUP indexes the Lth group. The variables FIRST and LAST are used mainly for notational convenience.

A slight variation of the data structure in Fig. 4-16 is pictured in Fig. 4-17, where unused memory cells are indicated by the shading. Observe that now there are some empty cells between the groups. Accordingly, a new element may be inserted in a group without necessarily moving the elements in any

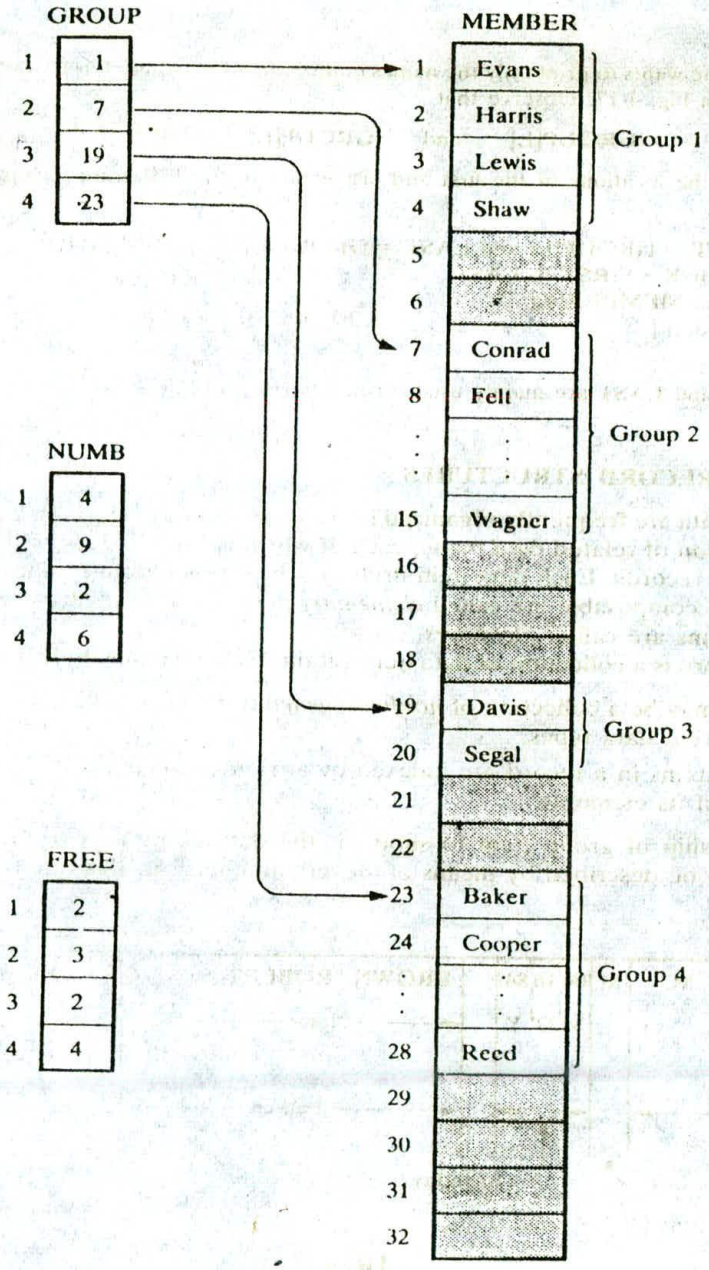


Fig. 4-17

other group. Using this data structure, one requires an array NUMB which gives the number of elements in each group. Observe that GROUP[K + 1] − GROUP[K] is the total amount of space available for Group K; hence

$$FREE[K] = GROUP[K + 1] - GROUP[K] - NUMB[K]$$

is the number of empty cells following GROUP K. Sometimes it is convenient to explicitly define the extra array FREE.

EXAMPLE 4.16

Suppose, again, one wants to print only the names in the Lth group, where L is part of the input, but now the groups are stored as in Fig. 4-17. Observe that

$$GROUP[L] \quad \text{and} \quad GROUP[L] + NUMB[L] - 1$$

contain, respectively, the locations of the first and last names in the Lth group. Thus the following module accomplishes our task:

1. Set FIRST := GROUP[L] and LAST := GROUP[L] + NUMB[L] − 1.
2. Repeat for K = FIRST to LAST:
   Write: MEMBER[K].
   [End of loop.]
3. Return.

The variables FIRST and LAST are mainly used for notational convenience.

4.11 RECORDS; RECORD STRUCTURES

Collections of data are frequently organized into a hierarchy of field, records and files. Specifically, a *record* is a collection of related data items, each of which is called a *field* or *attribute*, and a *file* is a collection of similar records. Each data item itself may be a group item composed of subitems; those items which are indecomposable are called *elementary items* or *atoms* or *scalars*. The names given to the various data items are called *identifiers*.

Although a record is a collection of data items, it differs from a linear array in the following ways:

(*a*) A record may be a collection of *nonhomogeneous* data; i.e., the data items in a record may have different data types.

(*b*) The data items in a record are indexed by attribute names, so there may not be a natural ordering of its elements.

Under the relationship of group item to subitem, the data items in a record form a hierarchical structure which can be described by means of "level" numbers, as illustrated in Examples 4.17 and 4.18.

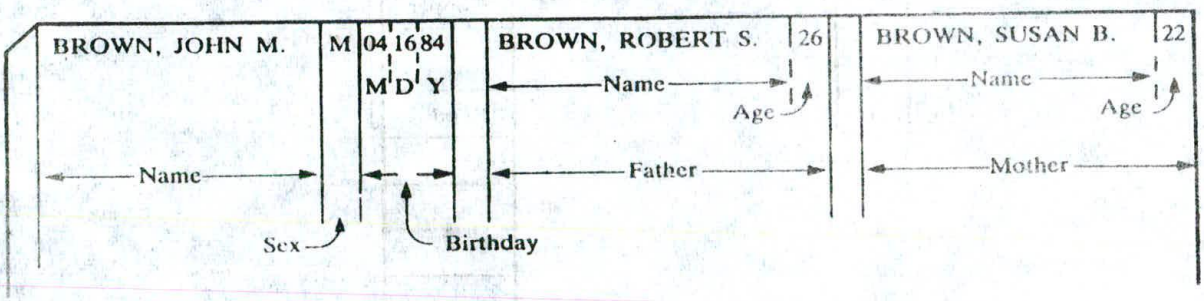


Fig. 4-18

EXAMPLE 4.17

Suppose a hospital keeps a record on each newborn baby which contains the following data items: Name, Sex, Birthday, Father, Mother. Suppose further that Birthday is a group item with subitems Month, Day and Year, and Father and Mother are group items, each with subitems Name and Age. Figure 4-18 shows how such a record could appear.

The structure of the above record is usually described as follows. (Note that Name appears three times and Age appears twice in the structure.)

```
1  Newborn
     2  Name
     2  Sex
     2  Birthday
          3  Month
          3  Day
          3  Year
     2  Father
          3  Name
          3  Age
     2  Mother
          3  Name
          3  Age
```

The number to the left of each identifier is called a level number. Observe that each group item is followed by its subitems, and the level of the subitems is 1 more than the level of the group item. Furthermore, an item is a group item if and only if it is immediately followed by an item with a greater level number.

Some of the identifiers in a record structure may also refer to arrays of elements. In fact, suppose the first line of the above structure is replaced by

```
1  Newborn(20)
```

This will indicate a file of 20 records, and the usual subscript notation will be used to distinguish between different records in the file. That is, we will write

$$\text{Newborn}_1,\quad \text{Newborn}_2,\quad \text{Newborn}_3, \ldots$$

or

$$\text{Newborn}[1],\quad \text{Newborn}[2],\quad \text{Newborn}[3], \ldots$$

to denote different records in the file.

EXAMPLE 4.18

A class of student records may be organized as follows:

```
1  Student(20)
     2  Name
          3  Last
          3  First
          3  MI (Middle Initial)
     2  Test(3)
     2  Final
     2  Grade
```

The identifier Student(20) indicates that there are 20 students. The identifier Test (3) indicates that there are three tests per student. Observe that there are 8 elementary items per Student, since Test is counted 3 times. Altogether, there are 160 elementary items in the entire Student structure.

Indexing Items in a Record

Suppose we want to access some data item in a record. In some cases, we cannot simply write the data name of the item since the same name may appear in different places in the record. For example,

Age appears in two places in the record in Example 4.17. Accordingly, in order to specify a particular item, we may have to *qualify* the name by using appropriate group item names in the structure. This *qualification* is indicated by using decimal points (periods) to separate group items from subitems.

EXAMPLE 4.19

(*a*) Consider the record structure Newborn in Example 4.17. Sex and year need no qualification, since each refer to a unique item in the structure. On the other hand, suppose we want to refer to the age of the father. This can be done by writing

Newborn.Father.Age or simply Father.Age

The first reference is said to be fully qualified. Sometimes one adds qualifying identifiers for clarity.

(*b*) Suppose the first line in the record structure in Example 4.17 is replaced by

1 Newborn(20)

That is, Newborn is defined to be a file with 20 records. Then every item automatically becomes a 20-element array. Some languages allow the sex of the sixth newborn to be referenced by writing

Newborn.Sex[6] or simply Sex[6]

Analogously, the age of the father of the sixth newborn may be referenced by writing

Newborn.Father.Age[6] or simply Father.Age[6]

(*c*) Consider the record structure Student in Example 4.18. Since Student is declared to be a file with 20 students, all items automatically become 20-element arrays. Furthermore, Test becomes a two-dimensional array. In particular, the second test of the sixth student may be referenced by writing

Student.Test[6, 2] or simply Test[6, 2]

The order of the subscripts corresponds to the order of the qualifying identifiers. For example,

Test[3, 1]

does not refer to the third test of the first student, but to the first test of the third student.

Remark: Texts sometimes use functional notation instead of the dot notation to denote qualifying identifiers. For example, one writes

Age(Father(Newborn)) instead of Newborn.Father.Age

and

First(Name(Student[8])) instead of Student.Name.First[8]

Observe that the order of the qualifying identifiers in the functional notation is the reverse of the order in the dot notation.

4.12 REPRESENTATION OF RECORDS IN MEMORY; PARALLEL ARRAYS

Since records may contain nonhomogeneous data, the elements of a record cannot be stored in an array. Some programming languages, such as PL/1, Pascal and COBOL, do have record structures built into the language.

EXAMPLE 4.20

Consider the record structure Newborn in Example 4.17. One can store such a record in PL/1 by the following declaration, which defines a data aggregate called a *structure*:

```
DECLARE 1 NEWBORN,
          2 NAME CHAR(20),
          2 SEX CHAR(1),
          2 BIRTHDAY,
              3 MONTH FIXED,
              3 DAY FIXED,
              3 YEAR FIXED,
          2 FATHER,
              3 NAME CHAR(20),
              3 AGE FIXED,
          2 MOTHER
              3 NAME CHAR(20),
              3 AGE FIXED;
```

Observe that the variables SEX and YEAR are unique; hence references to them need not be qualified. On the other hand, AGE is not unique. Accordingly, one should use

FATHER.AGE or MOTHER.AGE

depending on whether one wants to reference the father's age or the mother's age.

Suppose a programming language does not have available the hierarchical structures that are available in PL/1, Pascal and COBOL. Assuming the record contains nonhomogeneous data, the record may have to be stored in individual variables, one for each of its elementary data items. On the other hand, suppose one wants to store an entire file of records. Note that all data elements belonging to the same identifier do have the same type. Such a file may be stored in memory as a collection of *parallel arrays*; that is, where elements in the different arrays with the same subscript belong to the same record. This is illustrated in the next two examples.

EXAMPLE 4.21

Suppose a membership list contains the name, age, sex and telephone number of each member. One can store the file in four parallel arrays, NAME, AGE, SEX and PHONE, as pictured in Fig. 4-19; that is, for a given subscript K, the elements NAME[K], AGE[K], SEX[K] and PHONE[K] belong to the same record.

	NAME	AGE	SEX	PHONE
1	John Brown	28	Male	234-5186
2	Paul Cohen	33	Male	456-7272
3	Mary Davis	24	Female	777-1212
4	Linda Evans	27	Female	876-4478
5	Mark Green	31	Male	255-7654
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

Fig. 4-19

EXAMPLE 4.22

Consider again the Newborn record in Example 4.17. One can store a file of such records in nine linear arrays, such as

NAME, SEX, MONTH, DAY, YEAR, FATHERNAME, FATHERAGE, MOTHERNAME, MOTHERAGE

one array for each elementary data item. Here we must use different variable names for the name and age of the

father and mother, which was not necessary in the previous example. Again, we assume that the arrays are parallel, i.e., that for a fixed subscript K, the elements

NAME[K], SEX[K], MONTH[K], . . . , MOTHERAGE[K]

belong to the same record.

Records with Variable Lengths

Suppose an elementary school keeps a record for each student which contains the following data: Name, Telephone Number, Father, Mother, Siblings. Here Father, Mother and Siblings contain, respectively, the names of the student's father, mother, and brothers or sisters attending the same school. Three such records may be as follows:

Adams, John;	345-6677;	Richard;	Mary;	Jane, William, Donald
Bailey, Susan;	222-1234;	Steven;	Sheila;	XXXX
Clark, Bruce;	567-3344	XXXX;	Barbara;	David, Lisa

Here XXXX means that the parent has died or is not living with the student, or that the student has no sibling at the school.

The above is an example of a variable-length record, since the data element Siblings can contain zero or more names. One way of storing the file in arrays is pictured in Fig. 4-20, where there are linear arrays NAME, PHONE, FATHER and MOTHER taking care of the first four data items in the records, and arrays NUMB and PTR giving, respectively, the number and location of siblings in an array SIBLING.

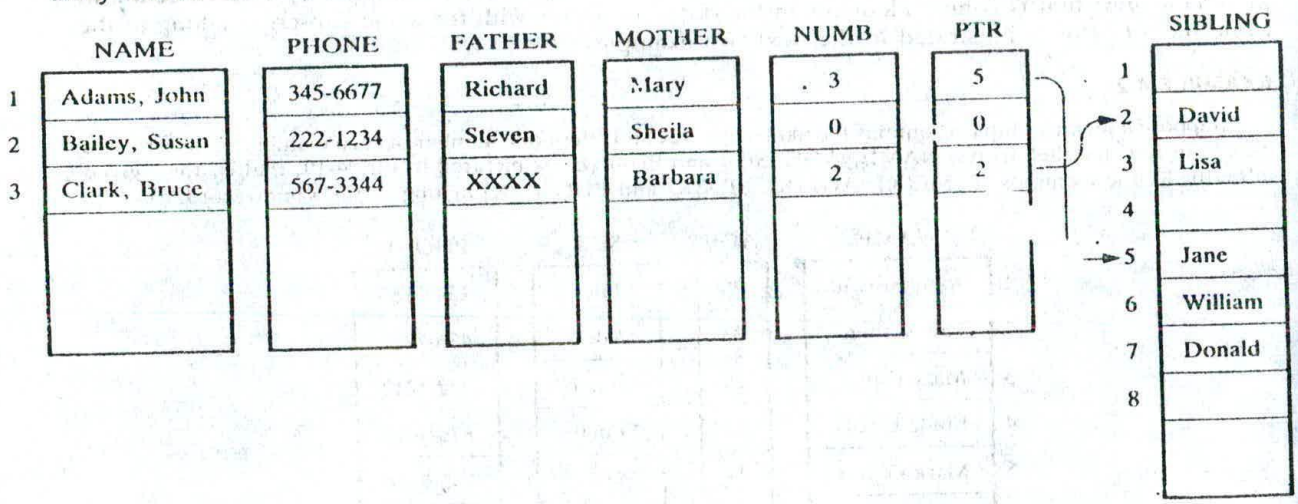


Fig. 4-20

4.13 MATRICES

"Vectors" and "matrices" are mathematical terms which refer to collections of numbers which are analogous, respectively, to linear and two-dimensional arrays. That is,

(*a*) An *n*-element *vector* V is a list of n numbers usually given in the form

$$V = (V_1, V_2, \ldots, V_n)$$

(*b*) An $m \times n$ *matrix* A is an array of $m \cdot n$ numbers arranged in m rows and n columns as follows:

$$A = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ A_{m1} & A_{m2} & \cdots & A_{mn} \end{pmatrix}$$

In the context of vectors and matrices, the term *scalar* is used for individual numbers.

A matrix with one row (column) may be viewed as a vector and, similarly, a vector may be viewed as a matrix with only one row (column).

A matrix with the same number n of rows and columns is called a *square matrix* or an *n-square matrix*. The *diagonal* or *main diagonal* of an n-square matrix A consists of the elements $A_{11}, A_{22}, \ldots, A_{nn}$.

The next section will review certain algebraic operations associated with vectors and matrices. Then the following section discusses efficient ways of storing certain types of matrices, called sparse matrices.

Algebra of Matrices

Suppose A and B are $m \times n$ matrices. The *sum* of A and B, written $A + B$, is the $m \times n$ matrix obtained by adding corresponding elements from A and B; and the *product* of a scalar k and the matrix A, written $k \cdot A$, is the $m \times n$ matrix obtained by multiplying each element of A by k. (Analogous operations are defined for n-element vectors.)

Matrix multiplication is best described by first defining the scalar product of two vectors. Suppose U and V are n-element vectors. Then the *scalar product* of U and V, written $U \cdot V$, is the scalar obtained by multiplying the elements of U by the corresponding elements of V, and then adding:

$$U \cdot V = U_1V_1 + U_2V_2 + \cdots + U_nV_n = \sum_{k=1}^{n} U_kV_k$$

We emphasize that $U \cdot V$ is a scalar, not a vector.

Now suppose A is an $m \times p$ and suppose B is a $p \times n$ matrix. The *product* of A and B, written AB, is the $m \times n$ matrix C whose ijth element C_{ij} is given by

$$C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} + \cdots + A_{ip}B_{pj} = \sum_{k=1}^{p} A_{ik}B_{kj}$$

That is, C_{ij} is equal to the scalar product of row i of A and column j of B.

EXAMPLE 4.23

(*a*) Suppose

$$A = \begin{pmatrix} 1 & -2 & 3 \\ 0 & 4 & 5 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 3 & 0 & -6 \\ 2 & -3 & 1 \end{pmatrix}$$

Then:

$$A + B = \begin{pmatrix} 1+3 & -2+0 & 3+(-6) \\ 0+2 & 4+(-3) & 5+1 \end{pmatrix} = \begin{pmatrix} 4 & -2 & -3 \\ 2 & 1 & 6 \end{pmatrix}$$

$$3A = \begin{pmatrix} 3\cdot 1 & 3\cdot(-2) & 3\cdot 3 \\ 3\cdot 0 & 3\cdot 4 & 3\cdot 5 \end{pmatrix} = \begin{pmatrix} 3 & -6 & 9 \\ 0 & 12 & 15 \end{pmatrix}$$

(*b*) Suppose $U = (1, -3, 4, 5)$, $V = (2, -3, -6, 0)$ and $W = (3, -5, 2, -1)$. Then:

$$U \cdot V = 1\cdot 2 + (-3)\cdot(-3) + 4\cdot(-6) + 5\cdot 0 = 2 + 9 - 24 + 0 = -13$$
$$U \cdot W = 1\cdot 3 + (-3)\cdot(-5) + 4\cdot 2 + 5\cdot(-1) = 3 + 15 + 8 - 5 = 21$$

(*c*) Suppose

$$A = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 2 & 0 & -4 \\ 3 & 2 & 6 \end{pmatrix}$$

The product matrix AB is defined and is a 2×3 matrix. The elements in the first row of AB are obtained, respectively, by multiplying the first row of A by each of the columns of B:

$$\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}\begin{pmatrix} 2 & 0 & -4 \\ 3 & 2 & 6 \end{pmatrix} = \begin{pmatrix} 1\cdot 2 + 3\cdot 3 & 1\cdot 0 + 3\cdot 2 & 1\cdot(-4) + 3\cdot 6 \end{pmatrix} = \begin{pmatrix} 11 & 6 & 14 \end{pmatrix}$$

Similarly, the elements in the second row of AB are obtained, respectively, by multiplying the second row of A by each of the columns of B:

$$\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}\begin{pmatrix} 2 & 0 & -4 \\ 3 & 2 & 6 \end{pmatrix} = \begin{pmatrix} 11 & 6 & 14 \\ 2\cdot 2 + 4\cdot 3 & 2\cdot 0 + 4\cdot 2 & 2\cdot(-4) + 4\cdot 6 \end{pmatrix} = \begin{pmatrix} 11 & 6 & 14 \\ 16 & 8 & 16 \end{pmatrix}$$

That is,

$$AB = \begin{pmatrix} 11 & 6 & 14 \\ 16 & 8 & 16 \end{pmatrix}$$

The following algorithm finds the product AB of matrices A and B, which are stored as two-dimensional arrays. (Algorithms for matrix addition and matrix scalar multiplication, which are very similar to algorithms for vector addition and scalar multiplication, are left as exercises for the reader.)

Algorithm 4.7: (Matrix Multiplication) MATMUL(A, B, C, M, P, N)
Let A be an M × P matrix array, and let B be a P × N matrix array. This algorithm stores the product of A and B in an M × N matrix array C.

```
1. Repeat Steps 2 to 4 for I = 1 to M:
2.     Repeat Steps 3 and 4 for J = 1 to N:
3.         Set C[I, J] := 0. [Initializes C[I, J].]
4.         Repeat for K = 1 to P:
               C[I, J] := C[I, J] + A[I, K] * B[K, J]
           [End of inner loop.]
       [End of Step 2 middle loop.]
   [End of Step 1 outer loop.]
5. Exit.
```

The complexity of a matrix multiplication algorithm is measured by counting the number C of multiplications. The reason that additions are not counted in such algorithms is that computer multiplication takes much more time than computer addition. The complexity of the above Algorithm 4.7 is equal to

$$C = m \cdot n \cdot p$$

This comes from the fact that Step 4, which contains the only multiplication is executed $m \cdot n \cdot p$ times. Extensive research has been done on finding algorithms for matrix multiplication which minimize the number of multiplications. The next example gives an important and surprising result in this area.

EXAMPLE 4.24

Suppose A and B are 2×2 matrices. We have:

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad B = \begin{pmatrix} e & f \\ g & h \end{pmatrix} \quad \text{and} \quad AB = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix}$$

In Algorithm 4.7, the product matrix AB is obtained using $C = 2 \cdot 2 \cdot 2 = 8$ multiplications. On the other hand, AB can also be obtained from the following, which uses only 7 multiplications:

$$AB = \begin{pmatrix} (1+4-5+7) & (3+5) \\ (2+4) & (1+3-2+6) \end{pmatrix}$$

(1) $(a+d)(e+h)$
(2) $(c+d)e$
(3) $a(f-h)$
(4) $d(g-e)$
(5) $(a+b)h$
(6) $(c-a)(e+f)$
(7) $(b-d)(g+h)$

Certain versions of the programming language BASIC have matrix operations built into the language. Specifically, the following are valid BASIC statements where A and B are two-dimensional arrays that have appropriate dimensions and K is a scalar:

```
MAT C = A + B
MAT D = (K) * A
MAT E = A * B
```

Each statement begins with the keyword MAT, which indicates that matrix operations will be performed. Thus C will be the matrix sum of A and B, D will be the scalar product of the matrix A by the scalar K, and E will be the matrix product of A and B.

4.14 SPARSE MATRICES

Matrices with a relatively high proportion of zero entries are called *sparse matrices*. Two general types of n-square sparse matrices, which occur in various applications, are pictured in Fig. 4-21. (It is sometimes customary to omit blocks of zeros in a matrix as in Fig. 4-21.) The first matrix, where all entries above the main diagonal are zero or, equivalently, where nonzero entries can only occur on or below the main diagonal, is called a (*lower*) *triangular matrix*. The second matrix, where nonzero entries can only occur on the diagonal or on elements immediately above or below the diagonal, is called a *tridiagonal matrix*.

$$\begin{pmatrix} 4 & & & & \\ 3 & -5 & & & \\ 1 & 0 & 6 & & \\ -7 & 8 & -1 & 3 & \\ 5 & -2 & 0 & 2 & -8 \end{pmatrix} \qquad \begin{pmatrix} 5 & -3 & & & & \\ 1 & 4 & 3 & & & \\ & 9 & -3 & 6 & & \\ & & 2 & 4 & -7 & \\ & & & 3 & -1 & 0 \\ & & & & 6 & -5 & 8 \\ & & & & & 3 & -1 \end{pmatrix}$$

(*a*) Triangular matrix. (*b*) Tridiagonal matrix.

Fig. 4-21

The natural method of representing matrices in memory as two-dimensional arrays may not be suitable for sparse matrices. That is, one may save space by storing only those entries which may be nonzero. This is illustrated for triangular matrices in the following example. Other cases will be discussed in the solved problems.

EXAMPLE 4.25

Suppose we want to place in memory the triangular array A in Fig. 4-22. Clearly it would be wasteful to store those entries above the main diagonal of A, since we know they are all zero; hence we store only the other entries of A in a linear array B as indicated by the arrows. That is, we let

$$B[1] = a_{11}, \qquad B[2] = a_{21}, \qquad B[3] = a_{22}, \qquad B[3] = a_{31}, \qquad \ldots$$

Observe first that B will contain only

$$1 + 2 + 3 + 4 + \cdots + n = \frac{1}{2}\, n(n+1)$$

elements, which is about half as many elements as a two-dimensional $n \times n$ array. Since we will require the value of a_{JK} in our programs, we will want the formula that gives us the integer L in terms of J and K where

$$B[L] = a_{JK}$$

Observe that L represents the number of elements in the list up to and including a_{JK}. Now there are

$$1 + 2 + 3 + \cdots + (J-1) = \frac{J(J-1)}{2}$$

elements in the rows above a_{JK}, and there are K elements in row J up to and including a_{JK}. Accordingly,

$$L = \frac{J(J-1)}{2} + K$$

yields the index that accesses the value a_{JK} from the linear array B.

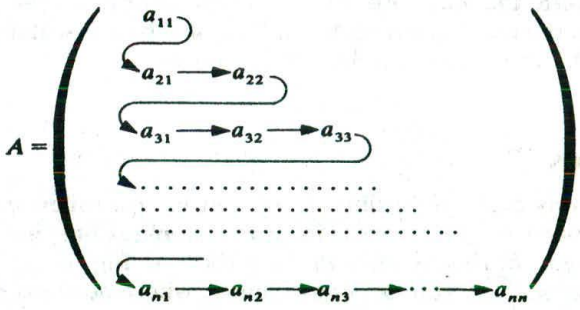


Fig. 4-22

Solved Problems

LINEAR ARRAYS

4.1 Consider the linear arrays AAA(5:50), BBB(−5:10) and CCC(18).

(*a*) Find the number of elements in each array.

(*b*) Suppose *Base*(AAA) = 300 and $w = 4$ words per memory cell for AAA. Find the address of AAA[15], AAA[35] and AAA[55].

(*a*) The number of elements is equal to the length; hence use the formula

$$\text{Length} = \text{UB} - \text{LB} + 1$$

Accordingly,

$$\text{Length(AAA)} = 50 - 5 + 1 = 46$$
$$\text{Length(BBB)} = 10 - (-5) + 1 = 16$$
$$\text{Length(CCC)} = 18 - 1 + 1 = 18$$

Note that Length(CCC) = UB, since LB = 1.

(*b*) Use the formula

$$LOC(AAA[K]) = Base(AAA) + w(K - LB)$$

Hence:

$$LOC(AAA[15]) = 300 + 4(15 - 5) = 340$$
$$LOC(AAA[35]) = 300 + 4(35 - 5) = 420$$

AAA[55] is not an element of AAA, since 55 exceeds UB = 50.

4.2 Suppose a company keeps a linear array YEAR(1920:1970) such that YEAR[K] contains the number of employees born in year K. Write a module for each of the following tasks:

(*a*) To print each of the years in which no employee was born.

(*b*) To find the number NNN of years in which no employee was born.

(*c*) To find the number N50 of employees who will be at least 50 years old at the end of the year. (Assume 1984 is the current year.)

(*d*) To find the number NL of employees who will be at least L years old at the end of the year. (Assume 1984 is the current year.)

Each module traverses the array.

(*a*) 1. Repeat for K = 1920 to 1970:
If YEAR[K] = 0, then: Write: K.
[End of loop.]
2. Return.

(*b*) 1. Set NNN := 0.
2. Repeat for K = 1920 to 1970:
If YEAR[K] = 0, then: Set NNN := NNN + 1.
[End of loop.]
3. Return.

(*c*) We want the number of employees born in 1934 or earlier.

1. Set N50 := 0.
2. Repeat for K = 1920 to 1934:
Set N50 := N50 + YEAR[K].
[End of loop.]
3. Return.

(*d*) We want the number of employees born in year 1984 − L or earlier.

1. Set NL := 0 and LLL := 1984 − L.
2. Repeat for K = 1920 to LLL:
Set NL := NL + YEAR[K].
[End of loop.]
3. Return.

4.3 Suppose a 10-element array A contains the values $a_1, a_2, \ldots, a_{10}$. Find the values in A after each loop.

(*a*) Repeat for K = 1 to 9:
Set A[K + 1] := A[K].
[End of loop.]

(*b*) Repeat for K = 9 to 1 by −1:
Set A[K + 1] := A[9].
[End of loop.]

Note that the index K runs from 1 to 9 in part (*a*) but in reverse order from 9 back to 1 in part (*b*).

(*a*) First A[2] := A[1] sets A[2] = a_1, the value of A[1].
Then A[3] := A[2] sets A[3] = a_1, the current value of A[2].
Then A[4] := A[3] sets A[4] = a_1, the current value of A[3]. And so on.
Thus every element of A will have the value x_1, the original value of A[1].

(*b*) First A[10] := A[9] sets A[10] = a_9.
Then A[9] := A[8] sets A[9] = a_8.
Then A[8] := A[7] sets A[8] = a_7. And so on.
Thus every value in A will move to the next location. At the end of the loop, we still have A[1] = x_1.

Remark: This example illustrates the reason that, in the insertion algorithm, Algorithm 4.4, the elements are moved downward in reverse order, as in loop (*b*) above.

4.4 Consider the alphabetized linear array NAME in Fig. 4-23.

NAME

1	Allen
2	Clark
3	Dickens
4	Edwards
5	Goodman
6	Hobbs
7	Irwin
8	Klein
9	Lewis
10	Morgan
11	Richards
12	Scott
13	Tucker
14	Walton

Fig. 4-23

(*a*) Find the number of elements that must be moved if Brown, Johnson and Peters are inserted into NAME at three different times.

(*b*) How many elements are moved if the three names are inserted at the same time?

(*c*) How does the telephone company handle insertions in a telephone directory?

(*a*) Inserting Brown requires 13 elements to be moved, inserting Johnson requires 7 elements to be moved and inserting Peters requires 4 elements to be moved. Hence 24 elements are moved.

(*b*) If the elements are inserted at the same time, then 13 elements need be moved, each only once (with the obvious algorithm).

(*c*) The telephone company keeps a running list of new numbers and then updates the telephone directory once a year.

SEARCHING, SORTING

4.5 Consider the alphabetized linear array NAME in Fig. 4-23.

(*a*) Using the linear search algorithm, Algorithm 4.5, how many comparisons C are used to locate Hobbs, Morgan and Fisher?

(*b*) Indicate how the algorithm may be changed for such a sorted array to make an unsuccessful search more efficient. How does this affect part (*a*)?

(*a*) C(Hobbs) = 6, since Hobbs is compared with each name, beginning with Allen, until Hobbs is found in NAME[6].

C(Morgan) = 10, since Morgan appears in NAME[10].

C(Fisher) = 15, since Fisher is initially placed in NAME[15] and then Fisher is compared with every name until it is found in NAME[15]. Hence the search is unsuccessful.

(*b*) Observe that NAME is alphabetized. Accordingly, the linear search can stop after a given name XXX is compared with a name YYY such that XXX < YYY (i.e., such that, alphabetically, XXX comes before YYY). With this algorithm, C(Fisher) = 5, since the search can stop after Fisher is compared with Goodman in NAME[5].

4.6 Suppose the binary search algorithm, Algorithm 4.6, is applied to the array NAME in Fig. 4-23 to find the location of Goodman. Find the ends BEG and END and the middle MID for the test segment in each step of the algorithm.

Recall that MID = INT((BEG + END)/2), where INT means integer value.

Step 1. Here BEG = 1 [Allen] and END = 14 [Walton], so MID = 7 [Irwin].

Step 2. Since Goodman < Irwin, reset END = 6. Hence MID = 3 [Dickens].

Step 3. Since Goodman > Dickens, reset BEG = 4. Hence MID = 5 [Goodman].

We have found the location LOC = 5 of Goodman in the array. Observe that there were $C = 3$ comparisons.

4.7 Modify the binary search algorithm, Algorithm 4.6, so that it becomes a search and insertion algorithm.

There is no change in the first four steps of the algorithm. The algorithm transfers control to Step 5 only when ITEM does not appear in DATA. In such a case, ITEM is inserted before or after DATA[MID] according to whether ITEM < DATA[MID] or ITEM > DATA[MID]. The algorithm follows.

Algorithm P4.7: (Binary Search and Insertion) DATA is a sorted array with N elements, and ITEM is a given item of information. This algorithm finds the location LOC of ITEM in DATA or inserts ITEM in its proper place in DATA.

Steps 1 through 4. Same as in Algorithm 4.6.

5. If ITEM < DATA[MID], then:
   Set LOC := MID.
   Else:
   Set LOC := MID + 1.
   [End of If structure.]
6. Insert ITEM into DATA[LOC] using Algorithm 4.2.
7. Exit.

4.8 Suppose A is a sorted array with 200 elements, and suppose a given element x appears with the same probability in any place in A. Find the worst-case running time $f(n)$ and the average-case running time $g(n)$ to find x in A using the binary search algorithm.

For any value of k, let n_k denote the number of those elements in A that will require k comparisons to be located in A. Then:

k:	1	2	3	4	5	6	7	8
n_k:	1	2	4	8	16	32	64	73

The 73 comes from the fact that $1 + 2 + 4 + \cdots + 64 = 127$ so there are only $200 - 127 = 73$ elements left. The worst-case running time $f(n) = 8$. The average-case running time $g(n)$ is obtained as follows:

$$g(n) = \frac{1}{n}\sum_{k=1}^{8} k \cdot n_k$$

$$= \frac{1 \cdot 1 + 2 \cdot 2 + 3 \cdot 4 + 4 \cdot 8 + 5 \cdot 16 + 6 \cdot 32 + 7 \cdot 64 + 8 \cdot 73}{200}$$

$$= \frac{1353}{200} = 6.765$$

Observe that, for the binary search, the average-case and worst-case running times are approximately equal.

4.9 Using the bubble sort algorithm, Algorithm 4.4, find the number C of comparisons and the number D of interchanges which alphabetize the $n = 6$ letters in PEOPLE.

The sequences of pairs of letters which are compared in each of the $n - 1 = 5$ passes follow: a square indicates that the pair of letters is compared and interchanged, and a circle indicates that the pair of letters is compared but not interchanged.

Pass 1. [P E] O P L E, E [P O] P L E, E O (P P) L E

E O P [P L] E, E O P L [P E] E O P L E P

Pass 2. (E O) P L E P, E (O P) L E P, E O [P L] E P

E O L [P E] P, E O L E P P

Pass 3. (E O) L E P P, E [O L] E P P, E L [O E] P P

E L E O P P

Pass 4. (E L) E O P P, E [L E] O P P, E E L O P P

Pass 5. (E E) L O P P, E E L O P P

Since $n = 6$, the number of comparisons will be $C = 5 + 4 + 3 + 2 + 1 = 15$. The number D of interchanges depends also on the data, as well as on the number n of elements. In this case $D = 9$.

4.10 Prove the following identity, which is used in the analysis of various sorting and searching algorithms:

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

Writing the sum S forward and backward, we obtain:

$$S = 1 + 2 + 3 + \cdots + (n-1) + n$$
$$S = n + (n-1) + (n-2) + \cdots + 2 + 1$$

We find the sum of the two values of S by adding pairs as follows:

$$2S = (n+1) + (n+1) + (n+1) + \cdots + (n+1) + (n+1)$$

There are n such sums, so $2S = n(n+1)$. Dividing by 2 gives us our result.

MULTIDIMENSIONAL ARRAYS; MATRICES

4.11 Suppose multidimensional arrays A and B are declared using

$$A(-2:2, 2:22) \quad \text{and} \quad B(1:8, -5:5, -10:5)$$

(*a*) Find the length of each dimension and the number of elements in A and B.

(*b*) Consider the element B[3, 3, 3] in B. Find the effective indices E_1, E_2, E_3 and the address of the element, assuming $Base(B) = 400$ and there are $w = 4$ words per memory location.

(*a*) The length of a dimension is obtained by:

$$\text{Length} = \text{upper bound} - \text{lower bound} + 1$$

Hence the lengths L_i of the dimensions of A are:

$$L_1 = 2 - (-2) + 1 = 5 \quad \text{and} \quad L_2 = 22 - 2 + 1 = 21$$

Accordingly, A has $5 \cdot 21 = 105$ elements. The lengths L_i of the dimensions of B are:

$$L_1 = 8 - 1 + 1 = 8 \qquad L_2 = 5 - (-5) + 1 = 11 \qquad L_3 = 5 - (-10) + 1 = 16$$

Therefore, B has $8 \cdot 11 \cdot 16 = 1408$ elements.

(*b*) The effective index E_i is obtained from $E_i = k_i - \text{LB}$, where k_i is the given index and LB is the lower bound. Hence

$$E_1 = 3 - 1 = 2 \qquad E_2 = 3 - (-5) = 8 \qquad E_3 = 3 - (-10) = 13$$

The address depends on whether the programming language stores B in row-major order or column-major order. Assuming B is stored in column-major order, we use Eq. (4.8):

$$E_3L_2 = 13 \cdot 11 = 143 \qquad E_3L_2 + E_2 = 143 + 8 = 151$$
$$(E_3L_2 + E_2)L_1 = 151 \cdot 8 = 1208 \qquad (E_3L_2 + E_2)L_1 + E_1 = 1208 + 2 = 1210$$

Therefore, $$\text{LOC}(B[3, 3, 3]) = 400 + 4(1210) = 400 + 4840 = 5240$$

4.12 Let A be an $n \times n$ square matrix array. Write a module which

(*a*) Finds the number NUM of nonzero elements in A

(*b*) Finds the SUM of the elements above the diagonal, i.e., elements A[I, J] where I < J

(*c*) Finds the product PROD of the diagonal elements $(a_{11}, a_{22}, \ldots, a_{nn})$

(*a*)
1. Set NUM := 0.
2. Repeat for I = 1 to N:
3. Repeat for J = 1 to N:
If A[I, J] ≠ 0, then: Set NUM := NUM + 1.
[End of inner loop.]
[End of outer loop.]
4. Return.

(*b*)
1. Set SUM := 0.
2. Repeat for J = 2 to N:
3. Repeat for I = 1 to J − 1:
Set SUM := SUM + A[I, J].
[End of inner Step 3 loop.]
4. Return.

(*c*)
1. Set PROD := 1. [This is analogous to setting SUM = 0.]
2. Repeat for K = 1 to N:
Set PROD := PROD * A[K, K].
[End of loop.]
3. Return.

4.13 Consider an n-square tridiagonal array A as shown in Fig. 4-24. Note that A has n elements on the diagonal and $n-1$ elements above and $n-1$ elements below the diagonal. Hence A contains at most $3n-2$ nonzero elements. Suppose we want to store A in a linear array B as indicated by the arrows in Fig. 4-24; i.e.,

$$B[1] = a_{11}, \quad B[2] = a_{12}, \quad B[3] = a_{21}, \quad B[4] = a_{22}, \quad \ldots$$

Find the formula that will give us L in terms of J and K such that

$$B[L] = A[J, K]$$

(so that one can access the value of A[J, K] from the array B).

Note that there are $3(J-2)+2$ elements above A[J, K] and $K-J+1$ elements to the left of A[J, K]. Hence

$$L = [3(J-2)+2] + [K-J+1] + 1 = 2J + K - 2$$

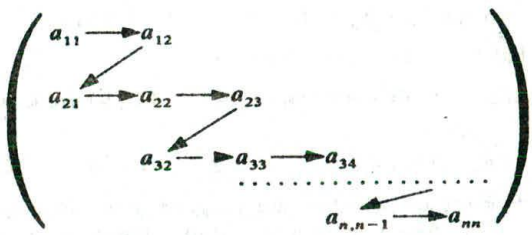


Fig. 4-24 Tridiagonal array.

4.14 **An n-square matrix array A is said to be *symmetric* if A[J, K] = A[K, J] for all J and K.**

(*a*) Which of the following matrices are symmetric?

$$\begin{pmatrix} 2 & -3 & 5 \\ -3 & -2 & 4 \\ 5 & 6 & 8 \end{pmatrix} \quad \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 3 & -7 \\ 3 & 6 & -1 \\ -7 & -1 & 2 \end{pmatrix}$$

(*b*) Describe an efficient way of storing a symmetric matrix A in memory.

(*c*) Suppose A and B are two n-square symmetric matrices. Describe an efficient way of storing A and B in memory.

(*a*) The first matrix is not symmetric, since $a_{23} = 4$ but $a_{32} = 6$. The second matrix is not a square matrix so it cannot be symmetric, by definition. The third matrix is symmetric.

(*b*) Since A[J, K] = A[K, J], we need only store those elements of A which lie on or below the diagonal. This can be done in the same way as that for triangular matrices described in Example 4.25.

(*c*) First note that, for a symmetric matrix, we need store only either those elements on or below the diagonal or those on or above the diagonal. Therefore, A and B can be stored in an $n \times (n+1)$ array C as pictured in Fig. 4-25, where C[J, K] = A[J, K] when $J \geq K$ but C[J, K] = B[J, K − 1] when $J < K$.

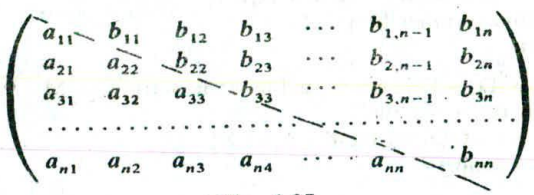


Fig. 4-25

POINTER ARRAYS; RECORD STRUCTURES

4.15 Three lawyers, Davis, Levine and Nelson, share the same office. Each lawyer has his own clients. Figure 4-26 shows three ways of organizing the data.

(*a*) Here there is an alphabetized array CLIENT and an array LAWYER such that LAWYER[K] is the lawyer for CLIENT[K].

(*b*) Here there are three separate arrays, DAVIS, LEVINE and NELSON, each array containing the list of the lawyer's clients.

(*c*) Here there is a LAWYER array, and arrays NUMB and PTR giving, respectively, the number and location of each lawyer's alphabetized list of clients in an array CLIENT.

Which data structure is most useful? Why?

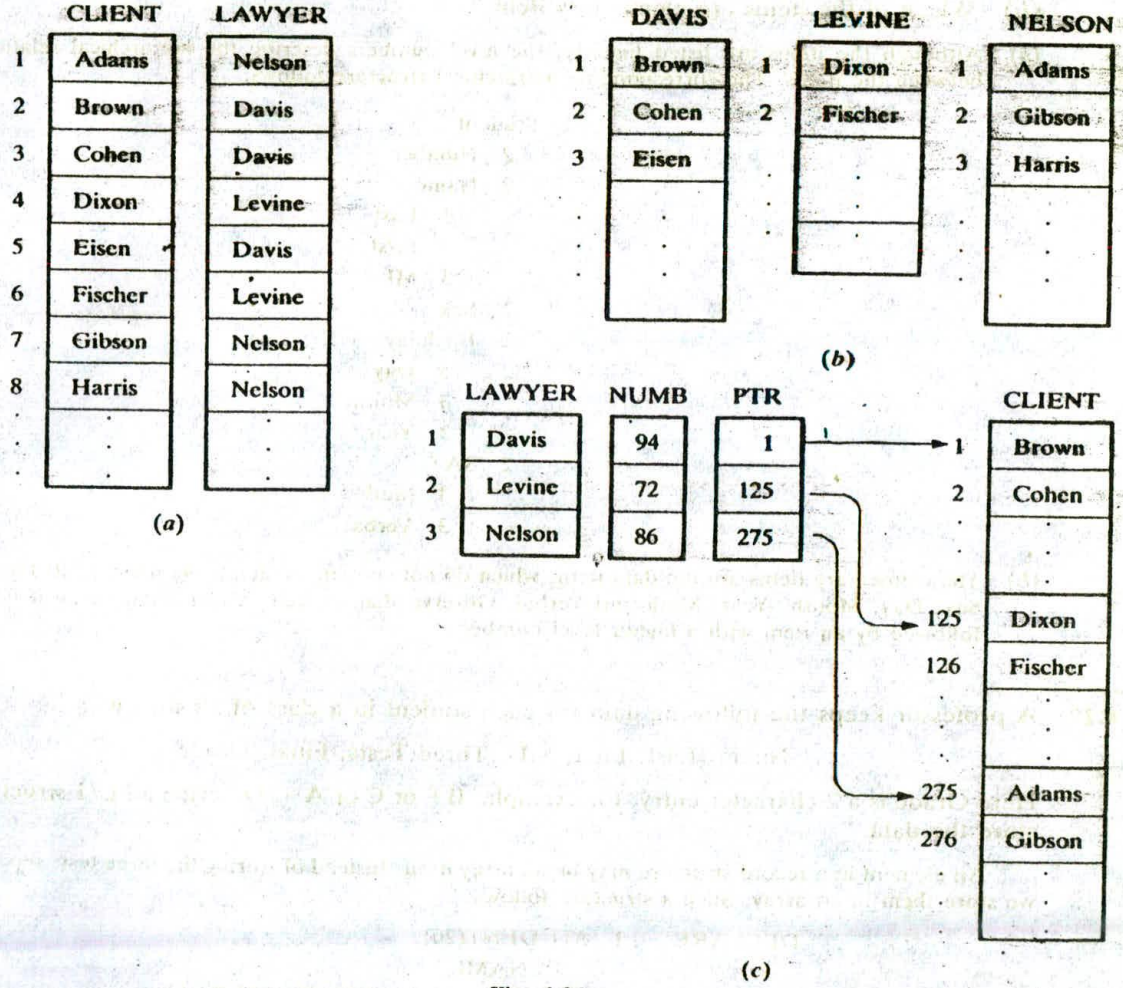


Fig. 4-26

The most useful data structure depends on how the office is organized and how the clients are processed.

Suppose there are only one secretary and one telephone number, and suppose there is a single monthly billing of the clients. Also, suppose clients frequently change from one lawyer to another. Then Fig. 4-26(*a*) would probably be the most useful data structure.

Suppose the lawyers operate completely independently: each lawyer has his own secretary and his own telephone number and bills his clients differently. Then Fig. 4-26(*b*) would likely be the most useful data structure.

Suppose the office processes all the clients frequently and each lawyer has to process his own clients frequently. Then Fig. 4-26(*c*) would likely be the most useful data structure.

4.16 The following is a list of entries, with level numbers, in a student's record:

1 Student 2 Number 2 Name 3 Last 3 First 3 MI (Middle Initial) 2 Sex 2 Birthday 3 Day 3 Month 3 Year 2 SAT 3 Math 3 Verbal

(*a*) Draw the corresponding hierarchical structure.

(*b*) Which of the items are elementary items?

(*a*) Although the items are listed linearly, the level numbers describe the hierarchical relationship between the items. The corresponding hierarchical structure follows:

```
1 Student
    2 Number
    2 Name
        3 Last
        3 First
        3 MI
    2 Sex
    2 Birthday
        3 Day
        3 Month
        3 Year
    2 SAT
        3 Math
        3 Verbal
```

(*b*) The elementary items are the data items which do not contain subitems: Number, Last, First, MI, Sex, Day, Month, Year, Math and Verbal. Observe that an item is elementary only if it is not followed by an item with a higher level number.

4.17 A professor keeps the following data for each student in a class of 20 students:

Name (Last, First, MI), Three Tests, Final, Grade

Here Grade is a 2-character entry, for example, B+ or C or A−. Describe a PL/1 structure to store the data.

An element in a record structure may be an array itself. Instead of storing the three tests separately, we store them in an array. Such a structure follows:

```
DECLARE  1 STUDENT(20),
           2 NAME,
             3 LAST   CHARACTER(10),
             3 FIRST  CHARACTER(10),
             3 MI     CHARACTER(1),
           2 TEST(3)  FIXED,
           2 FINAL    FIXED,
           2 GRADE    CHARACTER(2);
```

4.18 A college uses the following structure for a graduating class:

```
1 Student(200)
    2 Name
        3 Last
        3 First
        3 Middle Initial
    2 Major
    2 SAT
        3 Verbal
        3 Math
    2 GPA(4)
    2 CUM
```

Here, GPA[K] refers to the grade point average during the *k*th year and CUM refers to the cumulative grade point average.

(*a*) How many elementary items are there in the file?

(*b*) How does one access (i) the major of the eighth student and (ii) the sophomore GPA of the forty-fifth student?

(*c*) Find each output:
(i) Write: Name[15]
(ii) Write: CUM
(iii) Write: GPA[2].
(iv) Write: GPA[1, 3].

(*a*) Since GPA is counted 4 times per student, there are 11 elementary items per student, so there are altogether 2200 elementary items.

(*b*) (i) Student.Major[8] or simply MAJOR[8]. (ii) GPA[45, 2].

(*c*) (i) Here Name[15] refers to the name of the fifteenth student. But Name is a group item. Hence LAST[15], First[15] and MI[15] are printed.

(ii) Here CUM refers to all the CUM values. That is,

CUM[1], CUM[2], CUM[3], . . . , CUM[200]

are printed.

(iii) GPA[2] refers to the GPA array of the second student. Hence,

GPA[2, 1], GPA[2, 2], GPA[2, 3], GPA[2, 4]

are printed.

(iv) GPA[1, 3] is a single item, the GPA during the junior year of the first student. That is, only GPA[1, 3] is printed.

4.19 An automobile dealership keeps track of the serial number and price of each of its automobiles in arrays AUTO and PRICE, respectively. In addition, it uses the data structure in Fig. 4-27, which combines a record structure with pointer variables. The new Chevys, new Buicks, new Oldsmobiles, and used cars are listed together in AUTO. The variables NUMB and PTR under USED give, respectively, the number and location of the list of used automobiles.

(*a*) How does one index the location of the list of new Buicks in AUTO?

(*b*) Write a procedure to print serial numbers of all new Buicks under $10 000.

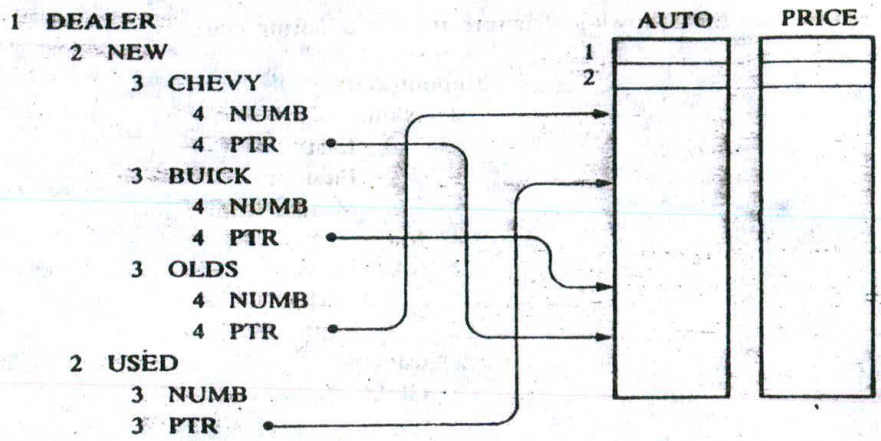


Fig. 4-27

(*a*) Since PTR appears more than once in the record structure, one must use BUICK.PTR to reference the location of the list of new Buicks in AUTO.

(*b*) One must traverse the list of new Buicks but print out only those Buicks whose price is less than \$10 000. The procedure follows:

Procedure P4.19: The data are stored in the structure in Fig. 4-27. This procedure outputs those new Buicks whose price is less than \$10 000.

1. Set FIRST := BUICK.PTR. [Location of first element in Buick list.]
2. Set LAST := FIRST + BUICK.NUMB − 1. [Location of last element in list.]
3. Repeat for K = FIRST to LAST.
   If PRICE[K] < 10 000, then:
   Write: AUTO[K], PRICE[K].
   [End of If structure.]
   [End of loop.]
4. Exit.

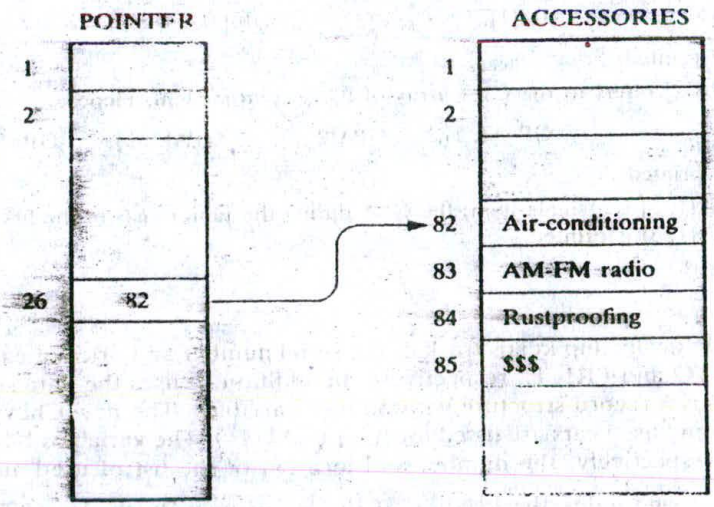


Fig. 4-28

4.20 **Suppose in Prob. 4.19 the dealership had also wanted to keep track of the accessories of each automobile, such as air-conditioning, radio, and rustproofing. Since this involves variable-length data, how might this be done?**

This can be accomplished as in Fig. 4-28. That is, besides AUTO and PRICE, there is an array POINTER such that POINTER[K] gives the location in an array ACCESSORIES of the list of accessories (with sentinel '$$$') of AUTO[K].

Supplementary Problems

ARRAYS

4.21 Consider the linear arrays XXX(−10:10), YYY(1935:1985), ZZZ(35). (*a*) Find the number of elements in each array. (*b*) Suppose *Base*(YYY) = 400 and $w = 4$ words per memory cell for YYY. Find the address of YYY[1942], YYY[1977] and YYY[1988].

4.22 Consider the following multidimensional arrays:

X(−5:5, 3:33) Y(3:10, 1:15, 10:20)

(*a*) Find the length of each dimension and the number of elements in X and Y.

(*b*) Suppose *Base*(Y) = 400 and there are $w = 4$ words per memory location. Find the effective indices E_1, E_2, E_3 and the address of Y[5, 10, 15] assuming (i) Y is stored in row-major order and (ii) Y is stored in column-major order.

4.23 An array A contains 25 positive integers. Write a module which

(*a*) Finds all pairs of elements whose sum is 25

(*b*) Finds the number EVNUM of elements of A which are even, and the number ODNUM of elements of A which are odd

4.24 Suppose A is a linear array with n numeric values. Write a procedure

MEAN(A, N, AVE)

which finds the average AVE of the values in A. The *arithmetic mean* or *average* $\bar{x}$ of the values $x_1, x_2, \ldots, x_n$ is defined by

$$\bar{x} = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

4.25 Each student in a class of 30 students takes 6 tests in which scores range between 0 and 100. Suppose the test scores are stored in a 30×6 array TEST. Write a module which

(*a*) Finds the average grade for each test

(*b*) Finds the final grade for each student where the final grade is the average of the student's five highest test scores

(*c*) Finds the number NUM of students who have failed, i.e., whose final grade is less than 60

(*d*) Finds the average of the final grades

POINTER ARRAYS; RECORD STRUCTURES

4.26 Consider the data in Fig. 4-26(*c*). (*a*) Write a procedure which prints the list of clients belonging to LAWYER[K]. (*b*) Assuming CLIENT has space for 400 elements, define an array FREE such that FREE[K] contains the number of empty cells following the list of clients belonging to LAWYER[K].

4.27 The following is a list of entries, with level numbers, in a file of employee records:

1 Employee(200), 2 SSN(Social Security Number), 2 Name,
3 Last, 3 First, 3 MI (Middle Initial), 2 Address, 3 Street,
3 Area, 4 City, 4 State, 4 ZIP, 2 Age, 2 Salary, 2 Dependents

(*a*) Draw the corresponding hierarchical structure.

(*b*) Which of the items are elementary items?

(*c*) Describe a record structure—for example, a PL/1 structure or a Pascal record—to store the data.

4.28 Consider the data structure in Fig. 4-27. Write a procedure to carry out each of the following:

(*a*) Finding the number of new Oldsmobiles selling for under $10 000.

(*b*) Finding the number of new automobiles selling for under $10 000.

(*c*) Finding the number of automobiles selling for under $10 000.

(*d*) Listing all automobiles selling for under $10 000.

(*Note*: Parts (*c*) and (*d*) require only the arrays AUTO and PRICE together with the number of automobiles.)

4.29 A class of student records is organized as follows:

1 Student(35), 2 Name, 3 Last, 3 First, 3 MI (Middle Initial), 2 Major
2 Test(4), 2 Final, 2 Grade

(*a*) How many elementary items are there?

(*b*) Describe a record structure—for example, a PL/1 structure or a Pascal record, to store the data.

(*c*) Describe the output of each of the following Write statements: (i) Write: Final[15], (ii) Write: Name[15] and (iii) Write: Test[4].

4.30 Consider the data structure in Prob. 4.18. Write a procedure which

(*a*) Finds the average of the sophomore GPA scores

(*b*) Finds the number of biology majors

(*c*) Finds the number of CUM scores exceeding K

Programming Problems

ARRAYS

Assume that the data in Table 4-1 are stored in linear arrays SSN, LAST, GIVEN, CUM and YEAR (with space for 25 students) and that a variable NUM is defined which contains the actual number of students.

4.31 Write a program for each of the following:

(*a*) Listing all students whose CUM is K or higher. (Test the program using K = 3.00.)

(*b*) Listing all students in year L. (Test the program using L = 2, or sophomore.)

4.32 Translate the linear search algorithm into a subprogram LINEAR(ARRAY, LB, UB, ITEM, LOC) which either finds the location LOC where ITEM appears in ARRAY or returns LOC = 0.

4.33 Translate the binary search and insertion algorithm into a subprogram BINARY(ARRAY, LB, UB, ITEM, LOC) which finds either the location LOC where ITEM appears in ARRAY or the location LOC where ITEM should be inserted into ARRAY.

Table 4-1

Social Security Number	Last Name	Given Name	CUM	Year
211-58-1329	Adams	Bruce	2.55	2
169-38-4248	Bailey	Irene L.	3.25	4
166-48-5842	Cheng	Kim	3.40	1
187-52-4076	Davis	John C.	2.85	2
126-63-6382	Edwards	Steven	1.75	3
135-58-9565	Fox	Kenneth	2.80	2
172-48-1849	Green	Gerald S.	2.35	2
192-60-3157	Hopkins	Gary	2.70	2
160-60-1826	Klein	Deborah M.	3.05	1
166-52-4147	Lee	John	2.60	3
186-58-0430	Murphy	William	2.30	2
187-58-1123	Newman	Ronald P.	3.90	4
174-58-0732	Osborn	Paul	2.05	3
183-52-3865	Parker	David	1.55	2
135-48-1397	Rogers	Mary J.	1.85	1
182-52-6712	Schwab	Joanna	2.95	2
184-48-8539	Thompson	David E.	3.15	3
187-48-2377	White	Adam	2.50	2

4.34 Write a program which reads the social security number SOC of a student and uses LINEAR to find and print the student's record. Test the program using (*a*) 174-58-0732, (*b*) 172-55-5554 and (*c*) 126-63-6382.

4.35 Write a program which reads the (last) NAME of a student and uses BINARY to find and print the student's record. Test the program using (*a*) Rogers, (*b*) Johnson and (*c*) Bailey.

4.36 Write a program which reads the record of a student

SSNST, LASTST, GVNST, CUMST, YEARST

and uses BINARY to insert the record into the list. Test the program using:

(*a*) 168-48-2255, Quinn, Michael, 2.15, 3

(*b*) 177-58-0772, Jones, Amy, 2.75, 2

4.37 Write a program which reads the (last) NAME of a student and uses BINARY to delete the student's record from the list. Test the program using (*a*) Parker and (*b*) Fox.

4.38 Write a program for each of the following:

(*a*) Using the array SSN to define arrays NUMBER and PTR such that NUMBER is a sorted array of the elements in SSN and PTR[K] contains the location of NUMBER[K] in SSN.

(*b*) Reading the social security number SOC of a student and using BINAR and the array NUMBER to find and print the student's record. Test the program using (i) 174-58-0732, (ii) 172-55-5554 and (iii) 126-63-6382. (Compare with Prob. 4.34.)

POINTER ARRAYS

Assume the data in Table 4-2 are stored in a single linear array CLASS (with space for 50 names). Also assume that there are 2 empty cells between the sections, and that there are linear arrays NUMB, PTR and FREE defined so that NUMB[K] contains the number of elements in Section K, PTR[K] gives the location in CLASS of the first name in Section K, and FREE[K] gives the number of empty cells in CLASS following Section K.

Table 4-2

Section 1	Section 2	Section 3	Section 4
Brown	Abrams	Allen	Burns
Davis	Collins	Conroy	Cohen
Jones	Forman	Damario	Evans
Samuels	Hughes	Harris	Gilbert
	Klein	Rich	Harlan
	Lee	Sweeney	Lopez
	Moore		Meth
	Quinn		Ryan
	Rosen		Williams
	Scott		
	Taylor		
	Weaver		

4.39 Write a program which reads an integer K and prints the names in Section K. Test the program using (*a*) K = 2 and (*b*) K = 3.

4.40 Write a program which reads the NAME of a student and finds and prints the location and section number of the student. Test the program using (*a*) Harris, (*b*) Rivers and (*c*) Lopez.

4.41 Write a program which prints the names in columns as they appear in Table 4-2.

4.42 Write a program which reads the NAME and section number SECN of a student and inserts the student into CLASS. Test the program using (*a*) Eden, 3; (*b*) Novak, 4; (*c*) Parker, 2; (*d*) Vaughn, 3; and (*e*) Bennett, 3. (The program should handle OVERFLOW.)

4.43 Write a program which reads the NAME of a student and deletes the student from CLASS. Test the program using (*a*) Klein, (*b*) Daniels, (*c*) Meth and (*d*) Harris.

MISCELLANEOUS

4.44 Suppose A and B are n-element vector arrays in memory and X and Y are scalars. Write a program to find (*a*) XA + YB and (*b*) A · B. Test the program using A = (16, −6, 7), B = (4, 2, −3), X = 2 and Y = −5.

4.45 Translate the matrix multiplication algorithm, Algorithm 4.7, into a subprogram

MATMUL(A, B, C, M, P, N)

which finds the product C of an $m \times p$ matrix A and a $p \times n$ matrix B. Test the program using

$$A = \begin{pmatrix} 4 & -3 & 5 \\ 6 & 1 & -2 \end{pmatrix} \qquad B = \begin{pmatrix} 2 & 3 & -7 & -3 \\ 5 & -1 & 6 & 2 \\ 0 & 3 & -2 & 1 \end{pmatrix}$$

4.46 Consider the polynomial

$$f(x) = a_1x^n + a_2x^{n-1} + \cdots + a_nx + a_{n+1}$$

Evaluating the polynomial in the obvious way would require

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

multiplications and n additions. However, one can rewrite the polynomial by successively factoring out x as follows:

$$f(x) = ((\cdots((a_1x + a_2)x + a_3)x + \cdots)x + a_n)x + a_{n+1}$$

This uses only n multiplications and n additions. This second way of evaluating a polynomial is called Horner's method.

(*a*) Rewrite the polynomial $f(x) = 5x^4 - 6x^3 + 7x^2 + 8x - 9$ as it would be evaluated using Horner's method.

(*b*) Suppose the coefficients of a polynomial are in memory in a linear array A(N + 1). (That is, A[1] is the coefficient of x^n, A[2] is the coefficient of x^{n-1}, . . . , and A[N + 1] is the constant.) Write a procedure HORNER(A, N + 1, X, Y) which finds the value Y = F(X) for a given value X using Horner's method.

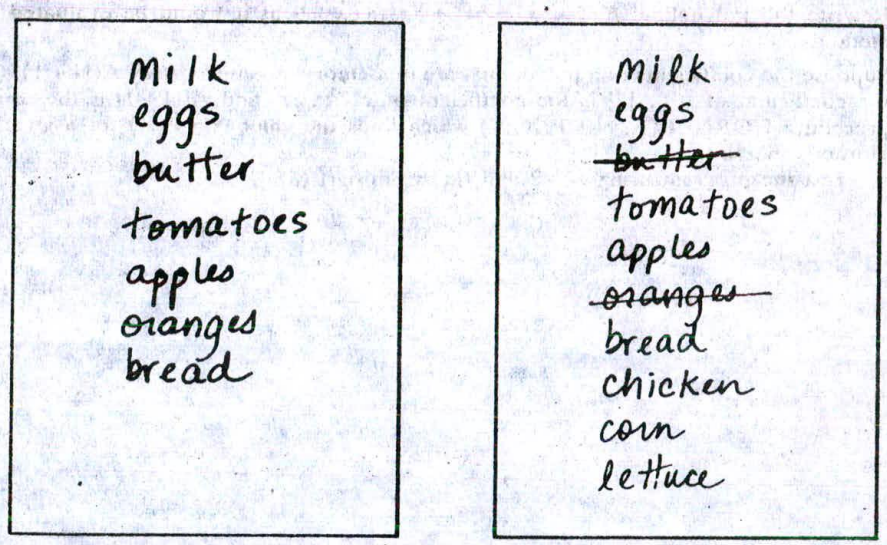
Test the program using X = 2 and $f(x)$ from part (*a*).

Chapter 5

Linked Lists

5.1 INTRODUCTION

The everyday usage of the term "list" refers to a linear collection of data items. Figure 5-1(*a*) shows a shopping list; it contains a first element, a second element, . . . , and a last element. Frequently, we want to add items to or delete items from a list. Figure 5-1(*b*) shows the shopping list after three items have been added at the end of the list and two others have been deleted (by being crossed out).



(*a*) (*b*)

Fig. 5-1

Data processing frequently involves storing and processing data organized into lists. One way to store such data is by means of arrays, discussed in Chap. 4. Recall that the linear relationship between the data elements of an array is reflected by the physical relationship of the data in memory, not by any information contained in the data elements themselves. This makes it easy to compute the address of an element in an array. On the other hand, arrays have certain disadvantages—e.g., it is relatively expensive to insert and delete elements in an array. Also, since an array usually occupies a block of memory space, one cannot simply double or triple the size of an array when additional space is required. (For this reason, arrays are called *dense lists* and are said to be *static* data structures.)

Another way of storing a list in memory is to have each element in the list contain a field, called a *link* or *pointer*, which contains the address of the next element in the list. Thus successive elements in the list need not occupy adjacent space in memory. This will make it easier to insert and delete elements in the list. Accordingly, if one were mainly interested in searching through data for inserting and deleting, as in word processing, one would not store the data in an array but rather in a list using pointers. This latter type of data structure is called a *linked list* and is the main subject matter of this chapter. We also discuss circular lists and two-way lists—which are natural generalizations of linked lists—and their advantages and disadvantages.

5.2 LINKED LISTS

A *linked list*, or *one-way list*, is a linear collection of data elements, called *nodes*, where the linear order is given by means of *pointers*. That is, each node is divided into two parts: the first part contains the information of the element, and the second part, called the *link field* or *nextpointer field*, contains the address of the next node in the list.

Figure 5-2 is a schematic diagram of a linked list with 6 nodes, Each node is pictured with two parts. The left part represents the information part of the node, which may contain an entire record of data items (e.g., NAME, ADDRESS, . . .). The right part represents the nextpointer field of the node, and there is an arrow drawn from it to the next node in the list. This follows the usual practice of drawing an arrow from a field to a node when the address of the node appears in the given field. The pointer of the last node contains a special value, called the *null* pointer, which is any invalid address.

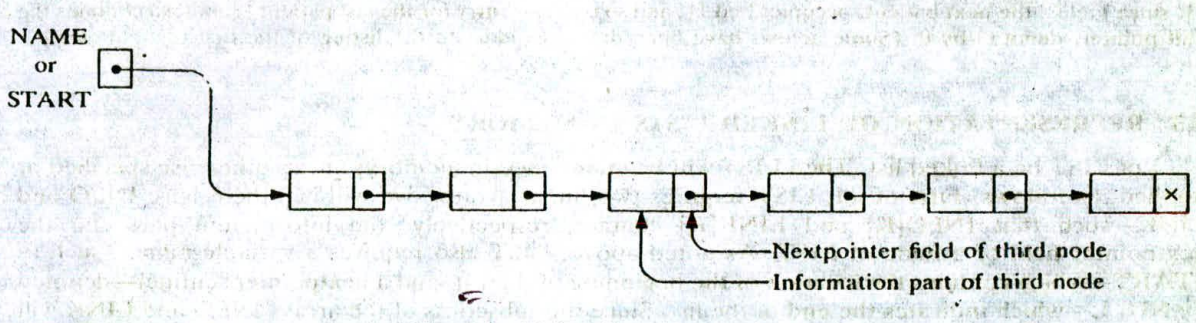


Fig. 5-2 Linked list with 6 nodes.

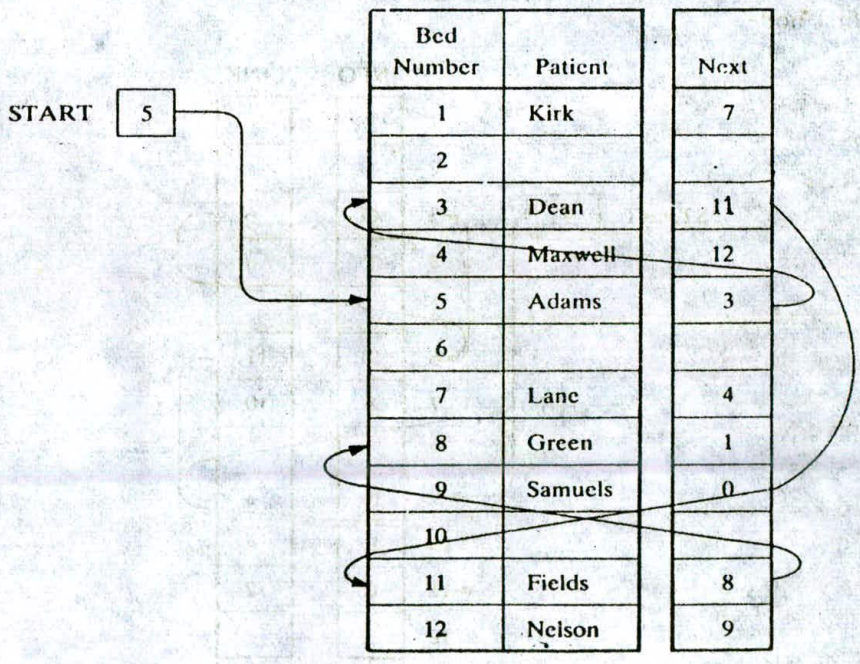


Fig. 5-3

(In actual practice, 0 or a negative number is used for the null pointer.) The null pointer, denoted by × in the diagram, signals the end of the list. The linked list also contains a *list pointer variable*—called START or NAME—which contains the address of the first node in the list; hence there is an arrow drawn from START to the first node. Clearly, we need only this address in START to trace through the list. A special case is the list that has no nodes. Such a list is called the *null list* or *empty list* and is denoted by the null pointer in the variable START.

EXAMPLE 5.1

A hospital ward contains 12 beds, of which 9 are occupied as shown in Fig. 5-3. Suppose we want an alphabetical listing of the patients. This listing may be given by the pointer field, called Next in the figure. We use the variable START to point to the first patient. Hence START contains 5, since the first patient, Adams, occupies bed 5. Also, Adams's pointer is equal to 3, since Dean, the next patient, occupies bed 3; Dean's pointer is 11, since Fields, the next patient, occupies bed 11; and so on. The entry for the last patient (Samuels) contains the null pointer, denoted by 0. (Some arrows have been drawn to indicate the listing of the first few patients.)

5.3 REPRESENTATION OF LINKED LISTS IN MEMORY

Let LIST be a linked list. Then LIST will be maintained in memory, unless otherwise specified or implied, as follows. First of all, LIST requires two linear arrays—we will call them here INFO and LINK—such that INFO[K] and LINK[K] contain, respectively, the information part and the nextpointer field of a node of LIST. As noted above, LIST also requires a variable name—such as START—which contains the location of the beginning of the list, and a nextpointer sentinel—denoted by NULL—which indicates the end of the list. Since the subscripts of the arrays INFO and LINK will usually be positive, we will choose NULL = 0, unless otherwise stated.

The following examples of linked lists indicate that the nodes of a list need not occupy adjacent elements in the arrays INFO and LINK, and that more than one list may be maintained in the same linear arrays INFO and LINK. However, each list must have its own pointer variable giving the location of its first node.

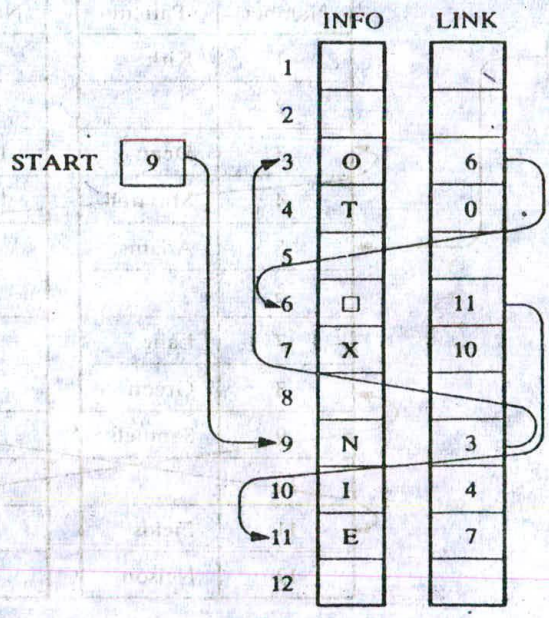


Fig. 5-4

EXAMPLE 5.2

Figure 5-4 pictures a linked list in memory where each node of the list contains a single character. We can obtain the actual list of characters, or, in other words, the string, as follows:

START = 9, so INFO[9] = N is the first character.
LINK[9] = 3, so INFO[3] = O is the second character.
LINK[3] = 6, so INFO[6] = □ (blank) is the third character.
LINK[6] = 11, so INFO[11] = E is the fourth character.
LINK[11] = 7, so INFO[7] = X is the fifth character.
LINK[7] = 10, so INFO[10] = I is the sixth character.
LINK[10] = 4, so INFO[4] = T is the seventh character.
LINK[4] = 0, the NULL value, so the list has ended.

In other words, NO EXIT is the character string.

EXAMPLE 5.3

Figure 5-5 pictures how two lists of test scores, here ALG and GEOM, may be maintained in memory where the nodes of both lists are stored in the same linear arrays TEST and LINK. Observe that the names of the lists are also used as the list pointer variables. Here ALG contains 11, the location of its first node, and GEOM contains 5, the location of its first node. Following the pointers, we see that ALG consists of the test scores

88, 74, 93, 82

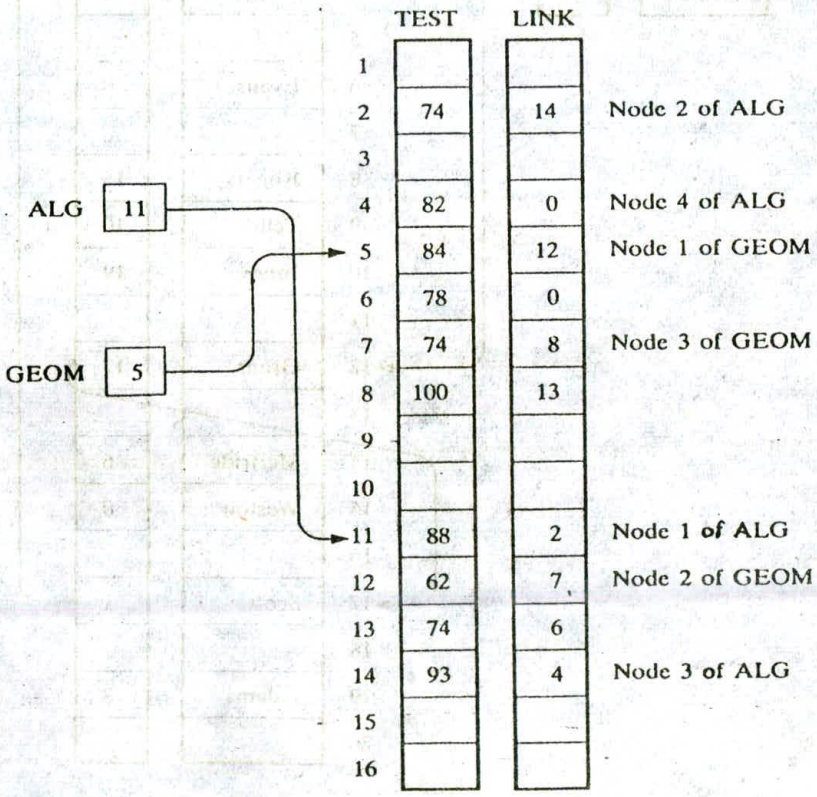


Fig. 5-5

and GEOM consists of the test scores

84, 62, 74, 100, 74, 78

(The nodes of ALG and some of the nodes of GEOM are explicitly labeled in the diagram.)

EXAMPLE 5.4

Suppose a brokerage firm has four brokers and each broker has his own list of customers. Such data may be organized as in Fig. 5-6. That is, all four lists of customers appear in the same array CUSTOMER, and an array LINK contains the nextpointer fields of the nodes of the lists. There is also an array BROKER which contains the list of brokers, and a pointer array POINT such that POINT[K] points to the beginning of the list of customers of BROKER[K].

Accordingly, Bond's list of customers, as indicated by the arrows, consists of

Grant, Scott, Vito, Katz

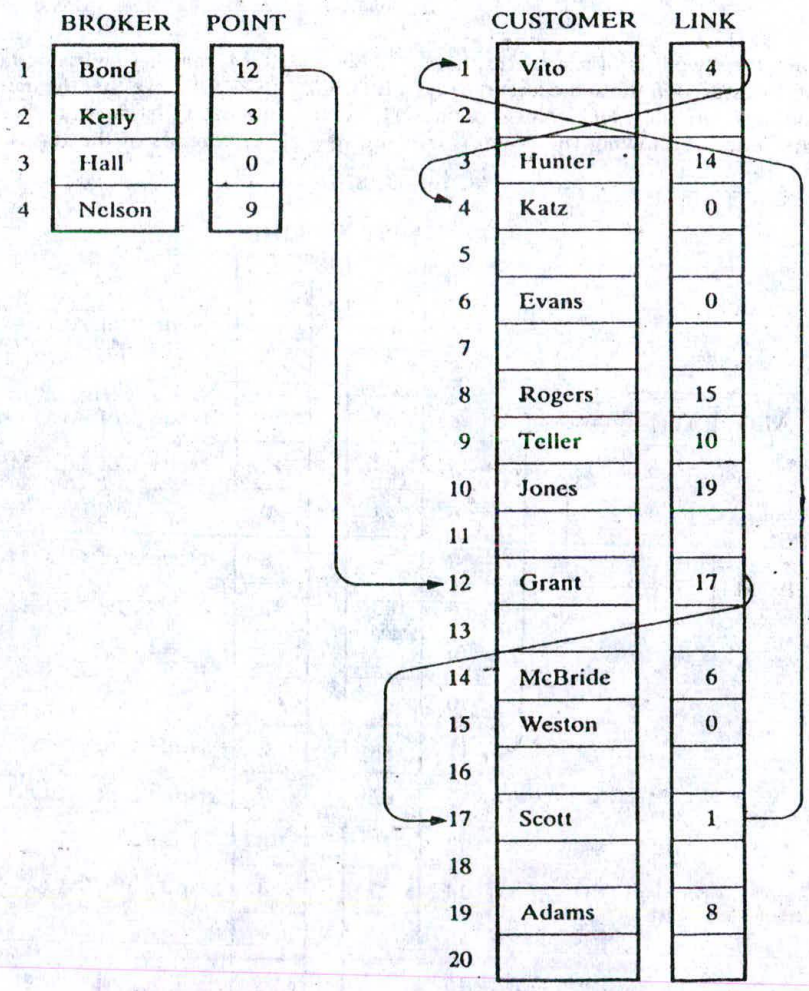


Fig. 5-6

Similarly, Kelly's list consists of

Hunter, McBride, Evans

and Nelson's list consists of

Teller, Jones, Adams, Rogers, Weston

Hall's list is the null list, since the null pointer 0 appears in POINT[3].

Generally speaking, the information part of a node may be a record with more than one data item. In such a case, the data must be stored in some type of record structure or in a collection of parallel arrays, such as that illustrated in the following example.

EXAMPLE 5.5

Suppose the personnel file of a small company contains the following data on its nine employees:

Name, Social Security Number, Sex, Monthly Salary

Normally, four parallel arrays, say NAME, SSN, SEX, SALARY, are required to store the data as discussed in Sec. 4.12. Figure 5-7 shows how the data may be stored as a sorted (alphabetically) linked list using only an additional array LINK for the nextpointer field of the list and the variable START to point to the first record in the list. Observe that 0 is used as the null pointer.

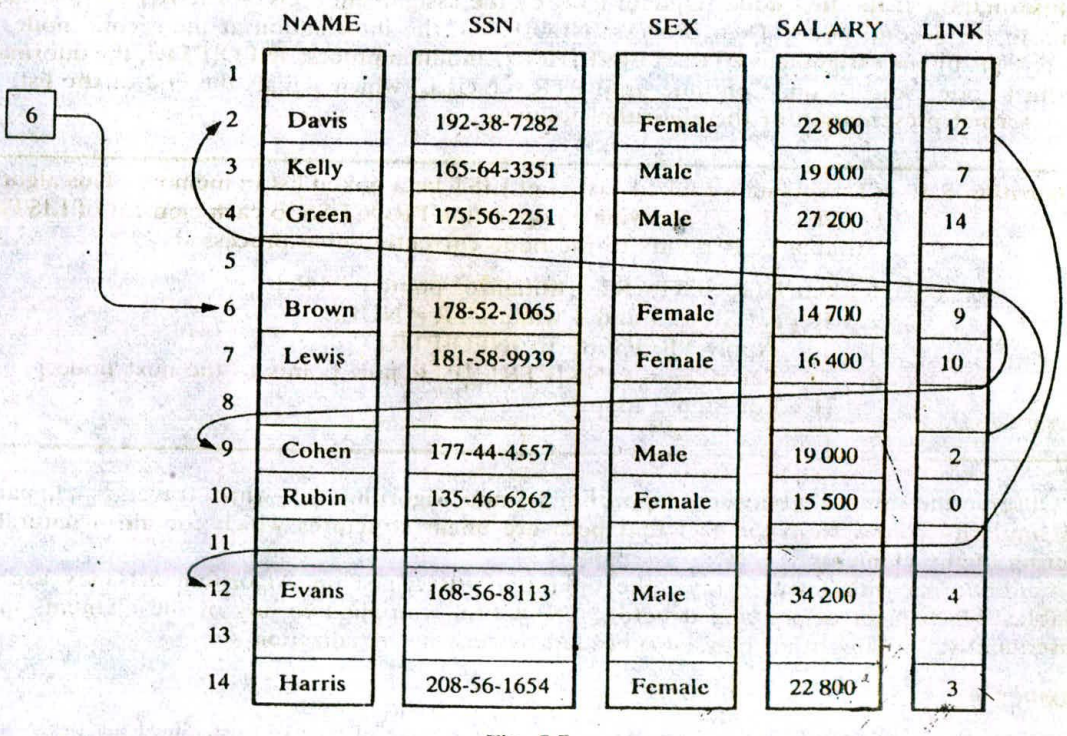


Fig. 5-7

5.4 TRAVERSING A LINKED LIST

Let LIST be a linked list in memory stored in linear arrays INFO and LINK with START pointing to the first element and NULL indicating the end of LIST. Suppose we want to traverse LIST in order to process each node exactly once. This section presents an algorithm that does so and then uses the algorithm in some applications.

Our traversing algorithm uses a pointer variable PTR which points to the node that is currently being processed. Accordingly, LINK[PTR] points to the next node to be processed. Thus the assignment

$$PTR := LINK[PTR]$$

moves the pointer to the next node in the list, as pictured in Fig. 5-8.

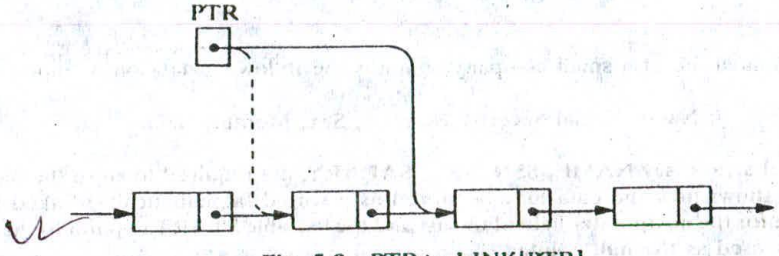


Fig. 5-8 PTR := LINK[PTR].

The details of the algorithm are as follows. Initialize PTR or START. Then process INFO[PTR], the information at the first node. Update PTR by the assignment PTR := LINK[PTR], so that PTR points to the second node. Then process INFO[PTR], the information at the second node. Again update PTR by the assignment PTR := LINK[PTR], and then process INFO[PTR], the information at the third node. And so on. Continue until PTR = NULL, which signals the end of the list.

A formal presentation of the algorithm follows.

Algorithm 5.1: (Traversing a Linked List) Let LIST be a linked list in memory. This algorithm traverses LIST, applying an operation PROCESS to each element of LIST. The variable PTR points to the node currently being processed.

1. Set PTR := START. [Initializes pointer PTR.]
2. Repeat Steps 3 and 4 while PTR ≠ NULL.
3. Apply PROCESS to INFO[PTR].
4. Set PTR := LINK[PTR]. [PTR now points to the next node.]
   [End of Step 2 loop.]
5. Exit.

Observe the similarity between Algorithm 5.1 and Algorithm 4.1, which traverses a linear array. The similarity comes from the fact that both are linear structures which contain a natural linear ordering of the elements.

Caution: As with linear arrays, the operation PROCESS in Algorithm 5.1 may use certain variables which must be initialized before PROCESS is applied to any of the elements in LIST. Consequently, the algorithm may be preceded by such an initialization step.

EXAMPLE 5.6

The following procedure prints the information at each node of a linked list. Since the procedure must traverse the list, it will be very similar to Algorithm 5.1.

Procedure: PRINT(INFO, LINK, START)
This procedure prints the information at each node of the list.

1. Set PTR := START.
2. Repeat Steps 3 and 4 while PTR ≠ NULL:
3. Write: INFO[PTR].
4. Set PTR := LINK[PTR]. [Updates pointer.]
[End of Step 2 loop.]
5. Return.

In other words, the procedure may be obtained by simply substituting the statement

Write: INFO[PTR]

for the processing step in Algorithm 5.1.

EXAMPLE 5.7

The following procedure finds the number NUM of elements in a linked list.

Procedure: COUNT(INFO, LINK, START, NUM)

1. Set NUM := 0. [Initializes counter.]
2. Set PTR := START. [Initializes pointer.]
3. Repeat Steps 4 and 5 while PTR ≠ NULL.
4. Set NUM := NUM + 1. [Increases NUM by 1.]
5. Set PTR := LINK[PTR]. [Updates pointer.]
[End of Step 3 loop.]
6. Return.

Observe that the procedure traverses the linked list in order to count the number of elements; hence the procedure is very similar to the above traversing algorithm, Algorithm 5.1. Here, however, we require an initialization step for the variable NUM before traversing the list. In other words, the procedure could have been written as follows:

Procedure: COUNT(INFO, LINK, START, NUM)

1. Set NUM := 0. [Initializes counter.]
2. Call Algorithm 5.1, replacing the processing step by:
Set NUM := NUM + 1.
3. Return.

Most list processing procedures have this form. (See Prob. 5.3.)

5.5 SEARCHING A LINKED LIST

Let LIST be a linked list in memory, stored as in Secs. 5.3 and 5.4. Suppose a specific ITEM of information is given. This section discusses two searching algorithms for finding the location LOC of the node where ITEM first appears in LIST. The first algorithm does not assume that the data in LIST are sorted, whereas the second algorithm does assume that LIST is sorted.

If ITEM is actually a key value and we are searching through a file for the record containing ITEM, then ITEM can appear only once in LIST.

LIST Is Unsorted

Suppose the data in LIST are not necessarily sorted. Then one searches for ITEM in LIST by traversing through the list using a pointer variable PTR and comparing ITEM with the contents INFO[PTR] of each node, one by one, of LIST. Before we update the pointer PTR by

PTR := LINK[PTR]

we require two tests. First we have to check to see whether we have reached the end of the list; i.e., first we check to see whether

$$PTR = NULL$$

If not, then we check to see whether

$$INFO[PTR] = ITEM$$

The two tests cannot be performed at the same time, since INFO[PTR] is not defined when PTR = NULL. Accordingly, we use the first test to control the execution of a loop, and we let the second test take place inside the loop. The algorithm follows.

Algorithm 5.2 SEARCH(INFO, LINK, START, ITEM, LOC)
LIST is a linked list in memory. This algorithm finds the location LOC of the node where ITEM first appears in LIST, or sets LOC = NULL.

```
1. Set PTR := START.
2. Repeat Step 3 while PTR ≠ NULL:
3.     If ITEM = INFO[PTR], then:
           Set LOC := PTR, and Exit.
       Else:
           Set PTR := LINK[PTR]. [PTR now points to the next node.]
       [End of If structure.]
   [End of Step 2 loop.]
4. [Search is unsuccessful.] Set LOC := NULL.
5. Exit.
```

The complexity of this algorithm is the same as that of the linear search algorithm for linear arrays discussed in Sec. 4.7. That is, the worst-case running time is proportional to the number n of elements in LIST, and the average-case running time is approximately proportional to $n/2$ (with the condition that ITEM appears once in LIST but with equal probability in any node of LIST).

EXAMPLE 5.8

Consider the personnel file in Fig. 5-7. The following module reads the social security number NNN of an employee and then gives the employee a 5 percent increase in salary.

```
1. Read: NNN.
2. Call SEARCH(SSN, LINK, START, NNN, LOC).
3. If LOC ≠ NULL, then:
       Set SALARY[LOC] := SALARY[LOC] + 0.05 * SALARY[LOC],
   Else:
       Write: NNN is not in file.
   [End of If structure.]
4. Return.
```

(The module takes care of the case in which there is an error in inputting the social security number.)

LIST Is Sorted

Suppose the data in LIST are sorted. Again we search for ITEM in LIST by traversing the list using a pointer variable PTR and comparing ITEM with the contents INFO[PTR] of each node, one by one, of LIST. Now, however, we can stop once ITEM exceeds INFO[PTR]. The algorithm follows on page 123.

The complexity of this algorithm is still the same as that of other linear search algorithms; that is, the worst-case running time is proportional to the number n of elements in LIST, and the average-case running time is approximately proportional to $n/2$.

Algorithm 5.3: SRCHSL(INFO, LINK, START, ITEM, LOC)
LIST is a sorted list in memory. This algorithm finds the location LOC of the node where ITEM first appears in LIST, or sets LOC = NULL.

```
1. Set PTR := START.
2. Repeat Step 3 while PTR ≠ NULL:
3.     If ITEM < INFO[PTR], then:
           Set PTR := LINK[PTR]. [PTR now points to next node.]
       Else if ITEM = INFO[PTR], then:
           Set LOC := PTR, and Exit. [Search is successful.]
       Else:
           Set LOC := NULL, and Exit. [ITEM now exceeds INFO[PTR].]
       [End of If structure.]
   [End of Step 2 loop.]
4. Set LOC := NULL.
5. Exit.
```

Recall that with a sorted linear array we can apply a binary search whose running time is proportional to $\log_2 n$. On the other hand, a *binary search algorithm cannot be applied to a sorted linked list, since there is no way of indexing the middle element in the list.* This property is one of the main drawbacks in using a linked list as a data structure.

EXAMPLE 5.9

Consider, again, the personnel file in Fig. 5-7. The following module reads the name EMP of an employee and then gives the employee a 5 percent increase in salary. (Compare with Example 5.8.)

```
1. Read: EMPNAME.
2. Call SRCHSL(NAME, LINK, START, EMPNAME, LOC).
3. If LOC ≠ NULL, then:
     Set SALARY[LOC] := SALARY[LOC] + 0.05 * SALARY[LOC].
   Else:
       Write: EMPNAME is not in list.
   [End of If structure.]
4. Return.
```

Observe that now we can use the second search algorithm, Algorithm 5.3, since the list is sorted alphabetically.

5.6 MEMORY ALLOCATION; GARBAGE COLLECTION

The maintenance of linked lists in memory assumes the possibility of inserting new nodes into the lists and hence requires some mechanism which provides unused memory space for the new nodes. Analogously, some mechanism is required whereby the memory space of deleted nodes becomes available for future use. These matters are discussed in this section, while the general discussion of the inserting and deleting of nodes is postponed until later sections.

Together with the linked lists in memory, a special list is maintained which consists of unused memory cells. This list, which has its own pointer, is called the *list of available space* or the *free-storage list* or the *free pool*.

Suppose our linked lists are implemented by parallel arrays as described in the preceding sections, and suppose insertions and deletions are to be performed on our linked lists. Then the unused memory cells in the arrays will also be linked together to form a linked list using AVAIL as its list pointer variable. (Hence this free-storage list will also be called the AVAIL list.) Such a data structure will frequently be denoted by writing

LIST(INFO, LINK, START, AVAIL)

EXAMPLE 5.10

Suppose the list of patients in Example 5.1 is stored in the linear arrays BED and LINK (so that the patient in bed K is assigned to BED[K]). Then the available space in the linear array BED may be linked as in Fig. 5-9. Observe that BED[10] is the first available bed, BED[2] is the next available bed, and BED[6] is the last available bed. Hence BED[6] has the null pointer in its nextpointer field; that is, LINK[6] = 0.

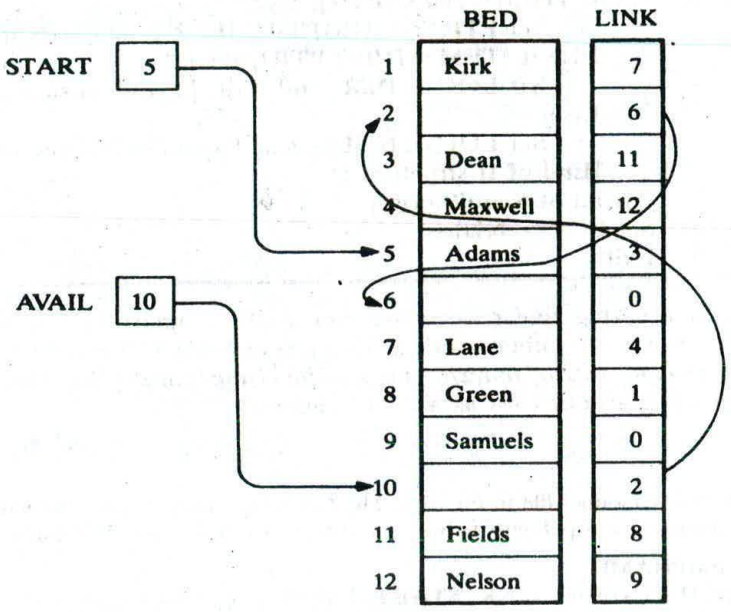


Fig. 5-9

EXAMPLE 5.11

(*a*) The available space in the linear array TEST in Fig. 5-5 may be linked as in Fig. 5-10. Observe that each of the lists ALG and GEOM may use the AVAIL list. Note that AVAIL = 9, so TEST[9] is the first free node in the AVAIL list. Since LINK[AVAIL] = LINK[9] = 10, TEST[10] is the second free node in the AVAIL list. And so on.

(*b*) Consider the personnel file in Fig. 5-7. The available space in the linear array NAME may be linked as in Fig. 5-11. Observe that the free-storage list in NAME consists of NAME[8], NAME[11], NAME[13], NAME[5] and NAME[1]. Moreover, observe that the values in LINK simultaneously list the free-storage space for the linear arrays SSN, SEX and SALARY.

(*c*) The available space in the array CUSTOMER in Fig. 5-6 may be linked as in Fig. 5-12. We emphasize that each of the four lists may use the AVAIL list for a new customer.

EXAMPLE 5.12

Suppose LIST(INFO, LINK, START, AVAIL) has memory space for $n = 10$ nodes. Furthermore, suppose LIST is initially empty. Figure 5-13 shows the values of LINK so that the AVAIL list consists of the sequence

INFO[1], INFO[2], . . . , INFO[10]

that is, so that the AVAIL list consists of the elements of INFO in the usual order. Observe that START = NULL, since the list is empty.

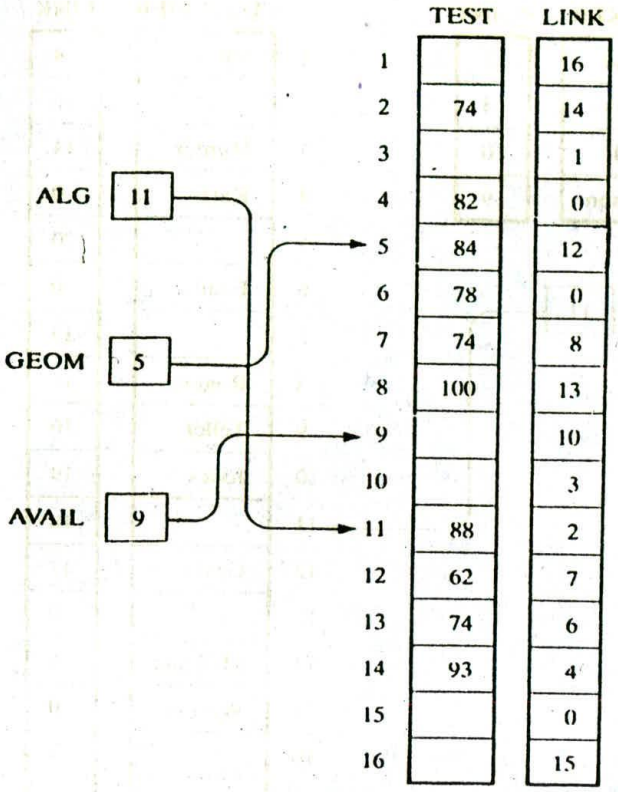


Fig. 5-10

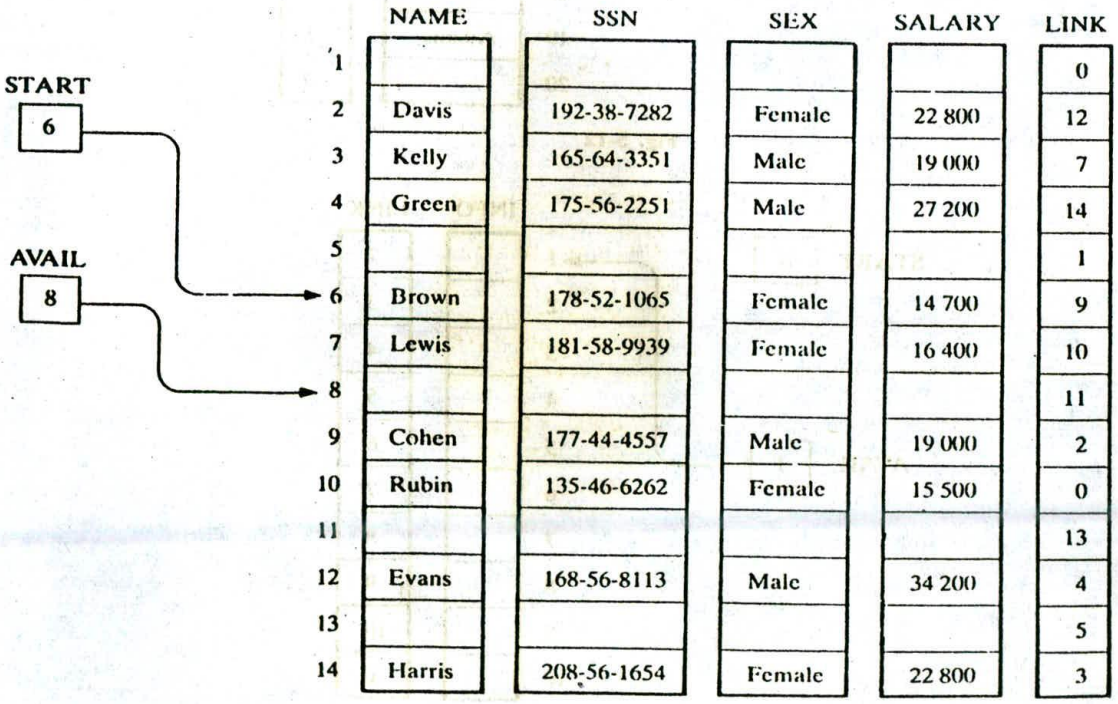


Fig. 5-11

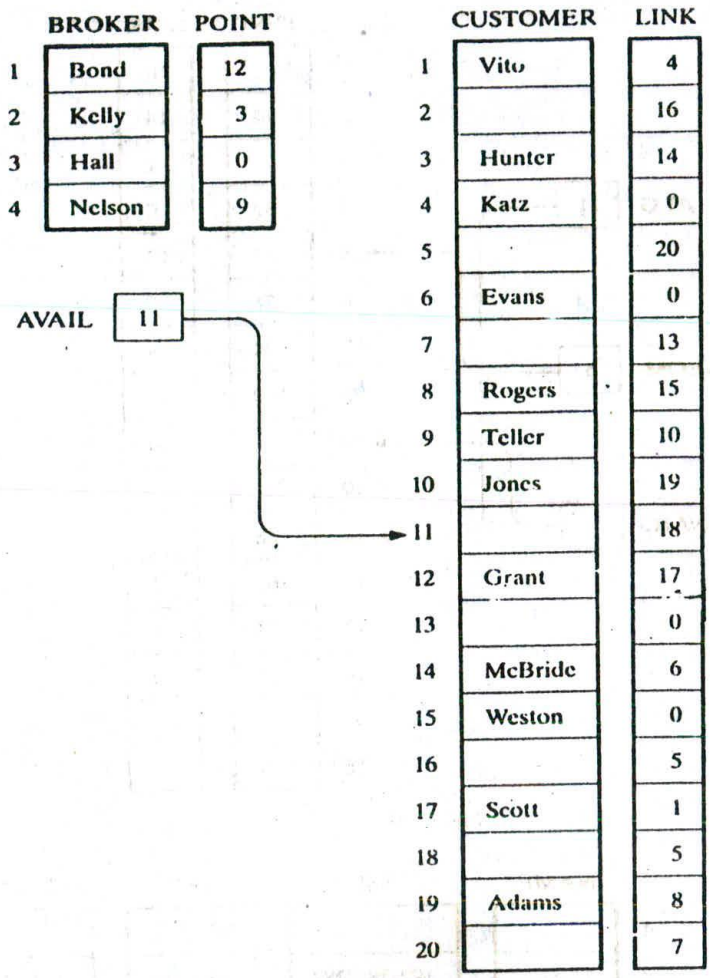


Fig. 5-12

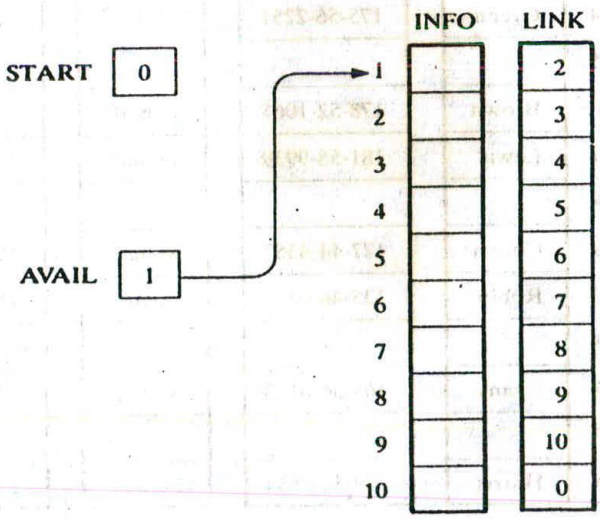


Fig. 5-13

Garbage Collection

Suppose some memory space becomes reusable because a node is deleted from a list or an entire list is deleted from a program. Clearly, we want the space to be available for future use. One way to bring this about is to immediately reinsert the space into the free-storage list. This is what we will do when we implement linked lists by means of linear arrays. However, this method may be too time-consuming for the operating system of a computer, which may choose an alternative method, as follows.

The operating system of a computer may periodically collect all the deleted space onto the free-storage list. Any technique which does this collection is called *garbage collection*. Garbage collection usually takes place in two steps. First the computer runs through all lists, tagging those cells which are currently in use, and then the computer runs through the memory, collecting all untagged space onto the free-storage list. The garbage collection may take place when there is only some minimum amount of space or no space at all left in the free-storage list, or when the CPU is idle and has time to do the collection. Generally speaking, the garbage collection is invisible to the programmer. Any further discussion about this topic of garbage collection lies beyond the scope of this text.

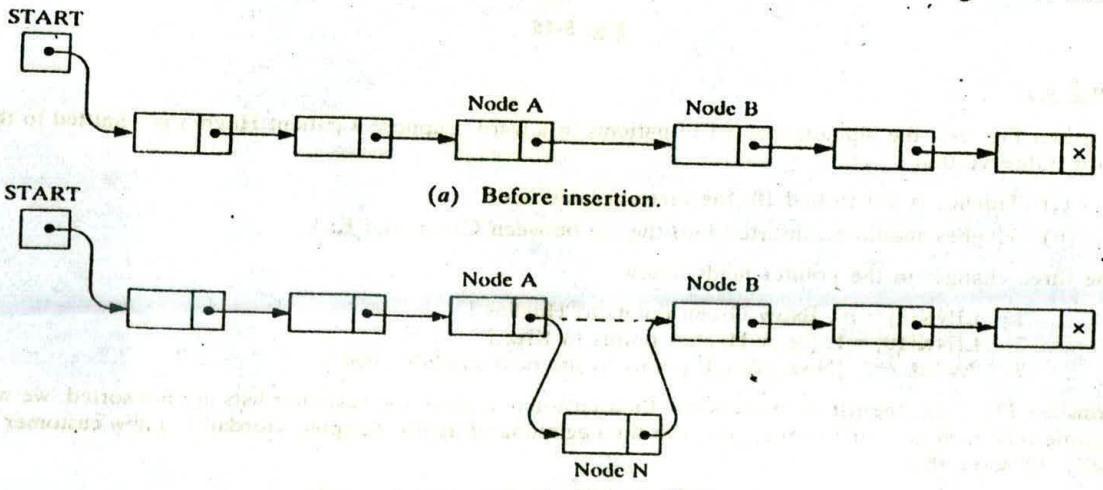
Overflow and Underflow

Sometimes new data are to be inserted into a data structure but there is no available space, i.e., the free-storage list is empty. This situation is usually called *overflow*. The programmer may handle overflow by printing the message OVERFLOW. In such a case, the programmer may then modify the program by adding space to the underlying arrays. Observe that overflow will occur with our linked lists when AVAIL = NULL and there is an insertion.

Analogously, the term *underflow* refers to the situation where one wants to delete data from a data structure that is empty. The programmer may handle underflow by printing the message UNDERFLOW. Observe that underflow will occur with our linked lists when START = NULL and there is a deletion.

5.7 INSERTION INTO A LINKED LIST

Let LIST be a linked list with successive nodes A and B, as pictured in Fig. 5-14(*a*). Suppose a node N is to be inserted into the list between nodes A and B. The schematic diagram of such an



(*a*) Before insertion.

(*b*) After insertion.

Fig. 5-14

insertion appears in Fig. 5-14(*b*). That is, node A now points to the new node N, and node N points to node B, to which A previously pointed.

Suppose our linked list is maintained in memory in the form

LIST(INFO, LINK, START, AVAIL)

Figure 5-14 does not take into account that the memory space for the new node N will come from the AVAIL list. Specifically, for easier processing, the first node in the AVAIL list will be used for the new node N. Thus a more exact schematic diagram of such an insertion is that in Fig. 5-15. Observe that three pointer fields are changed as follows:

(1) The nextpointer field of node A now points to the new node N, to which AVAIL previously pointed.

(2) AVAIL now points to the second node in the free pool, to which node N previously pointed.

(3) The nextpointer field of node N now points to node B, to which node A previously pointed.

There are also two special cases. If the new node N is the first node in the list, then START will point to N; and if the new node N is the last node in the list, then N will contain the null pointer.

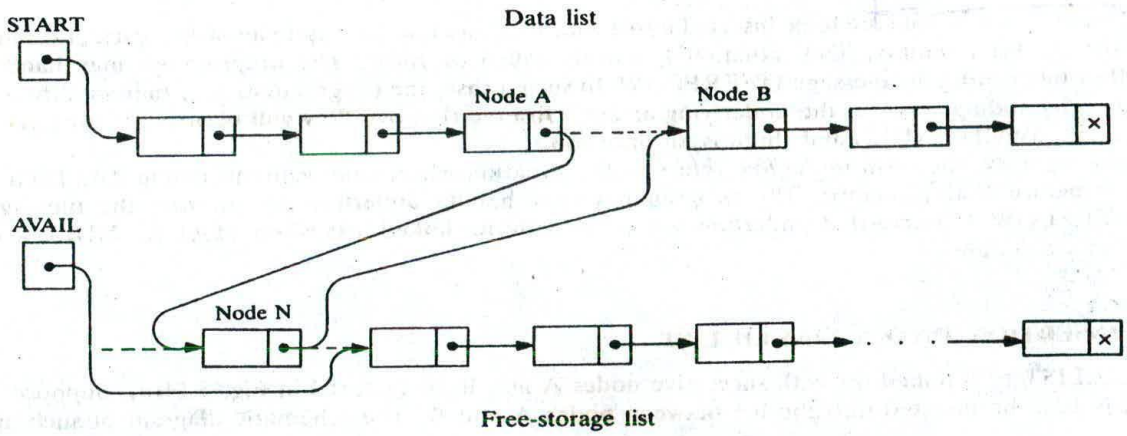


Fig. 5-15

EXAMPLE 5.13

(*a*) Consider Fig. 5-9, the alphabetical list of patients in a ward. Suppose a patient Hughes is admitted to the ward. Observe that

(i) Hughes is put in bed 10, the first available bed.

(ii) Hughes should be inserted into the list between Green and Kirk.

The three changes in the pointer fields follow.

1. LINK[8] = 10. [Now Green points to Hughes.]
2. LINK[10] = 1. [Now Hughes points to Kirk.]
3. AVAIL = 2. [Now AVAIL points to the next available bed.]

(*b*) Consider Fig. 5-12, the list of brokers and their customers. Since the customer lists are not sorted, we will assume that each new customer is added to the beginning of its list. Suppose Gordan is a new customer of Kelly. Observe that

(i) Gordan is assigned to CUSTOMER[11], the first available node.

(ii) Gordan is inserted before Hunter, the previous first customer of Kelly.

The three changes in the pointer fields follow:

1. POINT[2] = 11. [Now the list begins with Gordan.]
2. LINK[11] = 3. [Now Gordan points to Hunter.]
3. AVAIL = 18. [Now AVAIL points to the next available node.]

(*c*) Suppose the data elements A, B, C, D, E and F are inserted one after the other into the empty list in Fig. 5-13. Again we assume that each new node is inserted at the beginning of the list. Accordingly, after the six insertions, F will point to E, which points to D, which points to C, which points to B, which points to A; and A will contain the null pointer. Also, AVAIL = 7, the first available node after the six insertions, and START = 6, the location of the first node, F. Figure 5-16 shows the new list (where $n = 10$.)

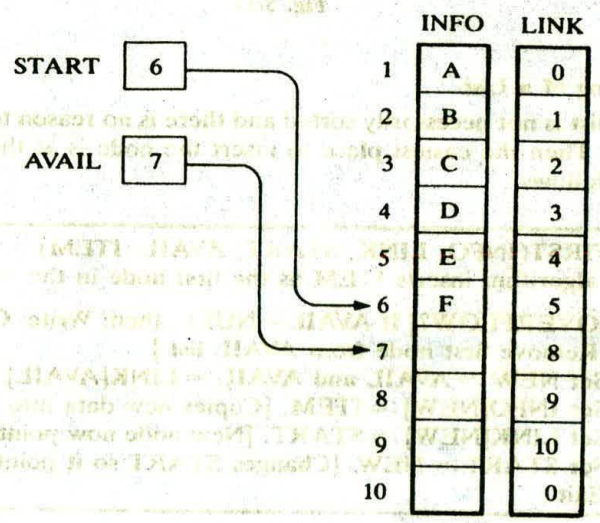


Fig. 5-16

Insertion Algorithms

Algorithms which insert nodes into linked lists come up in various situations. We discuss three of them here. The first one inserts a node at the beginning of the list, the second one inserts a node after the node with a given location, and the third one inserts a node into a sorted list. All our algorithms assume that the linked list is in memory in the form LIST(INFO, LINK, START, AVAIL) and that the variable ITEM contains the new information to be added to the list.

Since our insertion algorithms will use a node in the AVAIL list, all of the algorithms will include the following steps:

(*a*) Checking to see if space is available in the AVAIL list. If not, that is, if AVAIL = NULL, then the algorithm will print the message OVERFLOW.

(*b*) Removing the first node from the AVAIL list. Using the variable NEW to keep track of the location of the new node, this step can be implemented by the pair of assignments (in this order)

$$\text{NEW} := \text{AVAIL}, \qquad \text{AVAIL} := \text{LINK[AVAIL]}$$

(*c*) Copying new information into the new node. In other words,

$$\text{INFO[NEW]} := \text{ITEM}$$

The schematic diagram of the latter two steps is pictured in Fig. 5-17.

Fig. 5-17

Inserting at the Beginning of a List

Suppose our linked list is not necessarily sorted and there is no reason to insert a new node in any special place in the list. Then the easiest place to insert the node is at the begining of the list. An algorithm that does so follows.

Algorithm 5.4: INSFIRST(INFO, LINK, START, AVAIL, ITEM)
This algorithm inserts ITEM as the first node in the list.

1. [OVERFLOW?] If AVAIL = NULL, then: Write: OVERFLOW, and Exit.
2. [Remove first node from AVAIL list.]
   Set NEW := AVAIL and AVAIL := LINK[AVAIL].
3. Set INFO[NEW] := ITEM. [Copies new data into new node.]
4. Set LINK[NEW] := START. [New node now points to original first node.]
5. Set START := NEW. [Changes START so it points to the new node.]
6. Exit.

Steps 1 to 3 have already been discussed, and the schematic diagram of Steps 2 and 3 appears in Fig. 5-17. The schematic diagram of Steps 4 and 5 appears in Fig. 5-18.

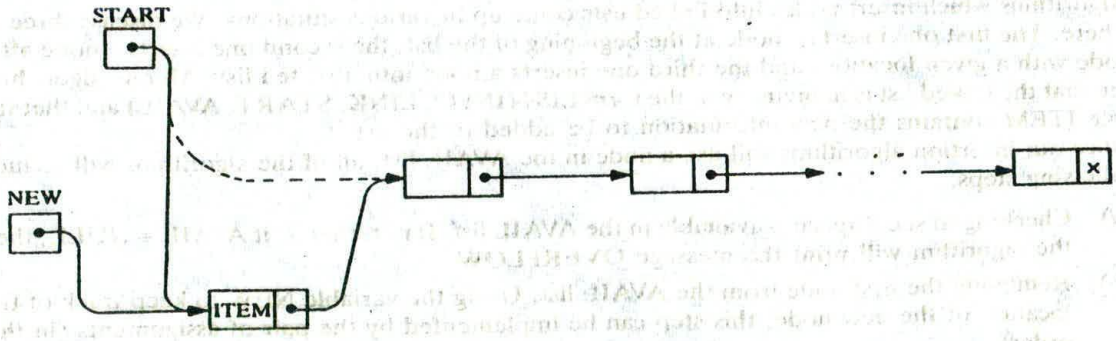


Fig. 5-18 Insertion at the beginning of a list.

EXAMPLE 5.14

Consider the lists of tests in Fig. 5-10. Suppose the test score 75 is to be added to the beginning of the geometry list. We simulate Algorithm 5.4. Observe that ITEM = 75, INFO = TEST and START = GEOM.

INSFIRST(TEST, LINK, GEOM, AVAIL, ITEM)

1. Since AVAIL ≠ NULL, control is transferred to Step 2.
2. NEW = 9, then AVAIL = LINK[9] = 10.
3. TEST[9] = 75.
4. LINK[9] = 5.
5. GEOM = 9.
6. Exit.

Figure 5-19 shows the data structure after 75 is added to the geometry list. Observe that only three pointers are changed, AVAIL, GEOM and LINK[9].

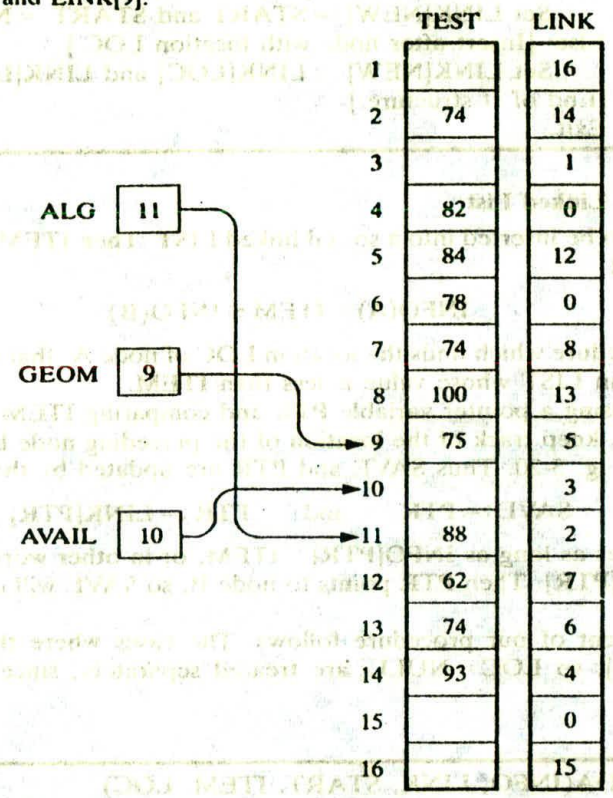


Fig. 5-19

Inserting after a Given Node

Suppose we are given the value of LOC where either LOC is the location of a node A in a linked LIST or LOC = NULL. The following is an algorithm which inserts ITEM into LIST so that ITEM follows node A or, when LOC = NULL, so that ITEM is the first node.

Let N denote the new node (whose location is NEW). If LOC = NULL, then N is inserted as the first node in LIST as in Algorithm 5.4. Otherwise, as pictured in Fig. 5-15, we let node N point to node B (which originally followed node A) by the assignment

$$\text{LINK[NEW]} := \text{LINK[LOC]}$$

and we let node A point to the new node N by the assignment

$$\text{LINK[LOC]} := \text{NEW}$$

A formal statement of the algorithm follows.

Algorithm 5.5: INSLOC(INFO, LINK, START, AVAIL, LOC, ITEM)
This algorithm inserts ITEM so that ITEM follows the node with location LOC or inserts ITEM as the first node when LOC = NULL.

1. [OVERFLOW?] If AVAIL = NULL, then: Write: OVERFLOW, and Exit.
2. [Remove first node from AVAIL list.]
   Set NEW := AVAIL and AVAIL := LINK[AVAIL].
3. Set INFO[NEW] := ITEM. [Copies new data into new node.]
4. If LOC = NULL, then: [Insert as first node.]
   Set LINK[NEW] := START and START := NEW.
   Else: [Insert after node with location LOC.]
   Set LINK[NEW] := LINK[LOC] and LINK[LOC] := NEW.
   [End of If structure.]
5. Exit.

Inserting into a Sorted Linked List

Suppose ITEM is to be inserted into a sorted linked LIST. Then ITEM must be inserted between nodes A and B so that

$$\text{INFO(A)} < \text{ITEM} \leq \text{INFO(B)}$$

The following is a procedure which finds the location LOC of node A, that is, which finds the location LOC of the last node in LIST whose value is less than ITEM.

Traverse the list, using a pointer variable PTR and comparing ITEM with INFO[PTR] at each node. While traversing, keep track of the location of the preceding node by using a pointer variable SAVE, as pictured in Fig. 5-20. Thus SAVE and PTR are updated by the assignments

$$\text{SAVE} := \text{PTR} \qquad \text{and} \qquad \text{PTR} := \text{LINK[PTR]}$$

The traversing continues as long as INFO[PTR] > ITEM, or in other words, the traversing stops as soon as ITEM ≤ INFO[PTR]. Then PTR points to node B, so SAVE will contain the location of the node A.

The formal statement of our procedure follows. The cases where the list is empty or where ITEM < INFO[START], so LOC = NULL, are treated separately, since they do not involve the variable SAVE.

Procedure 5.6: FINDA(INFO, LINK, START, ITEM, LOC)
This procedure finds the location LOC of the last node in a sorted list such that INFO[LOC] < ITEM, or sets LOC = NULL.

1. [List empty?] If START = NULL, then: Set LOC := NULL, and Return.
2. [Special case?] If ITEM < INFO[START], then: Set LOC := NULL, and Return.
3. Set SAVE := START and PTR := LINK[START]. [Initializes pointers.]
4. Repeat Steps 5 and 6 while PTR ≠ NULL.
5. If ITEM < INFO[PTR], then:
   Set LOC := SAVE, and Return.
   [End of If structure.]
6. Set SAVE := PTR and PTR := LINK[PTR]. [Updates pointers.]
   [End of Step 4 loop.]
7. Set LOC := SAVE.
8. Return.

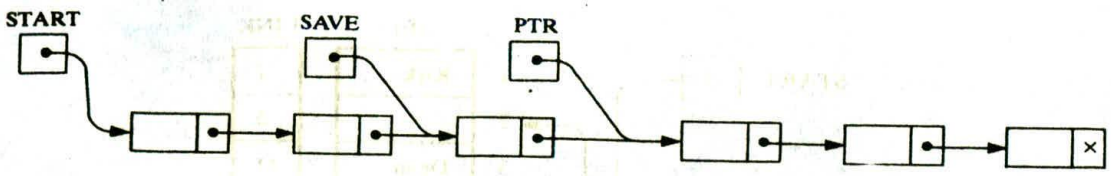


Fig. 5-20

Now we have all the components to present an algorithm which inserts ITEM into a linked list. The simplicity of the algorithm comes from using the previous two procedures.

Algorithm 5.7: INSSRT(INFO, LINK, START, AVAIL, ITEM)
This algorithm inserts ITEM into a sorted linked list.

1. [Use Procedure 5.6 to find the location of the node preceding ITEM.]
   Call FINDA(INFO, LINK, START, ITEM, LOC).
2. [Use Algorithm 5.5 to insert ITEM after the node with location LOC.]
   Call INSLOC(INFO, LINK, START, AVAIL, LOC, ITEM).
3. Exit.

EXAMPLE 5.15

Consider the alphabetized list of patients in Fig. 5-9. Suppose Jones is to be added to the list of patients. We simulate Algorithm 5.7, or more specifically, we simulate Procedure 5.6 and then Algorithm 5.5. Observe that ITEM = Jones and INFO = BED.

(*a*) FINDA(BED, LINK, START, ITEM, LOC)
1. Since START $\neq$ NULL, control is transferred to Step 2.
2. Since BED[5] = Adams < Jones, control is transferred to Step 3.
3. SAVE = 5 and PTR = LINK[5] = 3.
4. Steps 5 and 6 are repeated as follows:
   (*a*) BED[3] = Dean < Jones, so SAVE = 3 and PTR = LINK[3] = 11.
   (*b*) BED[11] = Fields < Jones, so SAVE = 11 and PTR = LINK[11] = 8.
   (*c*) BED[8] = Green < Jones, so SAVE = 8 and PTR = LINK[8] = 1.
   (*d*) Since BED[1] = Kirk > Jones, we have:
   LOC = SAVE = 8 and Return.

(*b*) INSLOC(BED, LINK, START, AVAIL, LOC, ITEM) [Here LOC = 8.]
1. Since AVAIL $\neq$ NULL, control is transferred to Step 2.
2. NEW = 10 and AVAIL = LINK[10] = 2.
3. BED[10] = Jones.
4. Since LOC $\neq$ NULL we have:
   LINK[10] = LINK[8] = 1 and LINK[8] = NEW = 10.
5. Exit.

Figure 5-21 shows the data structure after Jones is added to the patient list. We emphasize that only three pointers have been changed, AVAIL, LINK[10] and LINK[8].

Copying

Suppose we want to copy all or part of a given list, or suppose we want to form a new list that is the concatenation of two given lists. This can be done by defining a null list and then adding the appropriate elements to the list, one by one, by various insertion algorithms. A null list is defined by simply choosing a variable name or pointer for the list, such as NAME, and then setting NAME := NULL. These algorithms are covered in the problem sections.

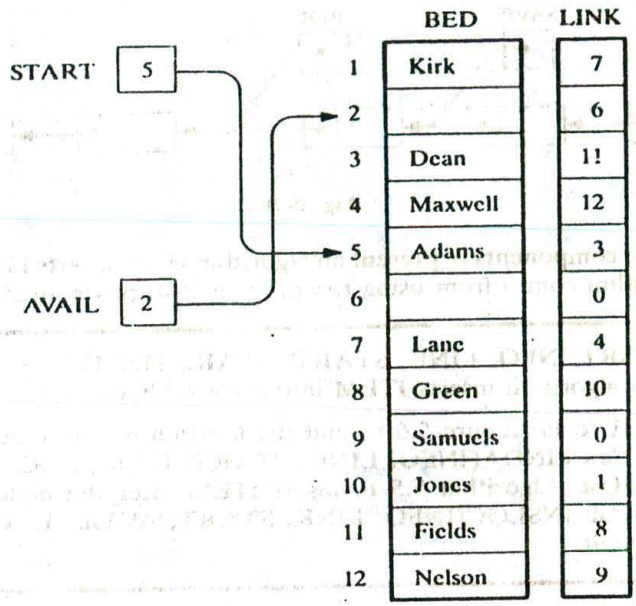


Fig. 5-21

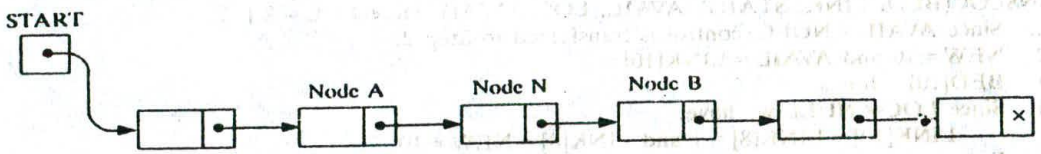
5.8 DELETION FROM A LINKED LIST

Let LIST be a linked list with a node N between nodes A and B, as pictured in Fig. 5-22(*a*). Suppose node N is to be deleted from the linked list. The schematic diagram of such a deletion appears in Fig. 5-22(*b*). The deletion occurs as soon as the nextpointer field of node A is changed so that it points to node B. (Accordingly, when performing deletions, one must keep track of the address of the node which immediately precedes the node that is to be deleted.)

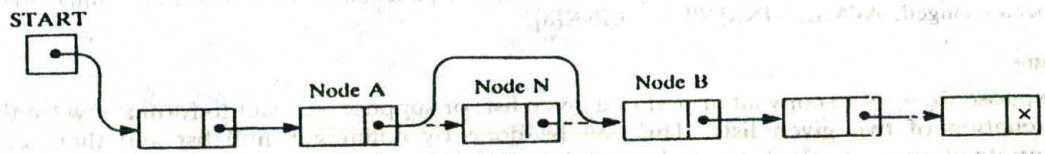
Suppose our linked list is maintained in memory in the form

LIST(INFO, LINK, START, AVAIL)

Figure 5-22 does not take into account the fact that, when a node N is deleted from our list, we will



(*a*) Before deletion.



(*b*) After deletion.

Fig. 5-22

immediately return its memory space to the AVAIL list. Specifically, for easier processing, it will be returned to the beginning of the AVAIL list. Thus a more exact schematic diagram of such a deletion is the one in Fig. 5-23. Observe that three pointer fields are changed as follows:

(1) The nextpointer field of node A now points to node B, where node N previously pointed.

(2) The nextpointer field of N now points to the original first node in the free pool, where AVAIL previously pointed.

(3) AVAIL now points to the deleted node N.

There are also two special cases. If the deleted node N is the first node in the list, then START will point to node B; and if the deleted node N is the last node in the list, then node A will contain the NULL pointer.

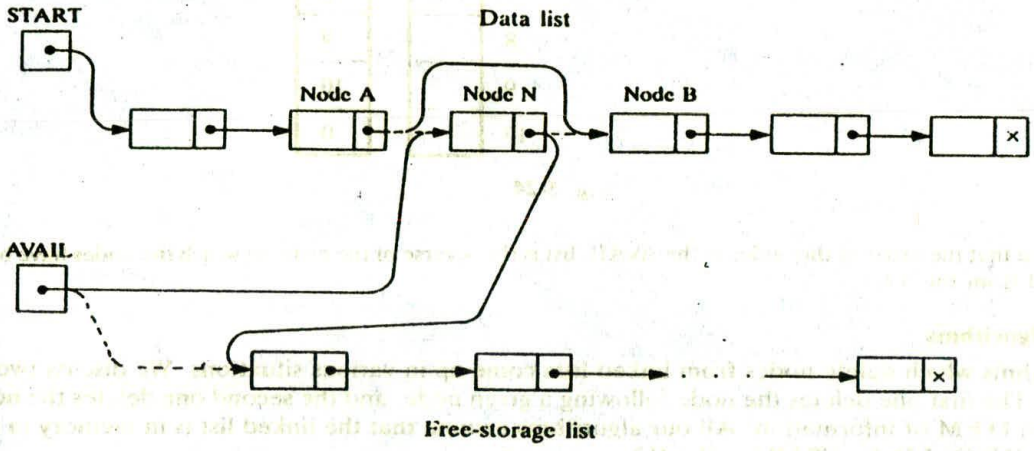


Fig. 5-23

EXAMPLE 5.16

(*a*) Consider Fig. 5-21, the list of patients in the hospital ward. Suppose Green is discharged, so that BED[8] is now empty. Then, in order to maintain the linked list, the following three changes in the pointer fields must be executed:

LINK[11] = 10 LINK[8] = 2 AVAIL = 8

By the first change, Fields, who originally preceded Green, now points to Jones, who originally followed Green. The second and third changes add the new empty bed to the AVAIL list. We emphasize that, before making the deletion, we had to find the node BED[11], which originally pointed to the deleted node BED[8].

(*b*) Consider Fig. 5-12, the list of brokers and their customers. Suppose Teller, the first customer of Nelson, is deleted from the list of customers. Then, in order to maintain the linked lists, the following three changes in the pointer fields must be executed:

POINT[4] = 10 LINK[9] = 11 AVAIL = 9

By the first change, Nelson now points to his original second customer, Jones. The second and third changes add the new empty node to the AVAIL list.

(*c*) Suppose the data elements E, B and C are deleted, one after the other, from the list in Fig. 5-16. The new list is pictured in Fig. 5-24. Observe that now the first three available nodes are:

INFO[3], which originally contained C
INFO[2], which originally contained B
INFO[5], which originally contained E

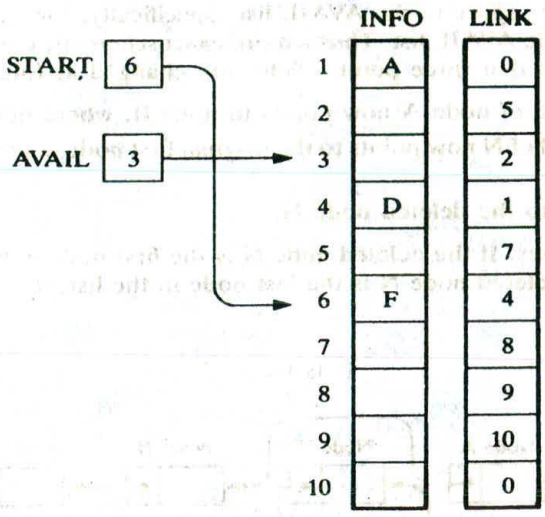


Fig. 5-24

Observe that the order of the nodes in the AVAIL list is the reverse of the order in which the nodes have been deleted from the list.

Deletion Algorithms

Algorithms which delete nodes from linked lists come up in various situations. We discuss two of them here. The first one deletes the node following a given node, and the second one deletes the node with a given ITEM of information. All our algorithms assume that the linked list is in memory in the form LIST(INFO, LINK, START, AVAIL).

All of our deletion algorithms will return the memory space of the deleted node N to the beginning of the AVAIL list. Accordingly, all of our algorithms will include the following pair of assignments, where LOC is the location of the deleted node N:

$$\text{LINK[LOC]} := \text{AVAIL} \quad \text{and then} \quad \text{AVAIL} := \text{LOC}$$

These two operations are pictured in Fig. 5-25.

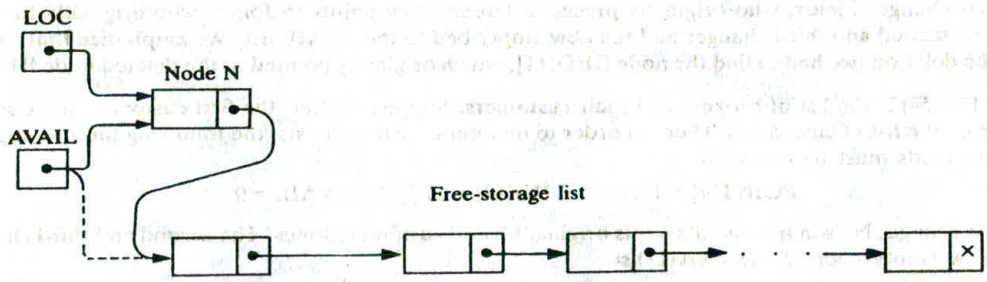


Fig. 5-25 LINK[LOC] := AVAIL and AVAIL := LOC.

Some of our algorithms may want to delete either the first node or the last node from the list. An algorithm that does so must check to see if there is a node in the list. If not, i.e., if START = NULL, then the algorithm will print the message UNDERFLOW.

Deleting the Node Following a Given Node

Let LIST be a linked list in memory. Suppose we are given the location LOC of a node N in LIST. Furthermore, suppose we are given the location LOCP of the node preceding N or, when N is the first node, we are given LOCP = NULL. The following algorithm deletes N from the list.

Algorithm 5.8: DEL(INFO, LINK, START, AVAIL, LOC, LOCP)
This algorithm deletes the node N with location LOC. LOCP is the location of the node which precedes N or, when N is the first node, LOCP = NULL.

1. If LOCP = NULL, then:
   Set START := LINK[START]. [Deletes first node.]
   Else:
   Set LINK[LOCP] := LINK[LOC]. [Deletes node N.]
   [End of If structure.]
2. [Return deleted node to the AVAIL list.]
   Set LINK[LOC] := AVAIL and AVAIL := LOC.
3. Exit.

Figure 5-26 is the schematic diagram of the assignment

START := LINK[START]

which effectively deletes the first node from the list. This covers the case when N is the first node.

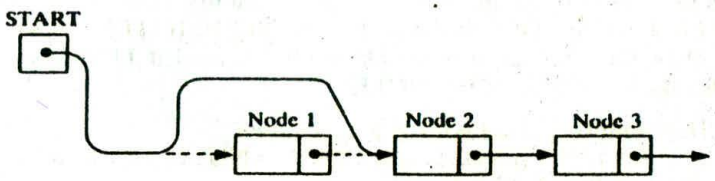


Fig. 5-26 START := LINK[START].

Figure 5-27 is the schematic diagram of the assignment

LINK[LOCP] := LINK[LOC]

which effectively deletes the node N when N is not the first node.

The simplicity of the algorithm comes from the fact that we are already given the location LOCP of the node which precedes node N. In many applications, we must first find LOCP.

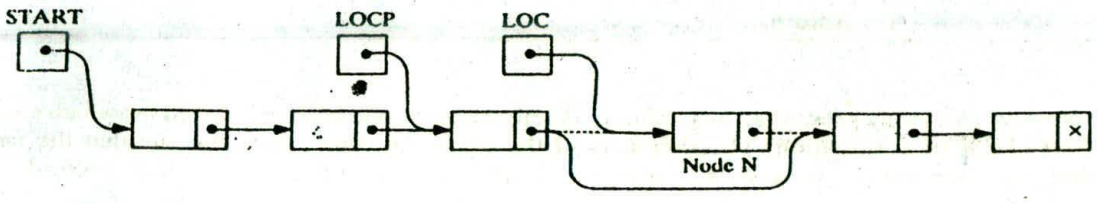


Fig. 5-27 LINK[LOCP] := LINK[LOC].

Deleting the Node with a Given ITEM of Information

Let LIST be a linked list in memory. Suppose we are given an ITEM of information and we want to delete from the LIST the first node N which contains ITEM. (If ITEM is a key value, then only one node can contain ITEM.) Recall that before we can delete N from the list, we need to know the location of the node preceding N. Accordingly, first we give a procedure which finds the location LOC of the node N containing ITEM and the location LOCP of the node preceding node N. If N is the first node, we set LOCP = NULL, and if ITEM does not appear in LIST, we set LOC = NULL. (This procedure is similar to Procedure 5.6.)

Traverse the list, using a pointer variable PTR and comparing ITEM with INFO[PTR] at each node. While traversing, keep track of the location of the preceding node by using a pointer variable SAVE, as pictured in Fig. 5-20. Thus SAVE and PTR are updated by the assignments

$$\text{SAVE} := \text{PTR} \quad \text{and} \quad \text{PTR} := \text{LINK[PTR]}$$

The traversing continues as long as INFO[PTR] $\neq$ ITEM, or in other words, the traversing stops as soon as ITEM = INFO[PTR]. Then PTR contains the location LOC of node N and SAVE contains the location LOCP of the node preceding N.

The formal statement of our procedure follows. The cases where the list is empty or where INFO[START] = ITEM (i.e., where node N is the first node) are treated separately, since they do not involve the variable SAVE.

Procedure 5.9: FINDB(INFO, LINK, START, ITEM, LOC, LOCP)
This procedure finds the location LOC of the first node N which contains ITEM and the location LOCP of the node preceding N. If ITEM does not appear in the list, then the procedure sets LOC = NULL; and if ITEM appears in the first node, then it sets LOCP = NULL.

```
1. [List empty?] If START = NULL, then:
       Set LOC := NULL and LOCP := NULL, and Return.
   [End of If structure.]
2. [ITEM in first node?] If INFO[START] = ITEM, then:
       Set LOC := START and LOCP = NULL, and Return.
   [End of If structure.]
3. Set SAVE := START and PTR := LINK[START]. [Initializes pointers.]
4. Repeat Steps 5 and 6 while PTR ≠ NULL.
5.     If INFO[PTR] = ITEM, then:
           Set LOC := PTR and LOCP := SAVE, and Return.
       [End of If structure.]
6.     Set SAVE := PTR and PTR := LINK[PTR]. [Updates pointers.]
   [End of Step 4 loop.]
7. Set LOC := NULL. [Search unsuccessful.]
8. Return.
```

Now we can easily present an algorithm to delete the first node N from a linked list which contains a given ITEM of information. The simplicity of the algorithm comes from the fact that the task of finding the location of N and the location of its preceding node has already been done in Procedure 5.9.

Algorithm 5.10: DELETE(INFO, LINK, START, AVAIL, ITEM)
This algorithm deletes from a linked list the first node N which contains the given ITEM of information.

1. [Use Procedure 5.9 to find the location of N and its preceding node.]
   Call FINDB(INFO, LINK, START, ITEM, LOC, LOCP)
2. If LOC = NULL, then: Write: ITEM not in list, and Exit.
3. [Delete node.]
   If LOCP = NULL, then:
      Set START := LINK[START]. [Deletes first node.]
   Else:
      Set LINK[LOCP] := LINK[LOC].
   [End of If structure.]
4. [Return deleted node to the AVAIL list.]
   Set LINK[LOC] := AVAIL and AVAIL := LOC.
5. Exit.

Remark: The reader may have noticed that Steps 3 and 4 in Algorithm 5.10 already appear in Algorithm 5.8. In other words, we could replace the steps by the following Call statement:

Call DEL(INFO, LINK, START, AVAIL, LOC, LOCP)

This would conform to the usual programming style of modularity.

EXAMPLE 5.17

Consider the list of patients in Fig. 5-21. Suppose the patient Green is discharged. We simulate Procedure 5.9 to find the location LOC of Green and the location LOCP of the patient preceding Green. Then we simulate Algorithm 5.10 to delete Green from the list. Here ITEM = Green, INFO = BED, START = 5 and AVAIL = 2.

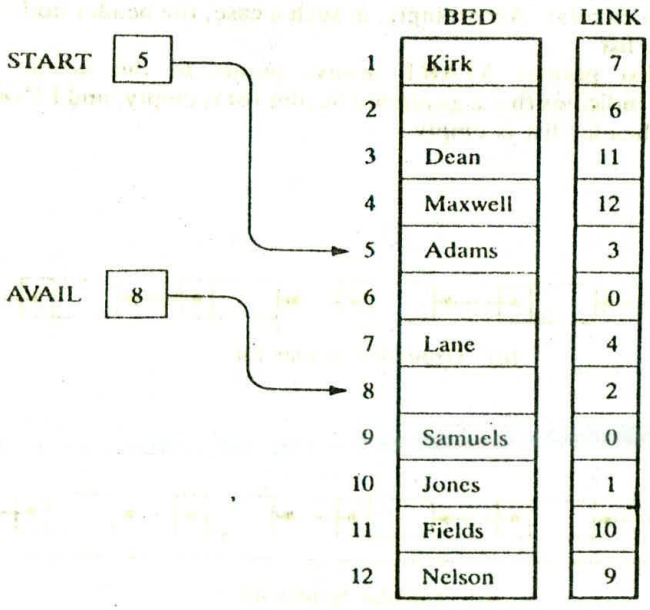


Fig. 5-28

(*a*) FINDB(BED, LINK, START, ITEM, LOC, LOCP)
1. Since START ≠ NULL, control is transferred to Step 2.
2. Since BED[5] = Adams ≠ Green, control is transferred to Step 3.
3. SAVE = 5 and PTR = LINK[5] = 3.
4. Steps 5 and 6 are repeated as follows:
   (*a*) BED[3] = Dean ≠ Green, so SAVE = 3 and PTR = LINK[3] = 11.
   (*b*) BED[11] = Fields ≠ Green, so SAVE = 11 and PTR = LINK[11] = 8.
   (*c*) BED[8] = Green, so we have:
   LOC = PTR = 8 and LOCP = SAVE = 11, and Return.

(*b*) DELLOC(BED, LINK, START, AVAIL, ITEM)
1. Call FINDB(BED, LINK, START, ITEM, LOC, LOCP). [Hence LOC = 8 and LOCP = 11.]
2. Since LOC ≠ NULL, control is transferred to Step 3.
3. Since LOCP ≠ NULL, we have:
   LINK[11] = LINK[8] = 10.
4. LINK[8] = 2 and AVAIL = 8.
5. Exit.

Figure 5-28 shows the data structure after Green is removed from the patient list. We emphasize that only three pointers have been changed, LINK[11], LINK[8] and AVAIL.

5.9 HEADER LINKED LISTS

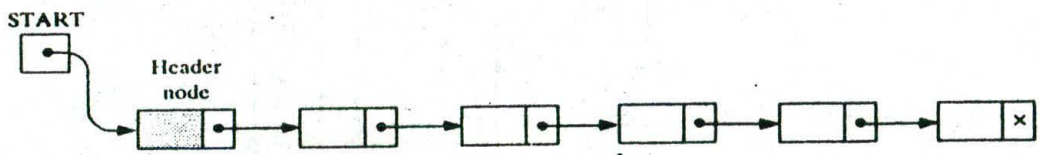
A *header* linked list is a linked list which always contains a special node, called the *header node*, at the beginning of the list. The following are two kinds of widely used header lists:

(1) A *grounded header list* is a header list where the last node contains the null pointer. (The term "grounded" comes from the fact that many texts use the electrical ground symbol to indicate the null pointer.)

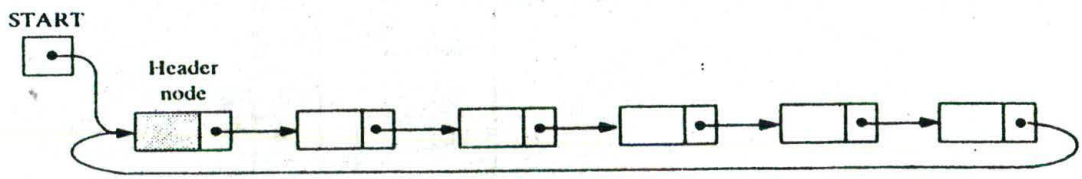
(2) A *circular header list* is a header list where the last node points back to the header node.

Figure 5-29 contains schematic diagrams of these header lists. Unless otherwise stated or implied, our header lists will always be circular. Accordingly, in such a case, the header node also acts as a sentinel indicating the end of the list.

Observe that the list pointer START always points to the header node. Accordingly, LINK[START] = NULL indicates that a grounded header list is empty, and LINK[START] = START indicates that a circular header list is empty.



(*a*) Grounded header list.



(*b*) Circular header list.

Fig. 5-29

Although our data may be maintained by header lists in memory, the AVAIL list will always be maintained as an ordinary linked list.

EXAMPLE 5.18

Consider the personnel file in Fig. 5-11. The data may be organized as a header list as in Fig. 5-30. Observe that LOC = 5 is now the location of the header record. Therefore, START = 5, and since Rubin is the last employee, LINK[10] = 5. The header record may also be used to store information about the entire file. For example, we let SSN[5] = 9 indicate the number of employees, and we let SALARY[5] = 191 600 indicate the total salary paid to the employees.

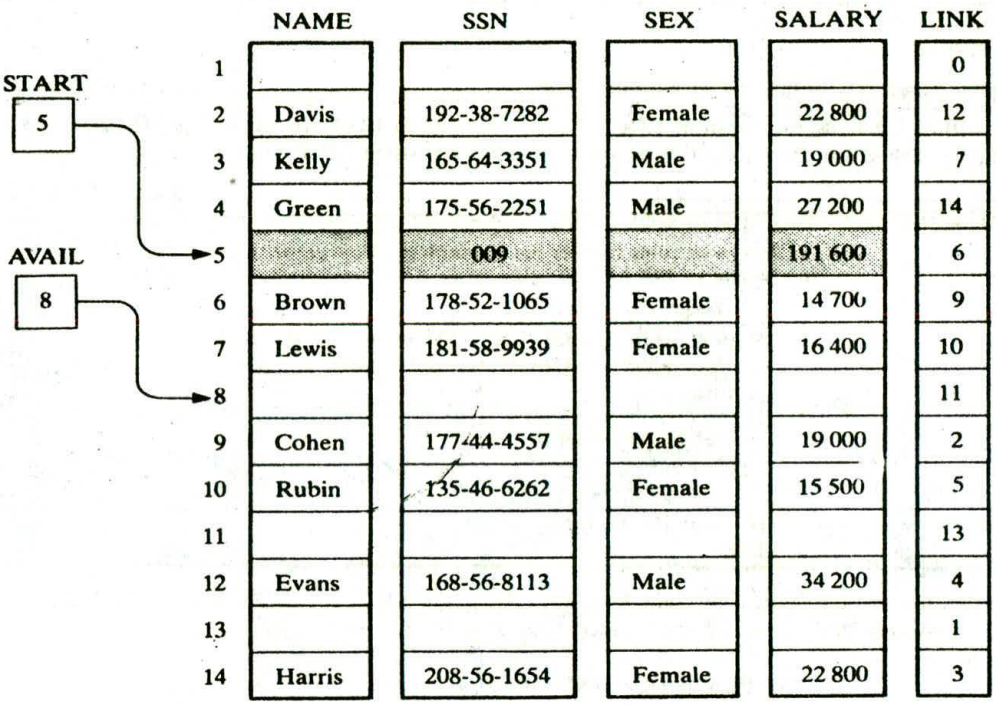


Fig. 5-30

The term "node," by itself, normally refers to an ordinary node, not the header node, when used with header lists. Thus the *first node* in a header list is the node following the header node, and the location of the first node is LINK[START], not START, as with ordinary linked lists.

Algorithm 5.11, which uses a pointer variable PTR to traverse a circular header list, is essentially the same as Algorithm 5.1, which traverses an ordinary linked list, except that now the algorithm (1) begins with PTR = LINK[START] (not PTR = START) and (2) ends when PTR = START (not PTR = NULL).

Circular header lists are frequently used instead of ordinary linked lists because many operations are much easier to state and implement using header lists. This comes from the following two properties of circular header lists:

(1) The null pointer is not used, and hence all pointers contain valid addresses.

(2) Every (ordinary) node has a predecessor, so the first node may not require a special case.

The next example illustrates the usefulness of these properties.

Algorithm 5.11: (Traversing a Circular Header List) Let LIST be a circular header list in memory. This algorithm traverses LIST, applying an operation PROCESS to each node of LIST.

1. Set PTR := LINK[START]. [Initializes the pointer PTR.]
2. Repeat Steps 3 and 4 while PTR ≠ START:
3. Apply PROCESS to INFO[PTR].
4. Set PTR := LINK[PTR]. [PTR now points to the next node.]
   [End of Step 2 loop.]
5. Exit.

EXAMPLE 5.19

Suppose LIST is a linked list in memory, and suppose a specific ITEM of information is given.

(*a*) Algorithm 5.2 finds the location LOC of the first node in LIST which contains ITEM when LIST is an ordinary linked list. The following is such an algorithm when LIST is a circular header list.

Algorithm 5.12: SRCHHL(INFO, LINK, START, ITEM, LOC)
LIST is a circular header list in memory. This algorithm finds the location LOC of the node where ITEM first appears in LIST or sets LOC = NULL.

1. Set PTR := LINK[START].
2. Repeat while INFO[PTR] ≠ ITEM and PTR ≠ START:
   Set PTR := LINK[PTR]. [PTR now points to the next node.]
   [End of loop.]
3. If INFO[PTR] = ITEM, then:
   Set LOC := PTR.
   Else:
   Set LOC := NULL.
   [End of If structure.]
4. Exit.

The two tests which control the searching loop (Step 2 in Algorithm 5.12) were not performed at the same time in the algorithm for ordinary linked lists; that is, we did not let Algorithm 5.2 use the analogous statement

Repeat while INFO[PTR] ≠ ITEM and PTR ≠ NULL:

because for ordinary linked lists INFO[PTR] is not defined when PTR = NULL.

(*b*) Procedure 5.9 finds the location LOC of the first node N which contains ITEM and also the location LOCP of the node preceding N when LIST is an ordinary linked list. The following is such a procedure when LIST is a circular header list.

Procedure 5.13: FINDBHL(INFO, LINK, START, ITEM, LOC, LOCP)

1. Set SAVE := START and PTR := LINK[START]. [Initializes pointers.]
2. Repeat while INFO[PTR] ≠ ITEM and PTR ≠ START.
   Set SAVE := PTR and PTR := LINK[PTR]. [Updates pointers.]
   [End of loop.]
3. If INFO[PTR] = ITEM, then:
   Set LOC := PTR and LOCP := SAVE.
   Else:
   Set LOC := NULL and LOCP := SAVE.
   [End of If structure.]
4. Exit.

Observe the simplicity of this procedure compared with Procedure 5.9. Here we did not have to consider the special case when ITEM appears in the first node, and here we can perform at the same time the two tests which control the loop.

(c) Algorithm 5.10 deletes the first node N which contains ITEM when LIST is an ordinary linked list. The following is such an algorithm when LIST is a circular header list.

Algorithm 5.14: DELLOCHL(INFO, LINK, START, AVAIL, ITEM)

1. [Use Procedure 5.13 to find the location of N and its preceding node.]
   Call LINDBHL(INFO, LINK, START, ITEM, LOC, LOCP).
2. If LOC = NULL, then: Write: ITEM not in list, and Exit.
3. Set LINK[LOCP] := LINK[LOC]. [Deletes node.]
4. [Return deleted node to the AVAIL list.]
   Set LINK[LOC] := AVAIL and AVAIL := LOC.
5. Exit.

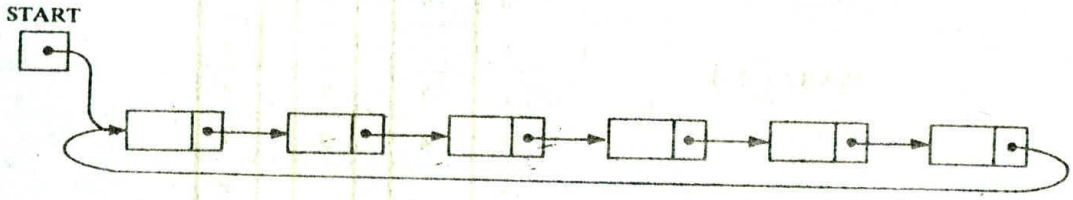
Again we did not have to consider the special case when ITEM appears in the first node, as we did in Algorithm 5.10.

Remark: There are two other variations of linked lists which sometimes appear in the literature:

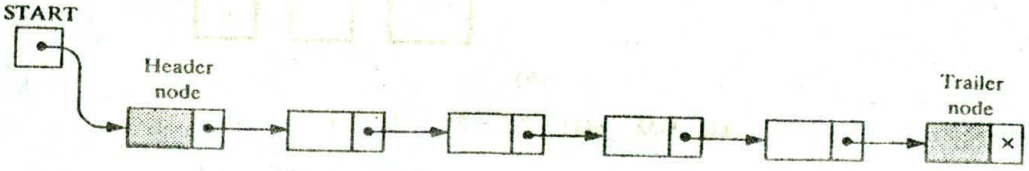
(1) A linked list whose last node points back to the first node instead of containing the null pointer, called a *circular list*

(2) A linked list which contains both a special header node at the beginning of the list and a special trailer node at the end of the list

Figure 5-31 contains schematic diagrams of these lists.



(*a*) Circular linked list.



(*b*) Linked list with header and trailer nodes.

Fig. 5-31

Polynomials

Header linked lists are frequently used for maintaining polynomials in memory. The header node plays an important part in this representation, since it is needed to represent the zero polynomial. This representation of polynomials will be presented in the context of a specific example.

EXAMPLE 5.20

Let $p(x)$ denote the following polynomial in one variable (containing four nonzero terms):

$$p(x) = 2x^8 - 5x^7 - 3x^2 + 4$$

Then $p(x)$ may be represented by the header list pictured in Fig. 5-32(a), where each node corresponds to a nonzero term of $p(x)$. Specifically, the information part of the node is divided into two fields representing, respectively, the coefficient and the exponent of the corresponding term, and the nodes are linked according to decreasing degree.

Observe that the list pointer variable POLY points to the header node, whose exponent field is assigned a negative number, in this case -1. Here the array representation of the list will require three linear arrays, which we will call COEF, EXP and LINK. One such representation appears in Fig. 5-32(b).

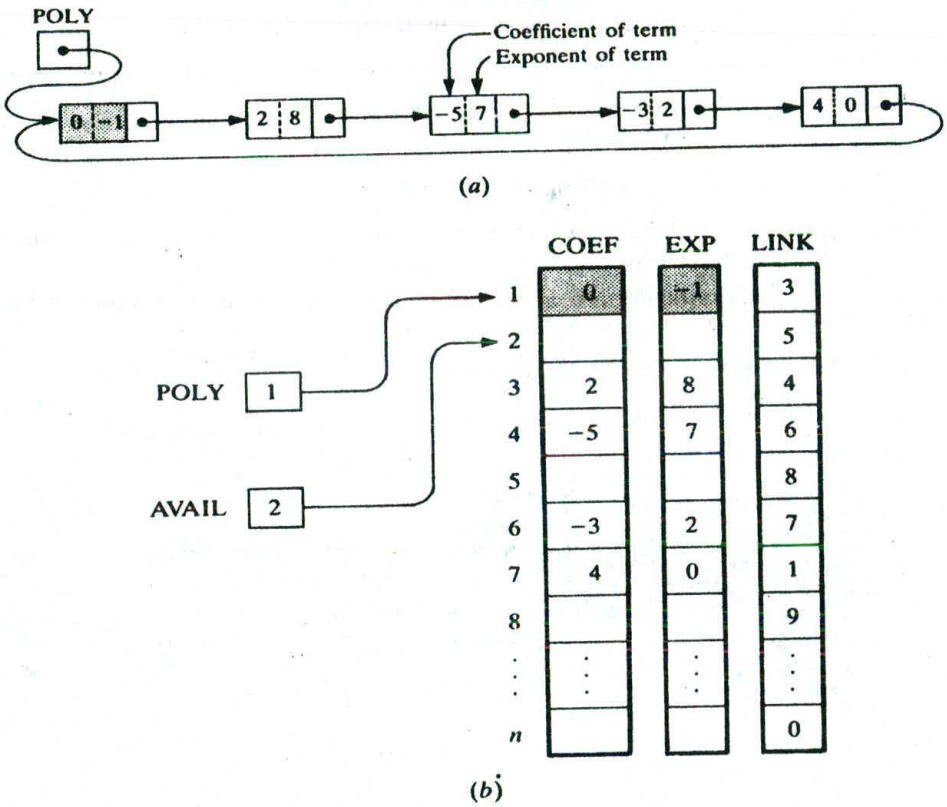


Fig. 5-32 $p(x) = 2x^8 - 5x^7 - 3x^2 + 4$.

5.10 TWO-WAY LISTS

Each list discussed above is called a one-way list, since there is only one way that the list can be traversed. That is, beginning with the list pointer variable START, which points to the first node or the header node, and using the nextpointer field LINK to point to the next node in the list, we can traverse the list in only one direction. Furthermore, given the location LOC of a node N in such a list, one has immediate access to the next node in the list (by evaluating LINK[LOC]), but one does not have access to the preceding node without traversing part of the list. This means, in particular, that one must traverse that part of the list preceding N in order to delete N from the list.

This section introduces a new list structure, called a two-way list, which can be traversed in two directions: in the usual forward direction from the beginning of the list to the end, or in the backward direction from the end of the list to the beginning. Furthermore, given the location LOC of a node N in the list, one now has immediate access to both the next node and the preceding node in the list. This means, in particular, that one is able to delete N from the list without traversing any part of the list.

A *two-way list* is a linear collection of data elements, called *nodes*, where each node N is divided into three parts:

(1) An information field INFO which contains the data of N

(2) A pointer field FORW which contains the location of the next node in the list

(3) A pointer field BACK which contains the location of the preceding node in the list

The list also requires two list pointer variables: FIRST, which points to the first node in the list, and LAST, which points to the last node in the list. Figure 5-33 contains a schematic diagram of such a list. Observe that the null pointer appears in the FORW field of the last node in the list and also in the BACK field of the first node in the list.

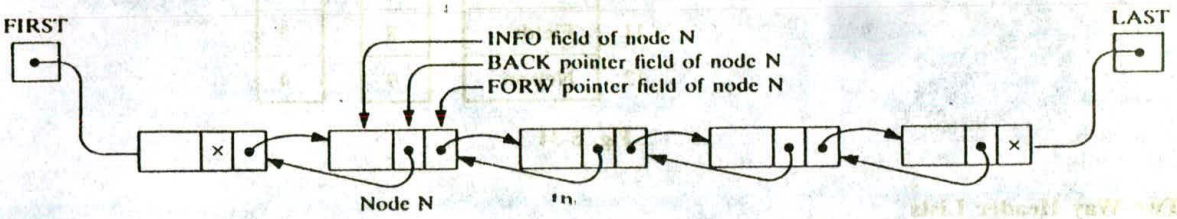


Fig. 5-33 Two-way list.

Observe that, using the variable FIRST and the pointer field FORW, we can traverse a two-way list in the forward direction as before. On the other hand, using the variable LAST and the pointer field BACK, we can also traverse the list in the backward direction.

Suppose LOCA and LOCB are the locations, respectively, of nodes A and B in a two-way list. Then the way that the pointers FORW and BACK are defined gives us the following:

Pointer property: FORW[LOCA] = LOCB if and only if BACK[LOCB] = LOCA

In other words, the statement that node B follows node A is equivalent to the statement that node A precedes node B.

Two-way lists may be maintained in memory by means of linear arrays in the same way as one-way lists except that now we require two pointer arrays, FORW and BACK, instead of one pointer array LINK, and we require two list pointer variables, FIRST and LAST, instead of one list pointer variable START. On the other hand, the list AVAIL of available space in the arrays will still be maintained as a one-way list—using FORW as the pointer field—since we delete and insert nodes only at the beginning of the AVAIL list.

EXAMPLE 5.21

Consider again the data in Fig. 5-9, the 9 patients in a ward with 12 beds. Figure 5-34 shows how the alphabetical listing of the patients can be organized into a two-way list. Observe that the values of FIRST and the pointer field FORW are the same, respectively, as the values of START and the array LINK; hence the list can be traversed alphabetically as before. On the other hand, using LAST and the pointer array BACK, the list can also be traversed in reverse alphabetical order. That is, LAST points to Samuels, the pointer field BACK of Samuels points to Nelson, the pointer field BACK of Nelson points to Maxwell, and so on.

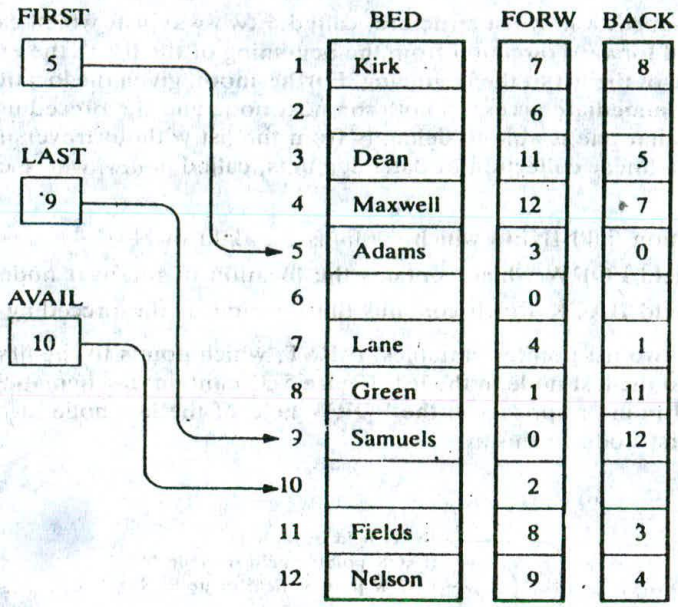


Fig. 5-34

Two-Way Header Lists

The advantages of a two-way list and a circular header list may be combined into a two-way circular header list as pictured in Fig. 5-35. The list is circular because the two end nodes point back to the header node. Observe that such a two-way list requires only one list pointer variable START, which points to the header node. This is because the two pointers in the header node point to the two ends of the list.

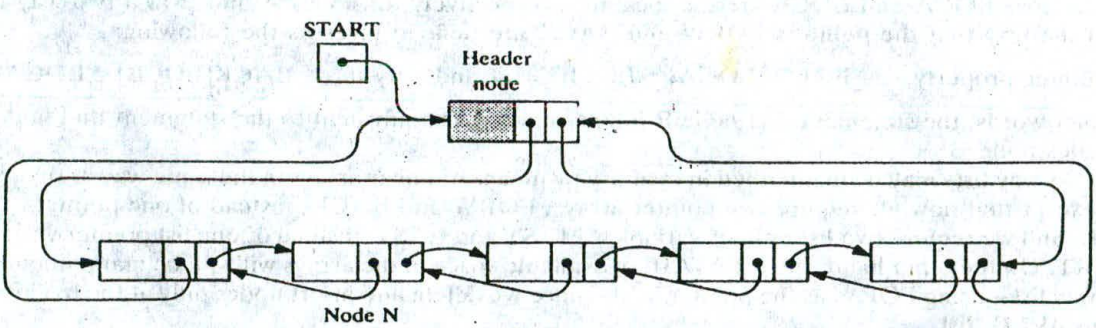


Fig. 5-35 Two-way circular header list.

EXAMPLE 5.22

Consider the personnel file in Fig. 5-30, which is organized as a circular header list. The data may be organized into a two-way circular header list by simply adding another array BACK which gives the locations of preceding nodes. Such a structure is pictured in Fig. 5-36, where LINK has been renamed FORW. Again the AVAIL list is maintained only as a one-way list.

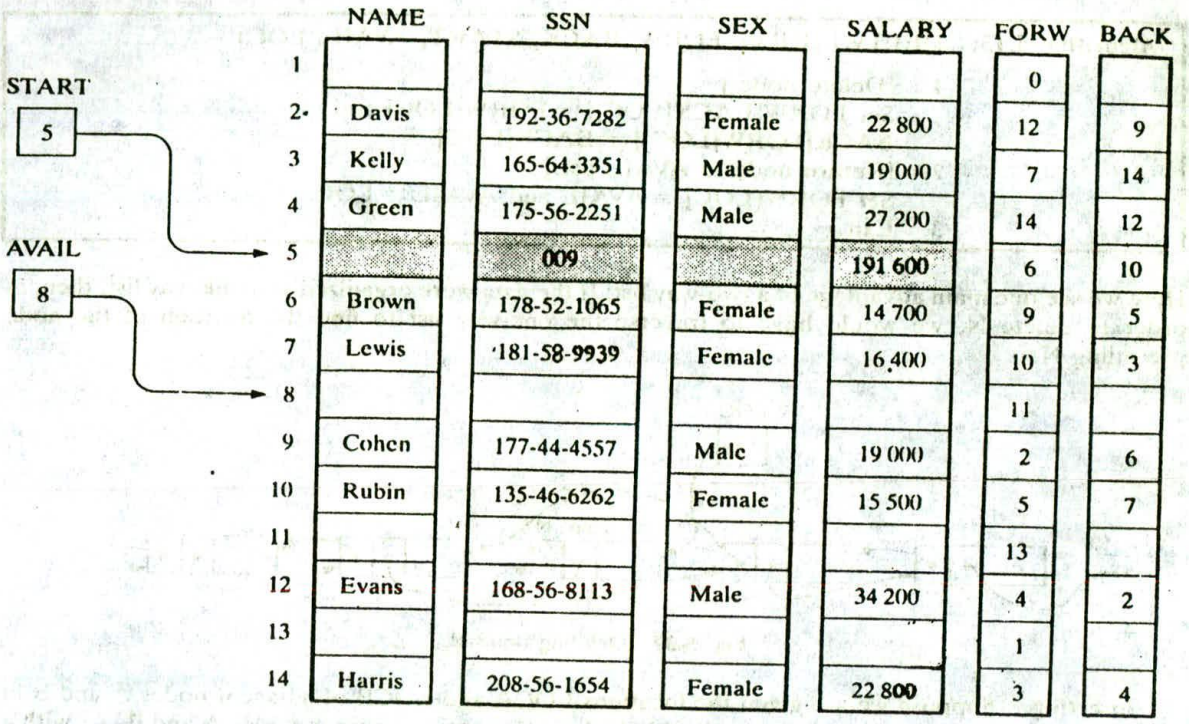


Fig. 5-36

Operations on Two-Way Lists

Suppose LIST is a two-way list in memory. This subsection discusses a number of operations on LIST.

Traversing. Suppose we want to traverse LIST in order to process each node exactly once. Then we can use Algorithm 5.1 if LIST is an ordinary two-way list, or we can use Algorithm 5.11 if LIST contains a header node. Here it is of no advantage that the data are organized as a two-way list rather than as a one-way list.

Searching. Suppose we are given an ITEM of information—a key value—and we want to find the location LOC of ITEM in LIST. Then we can use Algorithm 5.2 if LIST is an ordinary two-way list, or we can use Algorithm 5.12 if LIST has a header node. Here the main advantage is that we can search for ITEM in the backward direction if we have reason to suspect that ITEM appears near the end of the list. For example, suppose LIST is a list of names sorted alphabetically. If ITEM = Smith, then we would search LIST in the backward direction, but if ITEM = Davis, then we would search LIST in the forward direction.

Deleting. Suppose we are given the location LOC of a node N in LIST, and suppose we want to delete N from the list. We assume that LIST is a two-way circular header list. Note that BACK[LOC] and FORW[LOC] are the locations, respectively, of the nodes which precede and follow node N. Accordingly, as pictured in Fig. 5-37, N is deleted from the list by changing the following pair of pointers:

FORW[BACK[LOC]] := FORW[LOC] and BACK[FORW[LOC]] := BACK[LOC]

The deleted node N is then returned to the AVAIL list by the assignments:

FORW[LOC] := AVAIL and AVAIL := LOC

The formal statement of the algorithm follows.

Algorithm 5.15: DELTWL(INFO, FORW, BACK, START, AVAIL, LOC)

1. [Delete node.]
   Set FORW[BACK[LOC]] := FORW[LOC] and
   BACK[FORW[LOC]] := BACK[LOC].
2. [Return node to AVAIL list.]
   Set FORW[LOC] := AVAIL and AVAIL := LOC.
3. Exit.

Here we see one main advantage of a two-way list: If the data were organized as a one-way list, then, in order to delete N, we would have to traverse the one-way list to find the location of the node preceding N.

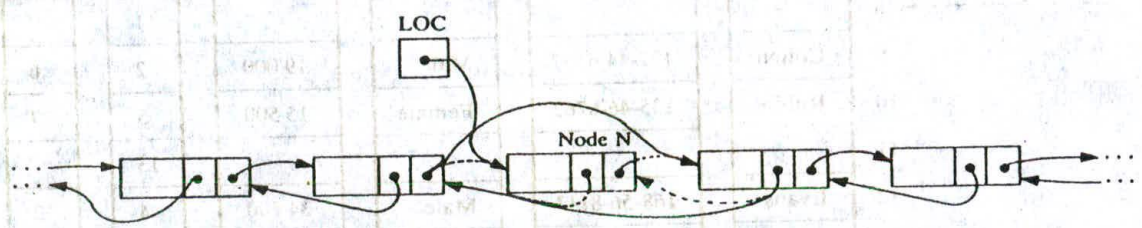


Fig. 5-37 Deleting node N.

Inserting. Suppose we are given the locations LOCA and LOCB of adjacent nodes A and B in LIST, and suppose we want to insert a given ITEM of information between nodes A and B. As with a one-way list, first we remove the first node N from the AVAIL list, using the variable NEW to keep track of its location, and then we copy the data ITEM into the node N; that is, we set:

NEW := AVAIL, AVAIL := FORW[AVAIL], INFO[NEW] := ITEM

Now, as pictured in Fig. 5-38, the node N with contents ITEM is inserted into the list by changing the following four pointers:

FORW[LOCA] := NEW, FORW[NEW] := LOCB
BACK[LOCB] := NEW, BACK[NEW] := LOCA

The formal statement of our algorithm follows.

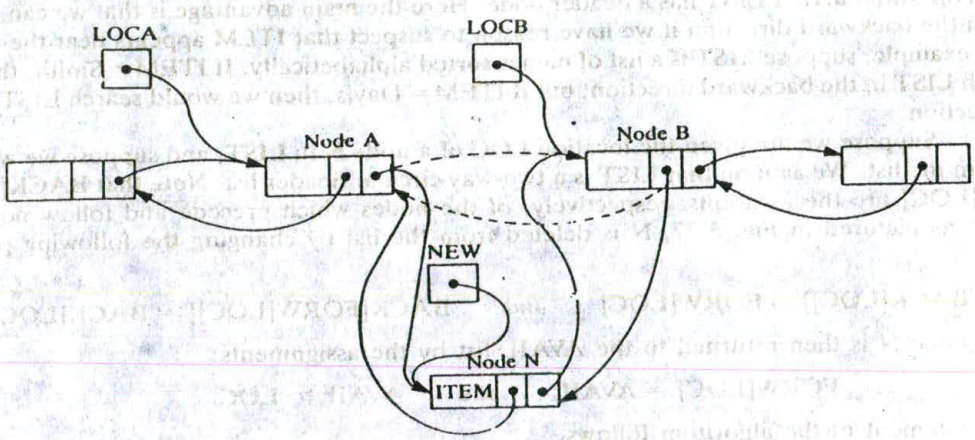


Fig. 5-38 Inserting node N.

Algorithm 5.16: INSTWL(INFO, FORW, BACK, START, AVAIL, LOCA, LOCB, ITEM)

1. [OVERFLOW?] If AVAIL = NULL, then: Write: OVERFLOW, and Exit.
2. [Remove node from AVAIL list and copy new data into node.]
   Set NEW := AVAIL, AVAIL := FORW[AVAIL], INFO[NEW] := ITEM.
3. [Insert node into list.]
   Set FORW[LOCA] := NEW, FORW[NEW] := LOCB,
   BACK[LOCB] := NEW, BACK[NEW] := LOCA.
4. Exit.

Algorithm 5.16 assumes that LIST contains a header node. Hence LOCA or LOCB may point to the header node, in which case N will be inserted as the first node or the last node. If LIST does not contain a header node, then we must consider the case that LOCA = NULL and N is inserted as the first node in the list, and the case that LOCB = NULL and N is inserted as the last node in the list.

Remark: Generally speaking, storing data as a two-way list, which requires extra space for the backward pointers and extra time to change the added pointers, rather than as a one-way list is not worth the expense unless one must frequently find the location of the node which precedes a given node N, as in the deletion above.

Solved Problems

LINKED LISTS

5.1 Find the character strings stored in the four linked lists in Fig. 5-39.

Here the four list pointers appear in an array CITY. Beginning with CITY[1], traverse the list, by following the pointers, to obtain the string PARIS. Beginning with CITY[2], traverse the list to obtain the string LONDON. Since NULL appears in CITY[3], the third list is empty, so it denotes Λ, the empty string. Beginning with CITY[4], traverse the list to obtain the string ROME. In other words, PARIS, LONDON, Λ and ROME are the four strings.

5.2 The following list of names is assigned (in order) to a linear array INFO:

Mary, June, Barbara, Paula, Diana, Audrey, Karen, Nancy, Ruth, Eileen, Sandra, Helen

That is, INFO[1] = Mary, INFO[2] = June, . . . , INFO[12] = Helen. Assign values to an array LINK and a variable START so that INFO, LINK and START form an alphabetical listing of the names.

The alphabetical listing of the names follows:

Audrey, Barbara, Diana, Eileen, Helen, June, Karen, Mary, Nancy, Paula, Ruth, Sandra

The values of START and LINK are obtained as follows:

(*a*) INFO[6] = Audrey, so assign START = 6.

(*b*) INFO[3] = Barbara, so assign LINK[6] = 3.

(*c*) INFO[5] = Diana, so assign LINK[3] = 5.

(*d*) INFO[10] = Eileen, so assign LINK[5] = 10.

And so on. Since INFO[11] = Sandra is the last name, assign LINK[11] = NULL. Figure 5-40 shows the data structure where, assuming INFO has space for only 12 elements, we set AVAIL = NULL.

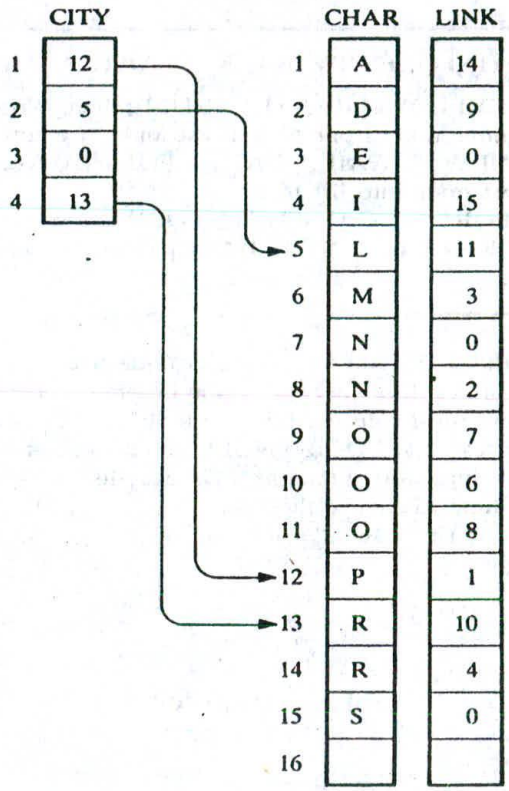


Fig. 5-39

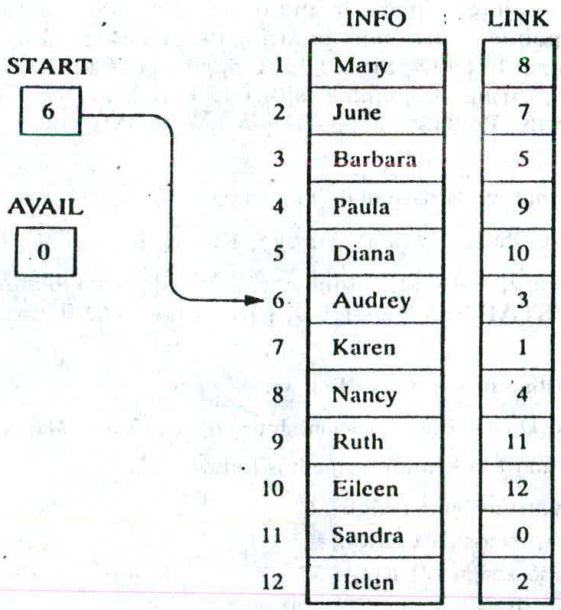


Fig. 5-40

5.3 Let LIST be a linked list in memory. Write a procedure which

(*a*) Finds the number NUM of times a given ITEM occurs in LIST

(*b*) Finds the number NUM of nonzero elements in LIST

(*c*) Adds a given value K to each element in LIST

Each procedure uses Algorithm 5.1 to traverse the list.

(*a*) **Procedure P5.3A:**
1. Set NUM := 0. [Initializes counter.]
2. Call Algorithm 5.1, replacing the processing step by:
   If INFO[PTR] = ITEM, then: Set NUM := NUM + 1.
3. Return

(*b*) **Procedure P5.3B:**
1. Set NUM := 0. [Initializes counter.]
2. Call Algorithm 5.1, replacing the processing step by:
   If INFO[PTR] ≠ 0, then: Set NUM := NUM + 1.
3. Return.

(*c*) **Procedure P5.3C:**
1. Call Algorithm 5.1, replacing the processing step by:
   Set INFO[PTR] := INFO[PTR] + K.
2. Return.

5.4 Consider the alphabetized list of patients in Fig. 5-9. Determine the changes in the data structure if (*a*) Walters is added to the list and then (*b*) Kirk is deleted from the list.

(*a*) Observe that Walters is put in bed 10, the first available bed, and Walters is inserted after Samuels, who is the last patient on the list. The three changes in the pointer fields follow:

1. LINK[9] = 10. [Now Samuels points to Walters.]
2. LINK[10] = 0. [Now Walters is the last patient in the list.]
3. AVAIL = 2. [Now AVAIL points to the next available bed.]

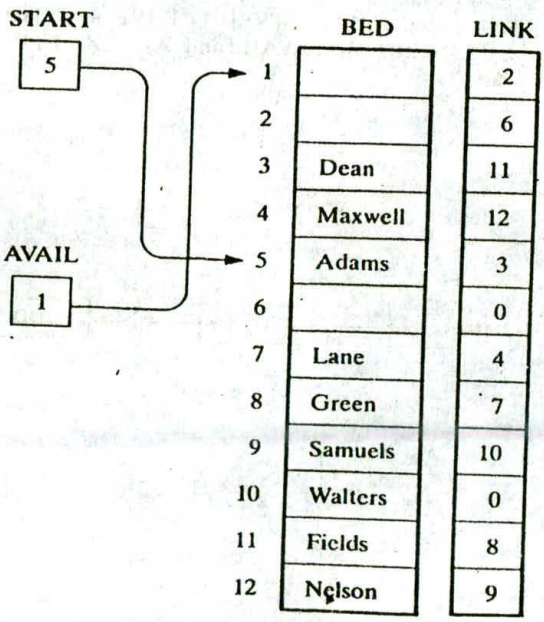


Fig. 5-41

(*b*) Since Kirk is discharged, BED[1] is now empty. The following three changes in the pointer fields must be executed:

$$\text{LINK}[8] = 7 \qquad \text{LINK}[1] = 2 \qquad \text{AVAIL} = 1$$

By the first change, Green, who originally preceded Kirk, now points to Lane, who originally followed Kirk. The second and third changes add the new empty bed to the AVAIL list. We emphasize that before making the deletion, we had to find the node BED[8], which originally pointed to the deleted node BED[1].

Figure 5-41 shows the new data structure.

5.5 Suppose LIST is in memory. Write an algorithm which deletes the last node from LIST.

The last node can be deleted only when one also knows the location of the next-to-last node. Accordingly, traverse the list using a pointer variable PTR, and keep track of the preceding node using a pointer variable SAVE. PTR points to the last node when LINK[PTR] = NULL, and in such a case, SAVE points to the next to last node. The case that LIST has only one node is treated separately, since SAVE can be defined only when the list has 2 or more elements. The algorithm follows.

Algorithm P5.5: DELLST(INFO, LINK, START, AVAIL)

1. [List empty?] If START = NULL, then Write: UNDERFLOW, and Exit.
2. [List contains only one element?]
   If LINK[START] = NULL, then:
   (*a*) Set START := NULL. [Removes only node from list.]
   (*b*) Set LINK[START] := AVAIL and AVAIL := START.
   [Returns node to AVAIL list.]
   (*c*) Exit.
   [End of If structure.]
3. Set PTR := LINK[START] and SAVE := START. [Initializes pointers.]
4. Repeat while LINK[PTR] ≠ NULL. [Traverses list, seeking last node.]
   Set SAVE := PTR and PTR := LINK[PTR]. [Updates SAVE and PTR.]
   [End of loop.]
5. Set LINK[SAVE] := LINK[PTR]. [Removes last node.]
6. Set LINK[PTR] := AVAIL and AVAIL := PTR. [Returns node to AVAIL list.]
7. Exit.

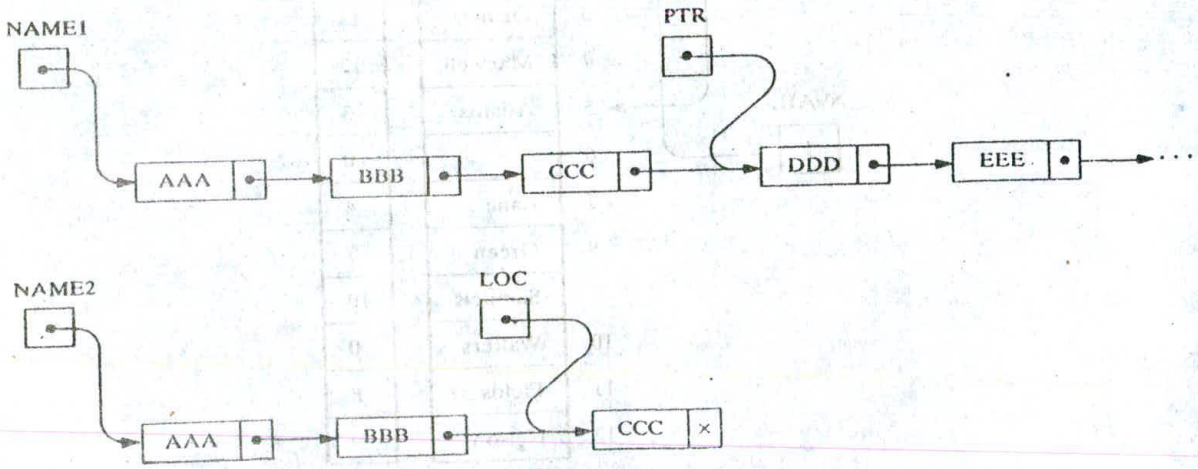


Fig. 5-42

5.6 Suppose NAME1 is a list in memory. Write an algorithm which copies NAME1 into a list NAME2.

First set NAME2 := NULL to form an empty list. Then traverse NAME1 using a pointer variable PTR, and while visiting each node of NAME1, copy its contents INFO[PTR] into a new node, which is then inserted at the end of NAME2. Use LOC to keep track of the last node of NAME2 during the traversal. (Figure 5-42 pictures PTR and LOC before the fourth node is added to NAME2.) Inserting the first node into NAME2 must be treated separateiy, since LOC is not defined until NAME2 has at least one node. The algorithm follows:

Algorithm P5.6: COPY(INFO, LINK, NAME1, NAME2, AVAIL)
This algorithm makes a copy of a list NAME1 using NAME2 as the list pointer variable of the new list.

1. Set NAME2 := NULL. [Forms empty list.]
2. [NAME1 empty?] If NAME1 = NULL, then: Exit.
3. [Insert first node of NAME1 into NAME2.]
   Call INSLOC(INFO, LINK, NAME2, AVAIL, NULL, INFO[NAME1]) or:
   (*a*) If AVAIL = NULL, then: Write: OVERFLOW, and Exit.
   (*b*) Set NEW := AVAIL and AVAIL := LINK[AVAIL]. [Removes first node from AVAIL list.]
   (*c*) Set INFO[NEW] := INFO[NAME1]. [Copies data into new node.]
   (*d*) [Insert new node as first node in NAME2.]
   Set LINK[NEW] := NAME2 and NAME2 := NEW.
4. [Initializes pointers PTR and LOC.]
   Set PTR := LINK[NAME1] and LOC := NAME2.
5. Repeat Steps 6 and 7 while PTR ≠ NULL:
6. Call INSLOC(INFO, LINK, NAME2, AVAIL, LOC, INFO[PTR]) or:
   (*a*) If AVAIL = NULL, then: Write: OVERFLOW, and Exit.
   (*b*) Set NEW := AVAIL and AVAIL := LINK[AVAIL].
   (*c*) Set INFO[NEW] := INFO[PTR]. [Copies data into new node.]
   (*d*) [Insert new node into NAME2 after the node with location LOC.]
   Set LINK[NEW] := LINK[LOC], and LINK[LOC] := NEW.
7. Set PTR := LINK[PTR] and LOC := LINK[LOC]. [Updates PTR and LOC.]
   [End of Step 5 loop.]
8. Exit.

HEADER LISTS, TWO-WAY LISTS

5.7 Form header (circular) lists from the one-way lists in Fig. 5-11.

Choose TEST[1] as a header node for the list ALG, and TEST[16] as a header node for the list GEOM. Then, for each list:

(*a*) Change the list pointer variable so that it points to the header node.

(*b*) Change the header node so that it points to the first node in the list.

(*c*) Change the last node so that it points back to the header node.

Finally reorganize the AVAIL list. Figure 5-43 shows the updated data structure.

5.8 Find the polynomials POLY1 and POLY2 stored in Fig. 5-44.

Beginning with POLY1, traverse the list by following the pointers to obtain the polynomial

$$p_1(x) = 3x^5 - 4x^3 + 6x - 5$$

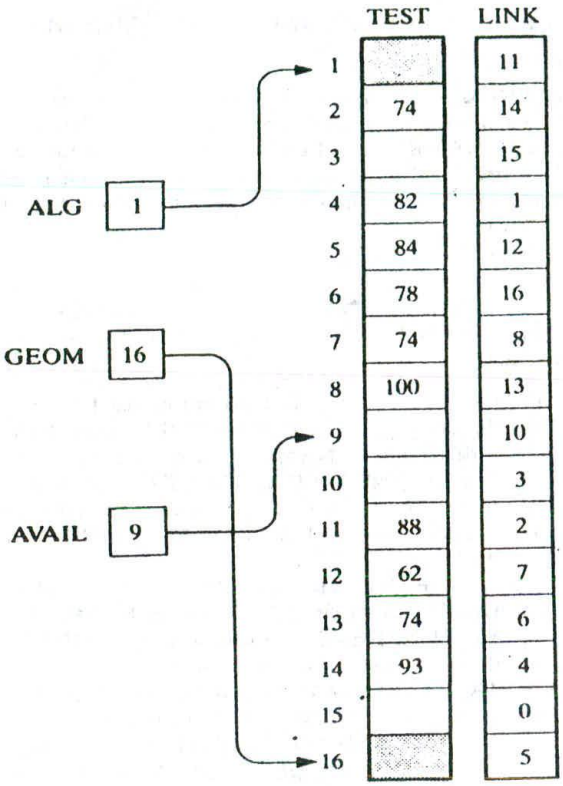


Fig. 5-43

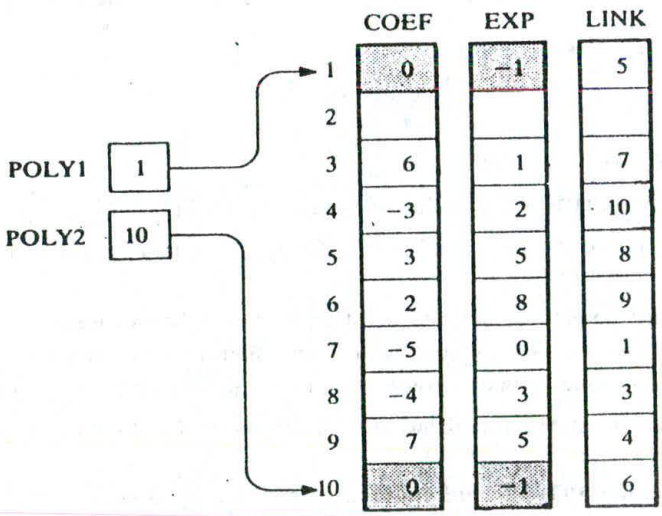


Fig. 5-44

Beginning with POLY2, traverse the list by following the pointers to obtain the polynomial

$$p_2(x) = 2x^8 + 7x^5 - 3x^2$$

Here COEF[K] and EXP[K] contain, respectively, the coefficient and exponent of a term of the polynomial. Observe that the header nodes are assigned -1 in the EXP field.

5.9 Consider a polynomial $p(x, y, z)$ in variables x, y and z. Unless otherwise stated, the terms in $p(x, y, z)$ will be ordered *lexicographically*. That is, first we order the terms according to decreasing degrees in x; those with the same degree in x we order according to decreasing degrees in y; those with the same degrees in x and y we order according to decreasing degrees in z. Suppose

$$p(x, y, z) = 8x^2y^2z - 6yz^8 + 3x^3yz + 2xy^7z - 5x^2y^3 - 4xy^7z$$

(*a*) Rewrite the polynomial so that the terms are ordered.

(*b*) Suppose the terms are stored in the order shown in the problem statement in the linear arrays COEF, XEXP, YEXP and ZEXP, with the HEAD node first. Assign values to LINK so that the linked list contains the ordered sequence of terms.

(*a*) Note that $3x^3yz$ comes first, since it has the highest degree in x. Note that $8x^2y^2z$ and $-5x^2y^3$ both have the same degree in x but $-5x^2y^3$ comes before $8x^2y^2z$, since its degree in y is higher. And so on. Finally we have

$$p(x, y, z) = 3x^3yz - 5x^2y^3 + 8x^2y^2z - 4xy^7z^3 + 2xy^7z - 6yz^8$$

(*b*) Figure 5-45 shows the desired data structure.

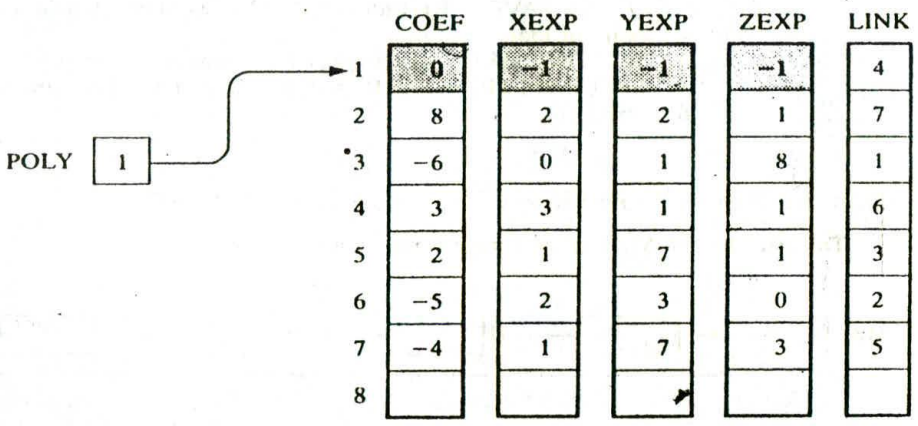


Fig. 5-45

5.10 Discuss the advantages, if any, of a two-way list over a one-way list for each of the following operations:

(*a*) Traversing the list to process each node

(*b*) Deleting a node whose location LOC is given

(*c*) Searching an unsorted list for a given element ITEM

(*d*) Searching a sorted list for a given element ITEM

(*e*) Inserting a node before the node with a given location LOC

(*f*) Inserting a node after the node with a given location LOC

(*a*) There is no advantage.

(*b*) The location of the preceding node is needed. The two-way list contains this information, whereas with a one-way list we must traverse the list.

(*c*) There is no advantage.

(*d*) There is no advantage unless we know that ITEM must appear at the end of the list, in which case we traverse the list backward. For example, if we are searching for Walker in an alphabetical listing, it may be quicker to traverse the list backward.

(*e*) As in part (*b*), the two-way list is more efficient.

(*f*) There is no advantage.

Remark: Generally speaking, a two-way list is not much more useful than a one-way list except in special circumstances.

5.11 Suppose LIST is a header (circular) list in memory. Write an algorithm which deletes the last node from LIST. (Compare with Prob. 5.5.)

The algorithm is the same as Algorithm P5.5, except now we can omit the special case when LIST has only one node. That is, we can immediately define SAVE when LIST is not empty.

Algorithm P5.11: DELLSTH(INFO, LINK, START, AVAIL)
This algorithm deletes the last node from the header list.

1. [List empty?] If LINK[START] = NULL, then: Write: UNDERFLOW, and Exit.
2. Set PTR := LINK[START] and SAVE := START. [Initializes pointers.]
3. Repeat while LINK[PTR] ≠ START: [Traverses list seeking last node.]
   Set SAVE := PTR and PTR := LINK[PTR]. [Updates SAVE and PTR.]
   [End of loop.]
4. Set LINK[SAVE] := LINK[PTR]. [Removes last node.]
5. Set LINK[PTR] := AVAIL and AVAIL := PTR. [Returns node to AVAIL list.]
6. Exit.

5.12 Form two-way lists from the one-way header lists in Fig. 5-43.

Traverse the list ALG in the forward direction to obtain:

We require the backward pointers. These are calculated node by node. For example, the last node (with location LOC = 4) must point to the next-to-last node (with location LOC = 14). Hence

$$BACK[4] = 14$$

The next-to-last node (with location LOC = 14) must point to the preceding node (with location LOC = 2). Hence

$$BACK[14] = 2$$

And so on. The header node (with location LOC = 1) must point to the last node (with location 4). Hence

$$BACK[1] = 4$$

A similar procedure is done with the list GEOM. Figure 5-46 pictures the two-way lists. Note that there is no difference between the arrays LINK and FORW. That is, only the array BACK need be calculated.

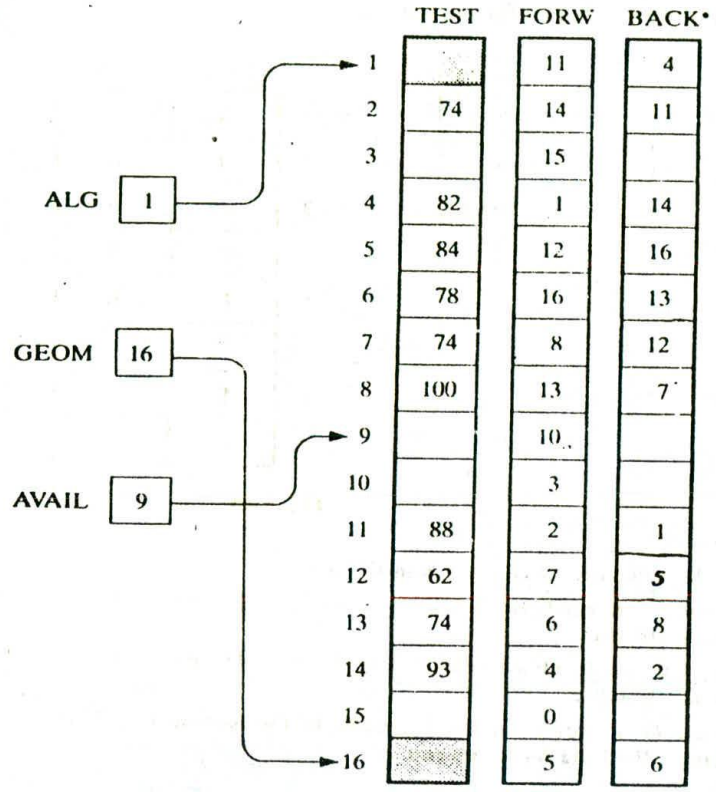
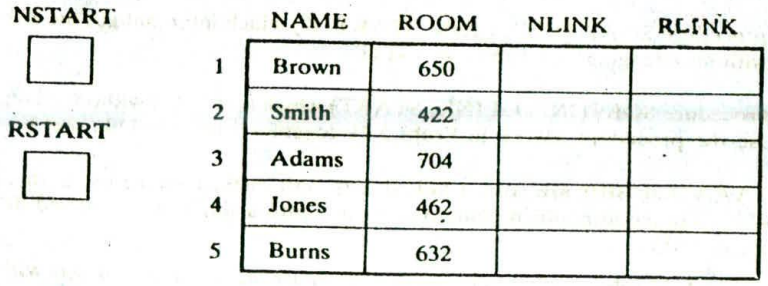


Fig. 5-46

Supplementary Problems

LINKED LISTS

5.13 Figure 5-47 is a list of five hospital patients and their room numbers. (*a*) Fill in values for NSTART and NLINK so that they form an alphabetical listing of the names. (*b*) Fill in values for RSTART and RLINK so that they form an ordering of the room numbers.



	NAME	ROOM	NLINK	RLINK
1	Brown	650		
2	Smith	422		
3	Adams	704		
4	Jones	462		
5	Burns	632		

Fig. 5-47

5.14 Figure 5-48 pictures a linked list in memory.

START

4

AVAIL

3

	INFO	LINK
1	A	2
2	B	8
3		6
4	C	7
5	D	0
6		0
7	E	1
8	F	5

Fig. 5-48

(*a*) Find the sequence of characters in the list.

(*b*) Suppose F and then C are deleted from the list and then G is inserted at the beginning of the list. Find the final structure.

(*c*) Suppose C and then F are deleted from the list and then G is inserted at the beginning of the list. Find the final structure.

(*d*) Suppose G is inserted at the beginning of the list and then F and then C are deleted from the structure. Find the final structure.

5.15 Suppose LIST is a linked list in memory consisting of numerical values. Write a procedure for each of the following:

(*a*) Finding the maximum MAX of the values in LIST

(*b*) Finding the average MEAN of the values in LIST

(*c*) Finding the product PROD of the elements in LIST

5.16 Given an integer K, write a procedure which deletes the Kth element from a linked list.

5.17 Write a procedure which adds a given ITEM of information at the end of a list.

5.18 Write a procedure which removes the first element of a list and adds it to the end of the list without changing any values in INFO. (Only START and LINK may be changed.)

5.19 Write a procedure SWAP(INFO, LINK, START, K) which interchanges the Kth and K + 1st elements in the list without changing any values in INFO.

5.20 Write a procedure SORT(INFO, LINK, START) which sorts a list without changing any values in INFO. (*Hint*: Use the procedure SWAP in Prob. 5.19 together with a bubble sort.)

5.21 Suppose AAA and BBB are sorted linked lists with distinct elements, both maintained in INFO and LINK. Write a procedure which combines the lists into a single sorted linked list CCC without changing any values in INFO.

Problems 5.22 to 5.24 refer to character strings which are stored as linked lists with one character per node and use the same arrays INFO and LINK.

5.22 Suppose STRING is a character string in memory.

(*a*) Write a procedure which prints SUBSTRING(STRING, K, N), which is the substring of STRING beginning with the Kth character and of length N.

(*b*) Write a procedure which creates a new string SUBKN in memory where

SUBKN = SUBSTRING(STRING, K, N)

5.23 Suppose STR1 and STR2 are character strings in memory. Write a procedure which creates a new string STR3 which is the concatenation of STR1 and STR2.

5.24 Suppose TEXT and PATTERN are strings in memory. Write a procedure which finds the value of INDEX(TEXT, PATTERN), the position where PATTERN first occurs as a substring of TEXT.

HEADER LISTS; TWO-WAY LISTS

5.25 Character strings are stored in the three linked lists in Fig. 5-49. (*a*) Find the three strings. (*b*) Form circular header lists from the one-way lists using CHAR[20], CHAR[19] and CHAR[18] as header nodes.

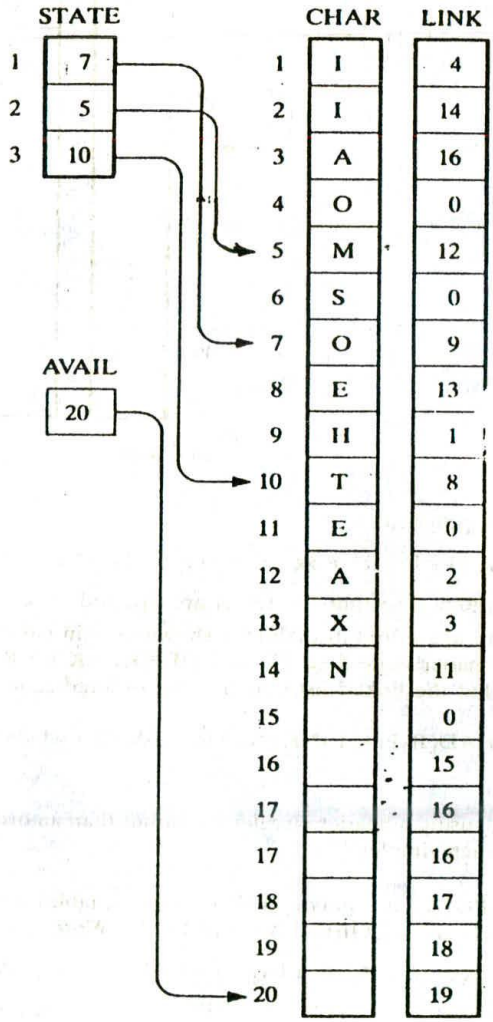


Fig. 5-49

5.26 Find the polynomials stored in the three header lists in Fig. 5-50.

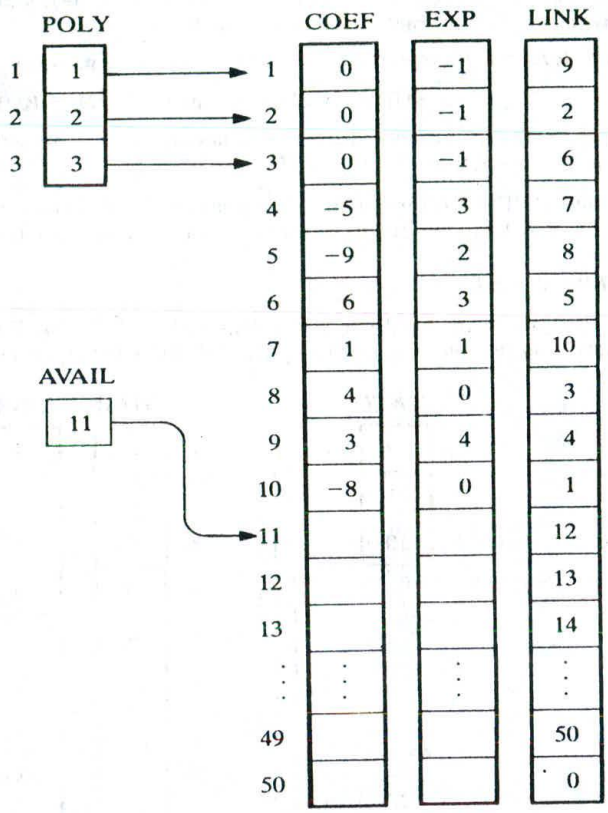


Fig. 5-50

5.27 Consider the following polynomial:

$$p(x, y, z) = 2xy^2z^3 + 3x^2yz^2 + 4xy^3z + 5x^2y^2 + 6y^3z + 7x^3z + 8xy^2z^5 + 9$$

(*a*) Rewrite the polynomial so that the terms are ordered lexicographically.

(*b*) Suppose the terms are stored in the order shown here in parallel arrays COEF, XEXP, YEXP and ZEXP with the header node first. (Thus COEF[K] = K for K = 2, 3, . . . , 9.) Assign values to an array LINK so that the linked list contains the ordered sequence of terms. (See Prob. 5.9.)

5.28 Write a procedure HEAD(INFO, LINK, START, AVAIL) which forms a header circular list from an ordinary one-way list.

5.29 Redo Probs. 5.16–5.20 using a header circular list rather than an ordinary one-way list. (Observe that the algorithms are now much simpler.)

5.30 Suppose POLY1 and POLY2 are polynomials (in one variable) which are stored as header circular lists using the same parallel arrays COEF, EXP and LINK. Write a procedure

ADD(COEF, EXP, LINK, POLY1, POLY2, AVAIL, SUMPOLY)

which finds the sum SUMPOLY of POLY1 and POLY2 (and which is also stored in memory using COEF, EXP and LINK).

5.31 For the polynomials POLY1 and POLY2 in Prob. 5.30, write a procedure

MULT(COEF, EXP, LINK, POLY1, POLY2, AVAIL, PRODPOLY)

which finds the product PRODPOLY of the polynomials POLY1 and POLY2.

5.32 Form two-way circular header lists from the one-way lists in Fig. 5-49 using, as in Prob. 5.25, CHAR[20], CHAR[19] and CHAR[18] as header nodes.

5.33 Given an integer K, write a procedure

DELK(INFO, FORW, BACK, START, AVAIL, K)

which deletes the Kth element from a two-way circular header list.

5.34 Suppose LIST(INFO, LINK, START, AVAIL) is a one-way circular header list in memory. Write a procedure

TWOWAY(INFO, LINK, BACK, START)

which assigns values to a linear array BACK to form a two-way list from the one-way list.

Programming Problems

Problems 5.35 to 5.40 refer to the data structure in Fig. 5-51, which consists of four alphabetized lists of clients and their respective lawyers.

5.35 Write a program which reads an integer K and prints the list of clients of lawyer K. Test the program for each K.

5.36 Write a program which prints the name and lawyer of each client whose age is L or higher. Test the program using (*a*) L = 41 and (*b*) L = 48.

5.37 Write a program which reads the name LLL of a lawyer and prints the lawyer's list of clients. Test the program using (*a*) Rogers, (*b*) Baker and (*c*) Levine.

5.38 Write a program which reads the NAME of a client and prints the client's name, age and lawyer. Test the program using (*a*) Newman, (*b*) Ford, (*c*) Rivers and (*d*) Hall.

5.39 Write a program which reads the NAME of the client and deletes the client's record from the structure. Test the program using (*a*) Lewis, (*b*) Klein and (*c*) Parker.

5.40 Write a program which reads the record of a new client, consisting of the client's name, age and lawyer, and inserts the record into the structure. Test the program using (*a*) Jones, 36, Levine; and (*b*) Olsen, 44, Nelson.

Problems 5.41 to 5.46 refer to the alphabetized list of employee records in Fig. 5-30, which are stored as a circular header list.

5.41 Write a program which prints out the entire alphabetized list of employee records.

5.42 Write a program which reads the name NNN of an employee and prints the [illegible]'s record. Test the program using (*a*) Evans, (*b*) Smith and (*c*) Lewis.

	LAWYER	POINT
1	Davis	4
2	Levine	12
3	Nelson	21
4	Rogers	8

AVAIL: 15

	CLIENT	AGE	LINK
1	Hall	35	16
2	Moss	28	13
3	Ford	47	25
4	Brown	54	22
5	Ginn	38	14
6	Pride	42	29
7			26
8	Berk	38	3
9	White	45	0
10			28
11	Todd	25	0
12	Dixon	32	24
13	Newman	46	6
14	Harris	42	30
15			7
16	Jackson	52	27
17			23
18	Roberts	40	0
19			0
20	Eisen	32	1
21	Adams	48	5
22	Cohen	36	20
23			19
24	Fisher	33	18
25	Graves	42	11
26			10
27	Parker	50	9
28			17
29	Singer	45	0
30	Lewis	28	2

Fig. 5-51

5.43 Write a program which reads the social security number SSS of an employee and prints the employee's record. Test the program using (*a*) 165-64-3351, (*b*) 136-46-6262 and (*c*) 177-44-5555.

5.44 Write a program which reads an integer K and prints the name of each male employee when K = 1 or of each female employee when K = 2. Test the program using (*a*) K = 2, (*b*) K = 5 and (*c*) K = 1.

5.45 Write a program which reads the name NNN of an employee and deletes the employee's record from the structure. Test the program using (*a*) Davis, (*b*) Jones and (*c*) Rubin.

5.46 Write a program which reads the record of a new employee and inserts the record into the file. Test the program using (*a*) Fletcher, 168-52-3388, Female, 21 000; and (*b*) Nelson, 175-32-2468, Male, 19 000.

Remark: Remember to update the header record whenever there is an insertion or a deletion.

Chapter 6

Stacks, Queues, Recursion

6.1 INTRODUCTION

The linear lists and linear arrays discussed in the previous chapters allowed one to insert and delete elements at any place in the list—at the beginning, at the end, or in the middle. There are certain frequent situations in computer science when one wants to restrict insertions and deletions so that they can take place only at the beginning or the end of the list, not in the middle. Two of the data structures that are useful in such situations are *stacks* and *queues*.

A stack is a linear structure in which items may be added or removed only at one end. Figure 6-1 pictures three everyday examples of such a structure: a stack of dishes, a stack of pennies and a stack of folded towels. Observe that an item may be added or removed only from the top of any of the stacks. This means, in particular, that the last item to be added to a stack is the first item to be removed. Accordingly, stacks are also called last-in first-out (LIFO) lists. Other names used for stacks are "piles" and "push-down lists." Although the stack may seem to be a very restricted type of data structure, it has many important applications in computer science.

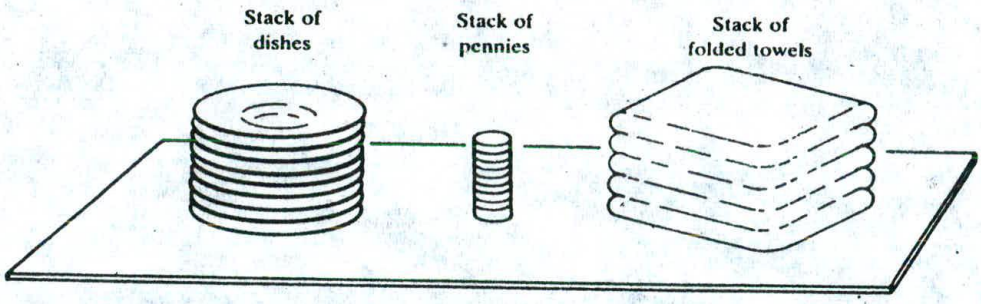


Fig. 6-1

A queue is a linear list in which items may be added only at one end and items may be removed only at the other end. The name "queue" likely comes from the everyday use of the term. Consider a queue of people waiting at a bus stop, as pictured in Fig. 6-2. Each new person who comes takes his or her place at the end of the line, and when the bus comes, the people at the front of the line board first. Clearly, the first person in the line is the first person to leave. Thus queues are also called first-in first-out (FIFO) lists. Another example of a queue is a batch of jobs waiting to be processed, assuming no job has higher priority than the others.

The notion of *recursion* is fundamental in computer science. This topic is introduced in this chapter because one way of simulating recursion is by means of a stack structure.

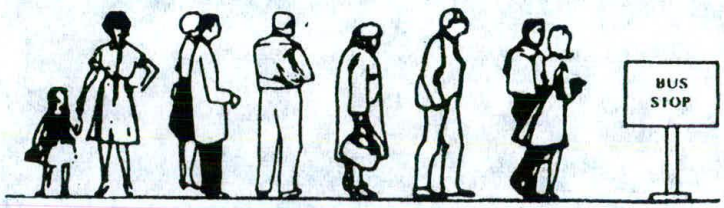


Fig. 6-2 Queue waiting for a bus.

6.2 STACKS

A *stack* is a list of elements in which an element may be inserted or deleted only at one end, called the *top* of the stack. This means, in particular, that elements are removed from a stack in the reverse order of that in which they were inserted into the stack.

Special terminology is used for two basic operations associated with stacks:

(*a*) "Push" is the term used to insert an element into a stack.

(*b*) "Pop" is the term used to delete an element from a stack.

We emphasize that these terms are used only with stacks, not with other data structures.

EXAMPLE 6.1

Suppose the following 6 elements are pushed, in order, onto an empty stack:

AAA, BBB, CCC, DDD, EEE, FFF

Figure 6-3 shows three ways of picturing such a stack. For notational convenience, we will frequently designate the stack by writing:

STACK: AAA, BBB, CCC, DDD, EEE, FFF

The implication is that the right-most element is the top element. We emphasize that, regardless of the way a stack is described, its underlying property is that insertions and deletions can occur only at the top of the stack. This means EEE cannot be deleted before FFF is deleted, DDD cannot be deleted before EEE and FFF are deleted, and so on. Consequently, the elements may be popped from the stack only in the reverse order of that in which they were pushed onto the stack.

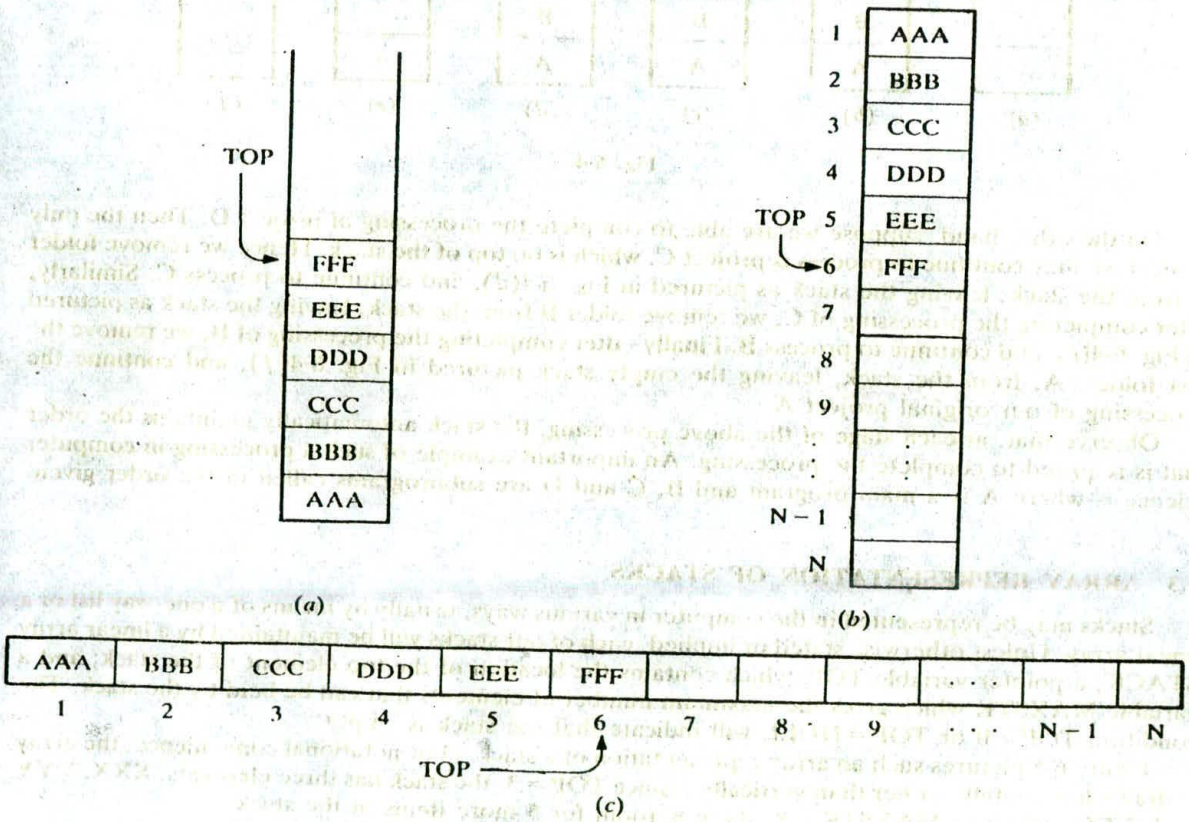


Fig. 6-3 Diagrams of stacks.

Consider again the AVAIL list of available nodes discussed in Chap. 5. Recall that free nodes were removed only from the beginning of the AVAIL list, and that new available nodes were inserted only at the beginning of the AVAIL list. In other words, the AVAIL list was implemented as a stack. This implementation of the AVAIL list as a stack is only a matter of convenience rather than an inherent part of the structure. In the following subsection we discuss an important situation where the stack is an essential tool of the processing algorithm itself.

Postponed Decisions

Stacks are frequently used to indicate the order of the processing of data when certain steps of the processing must be postponed until other conditions are fulfilled. This is illustrated as follows.

Suppose that while processing some project A we are required to move on to project B, whose completion is required in order to complete project A. Then we place the folder containing the data of A onto a stack, as pictured in Fig. 6-4(*a*), and begin to process B. However, suppose that while processing B we are led to project C, for the same reason. Then we place B on the stack above A, as pictured in Fig. 6-4(*b*), and begin to process C. Furthermore, suppose that while processing C we are likewise led to project D. Then we place C on the stack above B, as pictured in Fig. 6-4(*c*), and begin to process D.

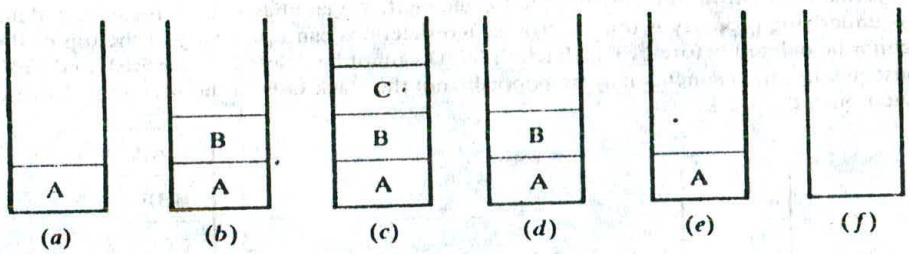


Fig. 6-4

On the other hand, suppose we are able to complete the processing of project D. Then the only project we may continue to process is project C, which is on top of the stack. Hence we remove folder C from the stack, leaving the stack as pictured in Fig. 6-4(*d*), and continue to process C. Similarly, after completing the processing of C, we remove folder B from the stack, leaving the stack as pictured in Fig. 6-4(*e*), and continue to process B. Finally, after completing the processing of B, we remove the last folder, A, from the stack, leaving the empty stack pictured in Fig. 6-4(*f*), and continue the processing of our original project A.

Observe that, at each stage of the above processing, the stack automatically maintains the order that is required to complete the processing. An important example of such a processing in computer science is where A is a main program and B, C and D are subprograms called in the order given.

6.3 ARRAY REPRESENTATION OF STACKS

Stacks may be represented in the computer in various ways, usually by means of a one-way list or a linear array. Unless otherwise stated or implied, each of our stacks will be maintained by a linear array STACK; a pointer variable TOP, which contains the location of the top element of the stack; and a variable MAXSTK which gives the maximum number of elements that can be held by the stack. The condition TOP = 0 or TOP = NULL will indicate that the stack is empty.

Figure 6-5 pictures such an array representation of a stack. (For notational convenience, the array is drawn horizontally rather than vertically.) Since TOP = 3, the stack has three elements, XXX, YYY and ZZZ; and since MAXSTK = 8, there is room for 5 more items in the stack

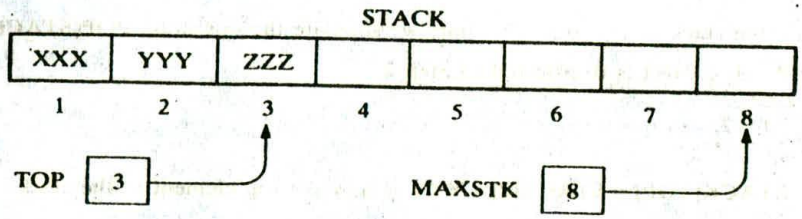


Fig. 6-5

The operation of adding (pushing) an item onto a stack and the operation of removing (popping) an item from a stack may be implemented, respectively, by the following procedures, called PUSH and POP. In executing the procedure PUSH, one must first test whether there is room in the stack for the new item; if not, then we have the condition known as overflow. Analogously, in executing the procedure POP, one must first test whether there is an element in the stack to be deleted; if not, then we have the condition known as underflow.

Procedure 6.1: PUSH(STACK, TOP, MAXSTK, ITEM)
This procedure pushes an ITEM onto a stack.

1. [Stack already filled?]
   If TOP = MAXSTK, then: Print: OVERFLOW, and Return.
2. Set TOP := TOP + 1. [Increases TOP by 1.]
3. Set STACK[TOP] := ITEM. [Inserts ITEM in new TOP position.]
4. Return.

Procedure 6.2: POP(STACK, TOP, ITEM)
This procedure deletes the top element of STACK and assigns it to the variable ITEM.

1. [Stack has an item to be removed?]
   If TOP = 0, then: Print: UNDERFLOW, and Return.
2. Set ITEM := STACK[TOP]. [Assigns TOP element to ITEM.]
3. Set TOP := TOP − 1. [Decreases TOP by 1.]
4. Return.

Frequently, TOP and MAXSTK are global variables; hence the procedures may be called using only

PUSH(STACK, ITEM) and POP(STACK, ITEM)

respectively. We note that the value of TOP is changed before the insertion in PUSH but the value of TOP is changed after the deletion in POP.

EXAMPLE 6.2

(*a*) Consider the stack in Fig. 6-5. We simulate the operation PUSH(STACK, WWW):

1. Since TOP = 3, control is transferred to Step 2.
2. TOP = 3 + 1 = 4.
3. STACK[TOP] = STACK[4] = WWW.
4. Return.

Note that WWW is now the top element in the stack.

(*b*) Consider again the stack in Fig. 6-5. This time we simulate the operation POP(STACK, ITEM):

1. Since TOP = 3, control is transferred to Step 2.
2. ITEM = ZZZ.
3. TOP = 3 − 1 = 2.
4. Return.

Observe that STACK[TOP] = STACK[2] = YYY is now the top element in the stack.

Minimizing Overflow

There is an essential difference between underflow and overflow in dealing with stacks. Underflow depends exclusively upon the given algorithm and the given input data, and hence there is no direct control by the programmer. Overflow, on the other hand, depends upon the arbitrary choice of the programmer for the amount of memory space reserved for each stack, and this choice does influence the number of times overflow may occur.

Generally speaking, the number of elements in a stack fluctuates as elements are added to or removed from a stack. Accordingly, the particular choice of the amount of memory for a given stack involves a time-space tradeoff. Specifically, initially reserving a great deal of space for each stack will decrease the number of times overflow may occur; however, this may be an expensive use of the space if most of the space is seldom used. On the other hand, reserving a small amount of space for each stack may increase the number of times overflow occurs; and the time required for resolving an overflow, such as by adding space to the stack, may be more expensive than the space saved.

Various techniques have been developed which modify the array representation of stacks so that the amount of space reserved for more than one stack may be more efficiently used. Most of these techniques lie beyond the scope of this text. We do illustrate one such technique in the following example.

EXAMPLE 6.3

Suppose a given algorithm requires two stacks, A and B. One can define an array STACKA with n_1 elements for stack A and an array STACKB with n_2 elements for stack B. Overflow will occur when either stack A contains more than n_1 elements or stack B contains more than n_2 elements.

Suppose instead that we define a single array STACK with $n = n_1 + n_2$ elements for stacks A and B together. As pictured in Fig. 6-6, we define STACK[1] as the bottom of stack A and let A "grow" to the right, and we define STACK[n] as the bottom of stack B and let B "grow" to the left. In this case, overflow will occur only when A and B together have more than $n = n_1 + n_2$ elements. This technique will usually decrease the number of times overflow occurs even though we have not increased the total amount of space reserved for the two stacks. In using this data structure, the operations of PUSH and POP will need to be modified.

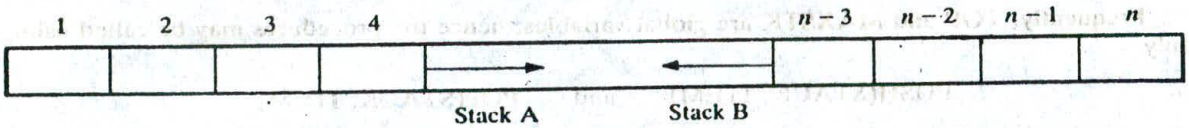


Fig. 6-6

6.4 ARITHMETIC EXPRESSIONS; POLISH NOTATION

Let Q be an arithmetic expression involving constants and operations. This section gives an algorithm which finds the value of Q by using reverse Polish (postfix) notation. We will see that the stack is an essential tool in this algorithm.

Recall that the binary operations in Q may have different levels of precedence. Specifically, we assume the following three levels of precedence for the usual five binary operations:

Highest:	Exponentiation (↑)
Next highest:	Multiplication (*) and division (/)
Lowest:	Addition (+) and subtraction (−)

(Observe that we use the BASIC symbol for exponentiation.) For simplicity, we assume that Q contains no unary operation (e.g., a leading minus sign). We also assume that in any parenthesis-free expression, the operations on the same level are performed from left to right. (This is not standard, since some languages perform exponentiations from right to left.)

EXAMPLE 6.4

Suppose we want to evaluate the following parenthesis-free arithmetic expression:

$$2 \uparrow 3 + 5 * 2 \uparrow 2 - 12 / 6$$

First we evaluate the exponentiations to obtain

$$8 + 5 * 4 - 12 / 6$$

Then we evaluate the multiplication and division to obtain $8 + 20 - 2$. Last, we evaluate the addition and subtraction to obtain the final result, 26. Observe that the expression is traversed three times, each time corresponding to a level of precedence of the operations.

Polish Notation

For most common arithmetic operations, the operator symbol is placed between its two operands. For example,

$$A + B \qquad C - D \qquad E * F \qquad G/H$$

This is called *infix notation*. With this notation, we must distinguish between

$$(A + B) * C \quad \text{and} \quad A + (B * C)$$

by using either parentheses or some operator-precedence convention such as the usual precedence levels discussed above. Accordingly, the order of the operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

Polish notation, named after the Polish mathematician Jan Lukasiewicz, refers to the notation in which the operator symbol is placed before its two operands. For example,

$$+AB \qquad -CD \qquad *EF \qquad /GH$$

We translate, step by step, the following infix expressions into Polish notation using brackets [ ] to indicate a partial translation:

$$(A + B) * C = [+AB] * C = *+ABC$$
$$A + (B * C) = A + [*BC] = +A*BC$$
$$(A + B)/(C - D) = [+AB]/[-CD] = /+AB-CD$$

The fundamental property of Polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expression. Accordingly, one never needs parentheses when writing expressions in Polish notation.

Reverse Polish notation refers to the analogous notation in which the operator symbol is placed after its two operands:

$$AB+ \qquad CD- \qquad EF* \qquad GH/$$

Again, one never needs parentheses to determine the order of the operations in any arithmetic expression written in reverse Polish notation. This notation is frequently called *postfix* (or *suffix*) *notation*, whereas *prefix notation* is the term used for Polish notation, discussed in the preceding paragraph.

The computer usually evaluates an arithmetic expression written in infix notation in two steps. First, it converts the expression to postfix notation, and then it evaluates the postfix expression. In each step, the stack is the main tool that is used to accomplish the given task. We illustrate these applications of stacks in reverse order. That is, first we show how stacks are used to evaluate postfix expressions, and then we show how stacks are used to transform infix expressions into postfix expressions.

Evaluation of a Postfix Expression

Suppose P is an arithmetic expression written in postfix notation. The following algorithm, which uses a STACK to hold operands, evaluates P.

Algorithm 6.3: This algorithm finds the VALUE of an arithmetic expression P written in postfix notation.

1. Add a right parenthesis ")" at the end of P. [This acts as a sentinel.]
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel ")" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator $\otimes$ is encountered, then:
   (*a*) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.
   (*b*) Evaluate $B \otimes A$.
   (*c*) Place the result of (*b*) back on STACK.
   [End of If structure.]
   [End of Step 2 loop.]
5. Set VALUE equal to the top element on STACK.
6. Exit.

We note that, when Step 5 is executed, there should be only one number on STACK.

EXAMPLE 6.5

Consider the following arithmetic expression P written in postfix notation:

P: 5, 6, 2, +, *, 12, 4, /, −

(Commas are used to separate the elements of P so that 5, 6, 2 is not interpreted as the number 562.) The

Symbol Scanned		STACK
(1)	5	5
(2)	6	5, 6
(3)	2	5, 6, 2
(4)	+	5, 8
(5)	*	40
(6)	12	40, 12
(7)	4	40, 12, 4
(8)	/	40, 3
(9)	−	37
(10)	)	

Fig. 6-7

equivalent infix expression Q follows:

Q: 5 * (6 + 2) − 12 / 4

Note that parentheses are necessary for the infix expression Q but not for the postfix expression P.

We evaluate P by simulating Algorithm 6.3. First we add a sentinel right parenthesis at the end of P to obtain

P:	5,	6,	2,	+,	*,	12,	4,	/,	−,	)
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)

The elements of P have been labeled from left to right for easy reference. Figure 6-7 shows the contents of STACK as each element of P is scanned. The final number in STACK, 37, which is assigned to VALUE when the sentinel ")" is scanned, is the value of P.

Transforming Infix Expressions into Postfix Expressions

Let Q be an arithmetic expression written in infix notation. Besides operands and operators, Q may also contain left and right parentheses. We assume that the operators in Q consist only of exponentiations (↑), multiplications (*), divisions (/), additions (+) and subtractions (−), and that they have the usual three levels of precedence as given above. We also assume that operators on the same level, including exponentiations, are performed from left to right unless otherwise indicated by parentheses. (This is not standard, since expressions may contain unary operators and some languages perform the exponentiations from right to left. However, these assumptions simplify our algorithm.)

The following algorithm transforms the infix expression Q into its equivalent postfix expression P. The algorithm uses a stack to temporarily hold operators and left parentheses. The postfix expression P will be constructed from left to right using the operands from Q and the operators which are removed from STACK. We begin by pushing a left parenthesis onto STACK and adding a right parenthesis at the end of Q. The algorithm is completed when STACK is empty.

Algorithm 6.4: POLISH(Q, P)
Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1. Push "(" onto STACK, and add ")" to the end of Q.
2. Scan Q from left to right and repeat Steps 3 to 6 for each element of Q until the STACK is empty:
3. If an operand is encountered, add it to P.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator ⊗ is encountered, then:
   (*a*) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than ⊗.
   (*b*) Add ⊗ to STACK.
   [End of If structure.]
6. If a right parenthesis is encountered, then:
   (*a*) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.
   (*b*) Remove the left parenthesis. [Do not add the left parenthesis to P.]
   [End of If structure.]
   [End of Step 2 loop.]
7. Exit.

The terminology sometimes used for Step 5 is that ⊗ will "sink" to its own level.

EXAMPLE 6.6

Consider the following arithmetic infix expression Q:

Q: A + (B * C − (D / E ↑ F) * G) * H

We simulate Algorithm 6.4 to transform Q into its equivalent postfix expression P.
First we push "(" onto STACK, and then we add ")" to the end of Q to obtain:

Q:	A	+	(	B	*	C	−	(	D	/	E	↑	F	)	*	G	)	*	H	)
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)	(19)	(20)

The elements of Q have now been labeled from left to right for easy reference. Figure 6-8 shows the status of STACK and of the string P as each element of Q is scanned. Observe that

(1) Each operand is simply added to P and does not change STACK.

(2) The subtraction operator (−) in row 7 sends * from STACK to P before it (−) is pushed onto STACK.

(3) The right parenthesis in row 14 sends ↑ and then / from STACK to P, and then removes the left parenthesis from the top of STACK.

(4) The right parenthesis in row 20 sends * and then + from STACK to P, and then removes the left parenthesis from the top of STACK.

After Step 20 is executed, the STACK is empty and

P: A B C * D E F ↑ / G * − H * +

which is the required postfix equivalent of Q.

Symbol Scanned	STACK	Expression P
(1) A	(	A
(2) +	(+	A
(3) (	(+ (	A
(4) B	(+ (	A B
(5) *	(+ (*	A B
(6) C	(+ (*	A B C
(7) −	(+ (−	A B C *
(8) (	(+ (− (	A B C *
(9) D	(+ (− (	A B C * D
(10) /	(+ (− (/	A B C * D
(11) E	(+ (− (/	A B C * D E
(12) ↑	(+ (− (/ ↑	A B C * D E
(13) F	(+ (− (/ ↑	A B C * D E F
(14))	(+ (−	A B C * D E F ↑ /
(15) *	(+ (− *	A B C * D E F ↑ /
(16) G	(+ (− *	A B C * D E F ↑ / G
(17))	(+	A B C * D E F ↑ / G * −
(18) *	(+ *	A B C * D E F ↑ / G * −
(19) H	(+ *	A B C * D E F ↑ / G * − H
(20))		A B C * D E F ↑ / G * − H * +

Fig. 6-8

6.5 QUICKSORT, AN APPLICATION OF STACKS

Let A be a list of n data items. "Sorting A" refers to the operation of rearranging the elements of A so that they are in some logical order, such as numerically ordered when A contains numerical data, or alphabetically ordered when A contains character data. The subject of sorting, including various sorting algorithms, is treated mainly in Chap. 9. This section gives only one sorting algorithm, called *quicksort*, in order to illustrate an application of stacks.

Quicksort is an algorithm of the divide-and-conquer type. That is, the problem of sorting a set is reduced to the problem of sorting two smaller sets. We illustrate this "reduction step" by means of a specific example.

Suppose A is the following list of 12 numbers:

(44), 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, (66)

The reduction step of the quicksort algorithm finds the final position of one of the numbers; in this illustration, we use the first number, 44. This is accomplished as follows. Beginning with the last number, 66, scan the list from right to left, comparing each number with 44 and stopping at the first number less than 44. The number is 22. Interchange 44 and 22 to obtain the list

(22), 33, 11, 55, 77, 90, 40, 60, 99, (44), 88, 66

(Observe that the numbers 88 and 66 to the right of 44 are each greater than 44.) Beginning with 22, next scan the list in the opposite direction, from left to right, comparing each number with 44 and stopping at the first number greater than 44. The number is 55. Interchange 44 and 55 to obtain the list

22, 33, 11, (44), 77, 90, 40, 60, 99, (55), 88, 66

(Observe that the numbers 22, 33 and 11 to the left of 44 are each less than 44.) Beginning this time with 55, now scan the list in the original direction, from right to left, until meeting the first number less than 44. It is 40. Interchange 44 and 40 to obtain the list

22, 33, 11, (40), 77, 90, (44), 60, 99, 55, 88, 66

(Again, the numbers to the right of 44 are each greater than 44.) Beginning with 40, scan the list from left to right. The first number greater than 44 is 77. Interchange 44 and 77 to obtain the list

22, 33, 11, 40, (44), 90, (77), 60, 99, 55, 88, 66

(Again, the numbers to the left of 44 are each less than 44.) Beginning with 77, scan the list from right to left seeking a number less than 44. We do not meet such a number before meeting 44. This means all numbers have been scanned and compared with 44. Furthermore, all numbers less than 44 now form the sublist of numbers to the left of 44, and all numbers greater than 44 now form the sublist of numbers to the right of 44, as shown below:

22, 33, 11, 40, (44), 90, 77, 60, 99, 55, 88, 66

First sublist Second sublist

Thus 44 is correctly placed in its final position, and the task of sorting the original list A has now been reduced to the task of sorting each of the above sublists.

The above reduction step is repeated with each sublist containing 2 or more elements. Since we can process only one sublist at a time, we must be able to keep track of some sublists for future processing. This is accomplished by using two stacks, called LOWER and UPPER, to temporarily "hold" such

sublists. That is, the addresses of the first and last elements of each sublist, called its *boundary values*, are pushed onto the stacks LOWER and UPPER, respectively; and the reduction step is applied to a sublist only after its boundary values are removed from the stacks. The following example illustrates the way the stacks LOWER and UPPER are used.

EXAMPLE 6.7

Consider the above list A with $n = 12$ elements. The algorithm begins by pushing the boundary values 1 and 12 of A onto the stacks to yield

LOWER: 1 UPPER: 12

In order to apply the reduction step, the algorithm first removes the top values 1 and 12 from the stacks, leaving

LOWER: (empty) UPPER: (empty)

and then applies the reduction step to the corresponding list A[1], A[2], . . . , A[12]. The reduction step, as executed above, finally places the first element, 44, in A[5]. Accordingly, the algorithm pushes the boundary values 1 and 4 of the first sublist and the boundary values 6 and 12 of the second sublist onto the stacks to yield

LOWER: 1, 6 UPPER: 4, 12

In order to apply the reduction step again, the algorithm removes the top values, 6 and 12, from the stacks, leaving

LOWER: 1 UPPER: 4

and then applies the reduction step to the corresponding sublist A[6], A[7], . . . , A[12]. The reduction step changes this list as in Fig. 6-9. Observe that the second sublist has only one element. Accordingly, the algorithm pushes only the boundary values 6 and 10 of the first sublist onto the stacks to yield

LOWER: 1, 6 UPPER: 4, 10

And so on. The algorithm ends when the stacks do not contain any sublist to be processed by the reduction step.

A[6],	A[7],	A[8],	A[9],	A[10],	A[11],	A[12],
(90),	77,	60,	99,	55,	88,	(66)
66,	77,	60,	(99),	55,	88,	(90)
66,	77,	60,	(90),	55,	(88),	99
66,	77,	60,	88,	55,	(90),	99
			First sublist		Second sublist	

Fig. 6-9

The formal statement of our quicksort algorithm follows (on page 175). For notational convenience and pedagogical considerations, the algorithm is divided into two parts. The first part gives a procedure, called QUICK, which executes the above reduction step of the algorithm, and the second part uses QUICK to sort the entire list.

Observe that Step 2(*c*) (iii) is unnecessary. It has been added to emphasize the symmetry between Step 2 and Step 3. The procedure does not assume the elements of A are distinct. Otherwise, the condition LOC ≠ RIGHT in Step 2(*a*) and the condition LEFT ≠ LOC in Step 3(*a*) could be omitted.

The second part of the algorithm follows (on page 175). As noted above, LOWER and UPPER are stacks on which the boundary values of the sublists are stored. (As usual, we use NULL = 0.)

Procedure 6.5: QUICK(A, N, BEG, END, LOC)

Here A is an array with N elements. Parameters BEG and END contain the boundary values of the sublist of A to which this procedure applies. LOC keeps track of the position of the first element A[BEG] of the sublist during the procedure. The local variables LEFT and RIGHT will contain the boundary values of the list of elements that have not been scanned.

1. [Initialize.] Set LEFT := BEG, RIGHT := END and LOC := BEG.
2. [Scan from right to left.]
   (*a*) Repeat while A[LOC] ≤ A[RIGHT] and LOC ≠ RIGHT:
   RIGHT := RIGHT − 1.
   [End of loop.]
   (*b*) If LOC = RIGHT, then: Return.
   (*c*) If A[LOC] > A[RIGHT], then:
   (i) [Interchange A[LOC] and A[RIGHT].]
   TEMP := A[LOC], A[LOC] := A[RIGHT],
   A[RIGHT] := TEMP.
   (ii) Set LOC := RIGHT.
   (iii) Go to Step 3.
   [End of If structure.]
3. [Scan from left to right.]
   (*a*) Repeat while A[LEFT] ≤ A[LOC] and LEFT ≠ LOC:
   LEFT := LEFT + 1.
   [End of loop.]
   (*b*) If LOC = LEFT, then: Return.
   (*c*) If A[LEFT] > A[LOC], then
   (i) [Interchange A[LEFT] and A[LOC].]
   TEMP := A[LOC], A[LOC] := A[LEFT],
   A[LEFT] := TEMP.
   (ii) Set LOC := LEFT.
   (iii) Go to Step 2.
   [End of If structure.]

Algorithm 6.6: (Quicksort) This algorithm sorts an array A with N elements.

1. [Initialize.] TOP := NULL.
2. [Push boundary values of A onto stacks when A has 2 or more elements.]
   If N > 1, then: TOP := TOP + 1, LOWER[1] := 1, UPPER[1] := N.
3. Repeat Steps 4 to 7 while TOP ≠ NULL.
4. [Pop sublist from stacks.]
   Set BEG := LOWER[TOP], END := UPPER[TOP],
   TOP := TOP − 1.
5. Call QUICK(A, N, BEG, END, LOC). [Procedure 6.5.]
6. [Push left sublist onto stacks when it has 2 or more elements.]
   If BEG < LOC − 1, then:
   TOP := TOP + 1, LOWER[TOP] := BEG,
   UPPER[TOP] = LOC − 1.
   [End of If structure.]
7. [Push right sublist onto stacks when it has 2 or more elements.]
   If LOC + 1 < END, then:
   TOP := TOP + 1, LOWER[TOP] := LOC + 1,
   UPPER[TOP] := END.
   [End of If structure.]
   [End of Step 3 loop.]
8. Exit.

Complexity of the Quicksort Algorithm

The running time of a sorting algorithm is usually measured by the number $f(n)$ of comparisons required to sort n elements. The quicksort algorithm, which has many variations, has been studied extensively. Generally speaking, the algorithm has a worst-case running time of order $n^2/2$, but an average-case running time of order $n \log n$. The reason for this is indicated below.

The worst case occurs when the list is already sorted. Then the first element will require n comparisons to recognize that it remains in the first position. Furthermore, the first sublist will be empty, but the second sublist will have $n - 1$ elements. Accordingly, the second element will require $n - 1$ comparisons to recognize that it remains in the second position. And so on. Consequently, there will be a total of

$$f(n) = n + (n-1) + \cdots + 2 + 1 = \frac{n(n+1)}{2} = \frac{n^2}{2} + O(n) = O(n^2)$$

comparisons. Observe that this is equal to the complexity of the bubble sort algorithm (Sec. 4.6).

The complexity $f(n) = O(n \log n)$ of the average case comes from the fact that, on the average, each reduction step of the algorithm produces two sublists. Accordingly:

(1) Reducing the initial list places 1 element and produces two sublists.

(2) Reducing the two sublists places 2 elements and produces four sublists.

(3) Reducing the four sublists places 4 elements and produces eight sublists.

(4) Reducing the eight sublists places 8 elements and produces sixteen sublists.

And so on. Observe that the reduction step in the kth level finds the location of 2^{k-1} elements; hence there will be approximately $\log_2 n$ levels of reductions steps. Furthermore, each level uses at most n comparisons, so $f(n) = O(n \log n)$. In fact, mathematical analysis and empirical evidence have both shown that

$$f(n) \approx 1.4\lceil n \log n \rceil$$

is the expected number of comparisons for the quicksort algorithm.

6.6 RECURSION

Recursion is an important concept in computer science. Many algorithms can be best described in terms of recursion. This section introduces this powerful tool, and Sec. 6.8 will show how recursion may be implemented by means of stacks.

Suppose P is a procedure containing either a Call statement to itself or a Call statement to a second procedure that may eventually result in a Call statement back to the original procedure P. Then P is called a *recursive procedure*. So that the program will not continue to run indefinitely, a recursive procedure must have the following two properties:

(1) There must be certain criteria, called *base criteria*, for which the procedure does not call itself.

(2) Each time the procedure does call itself (directly or indirectly), it must be closer to the base criteria.

A recursive procedure with these two properties is said to be *well-defined*.

Similarly, a function is said to be *recursively defined* if the function definition refers to itself. Again, in order for the definition not to be circular, it must have the following two properties:

(1) There must be certain arguments, called *base values*, for which the function does not refer to itself.

(2) Each time the function does refer to itself, the argument of the function must be closer to a base value.

A recursive function with these two properties is also said to be well-defined.

The following examples should help clarify these ideas.

Factorial Function

The product of the positive integers from 1 to n, inclusive, is called "n factorial" and is usually denoted by $n!$:

$$n! = 1 \cdot 2 \cdot 3 \cdots (n-2)(n-1)n$$

It is also convenient to define $0! = 1$, so that the function is defined for all nonnegative integers. Thus we have

$$0! = 1 \qquad 1! = 1 \qquad 2! = 1 \cdot 2 = 2 \qquad 3! = 1 \cdot 2 \cdot 3 = 6 \qquad 4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$$
$$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120 \qquad 6! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 = 720$$

and so on. Observe that

$$5! = 5 \cdot 4! = 5 \cdot 24 = 120 \qquad \text{and} \qquad 6! = 6 \cdot 5! = 6 \cdot 120 = 720$$

This is true for every positive integer n; that is,

$$n! = n \cdot (n-1)!$$

Accordingly, the factorial function may also be defined as follows:

Definition 6.1: (Factorial Function)

(*a*) If $n = 0$, then $n! = 1$.
(*b*) If $n > 0$, then $n! = n \cdot (n-1)!$

Observe that this definition of $n!$ is recursive, since it refers to itself when it uses $(n-1)!$ However, (*a*) the value of $n!$ is explicitly given when $n = 0$ (thus 0 is the base value); and (*b*) the value of $n!$ for arbitrary n is defined in terms of a smaller value of n which is closer to the base value 0. Accordingly, the definition is not circular, or in other words, the procedure is well-defined.

EXAMPLE 6.8

Let us calculate 4! using the recursive definition. This calculation requires the following nine steps:

(1) $4! = 4 \cdot 3!$
(2) $3! = 3 \cdot 2!$
(3) $2! = 2 \cdot 1!$
(4) $1! = 1 \cdot 0!$
(5) $0! = 1$
(6) $1! = 1 \cdot 1 = 1$
(7) $2! = 2 \cdot 1 = 2$
(8) $3! = 3 \cdot 2 = 6$
(9) $4! = 4 \cdot 6 = 24$

That is:

Step 1. This defines 4! in terms of 3!, so we must postpone evaluating 4! until we evaluate 3! This postponement is indicated by indenting the next step.

Step 2. Here 3! is defined in terms of 2!, so we must postpone evaluating 3! until we evaluate 2!

Step 3. This defines 2! in terms of 1!

Step 4. This defines 1! in terms of 0!

Step 5. This step can explicitly evaluate 0!, since 0 is the base value of the recursive definition.

Steps 6 to 9. We backtrack, using 0! to find 1!, using 1! to find 2!, using 2! to find 3!, and finally using 3! to find 4! This backtracking is indicated by the "reverse" indention.

Observe that we backtrack in the reverse order of the original postponed evaluations. Recall that this type of postponed processing lends itself to the use of stacks. (See Sec. 6.2.)

The following are two procedures that each calculate n factorial.

Procedure 6.7A: FACTORIAL(FACT, N)
This procedure calculates N! and returns the value in the variable FACT.

1. If N = 0, then: Set FACT := 1, and Return.
2. Set FACT := 1. [Initializes FACT for loop.]
3. Repeat for K = 1 to N.
   Set FACT := K * FACT.
   [End of loop.]
4. Return.

Procedure 6.7B: FACTORIAL(FACT, N)
This procedure calculates N! and returns the value in the variable FACT.

1. If N = 0, then: Set FACT := 1, and Return.
2. Call FACTORIAL(FACT, N − 1).
3. Set FACT := N * FACT.
4. Return.

Observe that the first procedure evaluates N! using an iterative loop process. The second procedure, on the other hand, is a recursive procedure, since it contains a call to itself. Some programming languages, notably FORTRAN, do not allow such recursive subprograms.

Suppose P is a recursive procedure. During the running of an algorithm or a program which contains P, we associate a *level number* with each given execution of procedure P as follows. The original execution of procedure P is assigned level 1; and each time procedure P is executed because of a recursive call, its level is 1 more than the level of the execution that has made the recursive call. In Example 6.8, Step 1 belongs to level 1. Hence Step 2 belongs to level 2, Step 3 to level 3, Step 4 to level 4 and Step 5 to level 5. On the other hand, Step 6 belongs to level 4, since it is the result of a return from level 5. In other words, Step 6 and Step 4 belong to the same level of execution. Similarly, Step 7 belongs to level 3, Step 8 to level 2, and the final step, Step 9, to the original level 1.

The *depth* of recursion of a recursive procedure P with a given set of arguments refers to the maximum level number of P during its execution.

Fibonacci Sequence

The celebrated Fibonacci sequence (usually denoted by $F_0, F_1, F_2, \ldots$) is as follows:

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

That is, $F_0 = 0$ and $F_1 = 1$ and each succeeding term is the sum of the two preceding terms. For example, the next two terms of the sequence are

$$34 + 55 = 89 \quad \text{and} \quad 55 + 89 = 144$$

A formal definition of this function follows:

Definition 6.2: (Fibonacci Sequence)

(*a*) If $n = 0$ or $n = 1$, then $F_n = n$.

(*b*) If $n > 1$, then $F_n = F_{n-2} + F_{n-1}$.

This is another example of a recursive definition, since the definition refers to itself when it uses F_{n-2} and F_{n-1}. Here (*a*) the base values are 0 and 1, and (*b*) the value of F_n is defined in terms of smaller values of n which are closer to the base values. Accordingly, this function is well-defined.

A procedure for finding the nth term F_n of the Fibonacci sequence follows.

Procedure 6.8: FIBONACCI(FIB, N)
This procedure calculates F_N and returns the value in the first parameter FIB.

1. If N = 0 or N = 1, then: Set FIB := N, and Return.
2. Call FIBONACCI(FIBA, N − 2).
3. Call FIBONACCI(FIBB, N − 1).
4. Set FIB := FIBA + FIBB.
5. Return.

This is another example of a recursive procedure, since the procedure contains a call to itself. In fact, this procedure contains two calls to itself. We note (see Prob. 6.16) that one can also write an iterative procedure to calculate F_n which does not use recursion.

Divide-and-Conquer Algorithms

Consider a problem P associated with a set S. Suppose A is an algorithm which partitions S into smaller sets such that the solution of the problem P for S is reduced to the solution of P for one or more of the smaller sets. Then A is called a divide-and-conquer algorithm.

Two examples of divide-and-conquer algorithms, previously treated, are the quicksort algorithm in Sec. 6.5 and the binary search algorithm in Sec. 4.7. Recall that the quicksort algorithm uses a reduction step to find the location of a single element and to reduce the problem of sorting the entire set to the problem of sorting smaller sets. The binary search algorithm divides the given sorted set into two halves so that the problem of searching for an item in the entire set is reduced to the problem of searching for the item in one of the two halves.

A divide-and-conquer algorithm A may be viewed as a recursive procedure. The reason for this is that the algorithm A may be viewed as calling itself when it is applied to the smaller sets. The base criteria for these algorithms are usually the one-element sets. For example, with a sorting algorithm, a one-element set is automatically sorted; and with a searching algorithm, a one-element set requires only a single comparison.

Ackermann Function

The Ackermann function is a function with two arguments each of which can be assigned any nonnegative integer: 0, 1, 2, ... This function is defined as follows:

Definition 6.3: (Ackermann Function)

(*a*) If $m = 0$, then $A(m, n) = n + 1$.

(*b*) If $m \neq 0$ but $n = 0$, then $A(m, n) = A(m - 1, 1)$.

(*c*) If $m \neq 0$ and $n \neq 0$, then $A(m, n) = A(m - 1, A(m, n - 1))$

Once more, we have a recursive definition, since the definition refers to itself in parts (*b*) and (*c*). Observe that $A(m, n)$ is explicitly given only when $m = 0$. The base criteria are the pairs

$$(0, 0), \quad (0, 1), \quad (0, 2), \quad (0, 3), \ldots, (0, n), \ldots$$

Although it is not obvious from the definition, the value of any $A(m, n)$ may eventually be expressed in terms of the value of the function on one or more of the base pairs.

The value of $A(1, 3)$ is calculated in Prob. 6.17. Even this simple case requires 15 steps. Generally speaking, the Ackermann function is too complex to evaluate on any but a trivial example. Its importance comes from its use in mathematical logic. The function is stated here mainly to give another example of a classical recursive function and to show that the recursion part of a definition may be complicated.

6.7 TOWERS OF HANOI

The preceding section gave examples of some recursive definitions and procedures. This section shows how recursion may be used as a tool in developing an algorithm to solve a particular problem. The problem we pick is known as the Towers of Hanoi problem.

Suppose three pegs, labeled A, B and C, are given, and suppose on peg A there are placed a finite number n of disks with decreasing size. This is pictured in Fig. 6-10 for the case $n = 6$. The object of the game is to move the disks from peg A to peg C using peg B as an auxiliary. The rules of the game are as follows:

(*a*) Only one disk may be moved at a time. Specifically, only the top disk on any peg may be moved to any other peg.

(*b*) At no time can a larger disk be placed on a smaller disk.

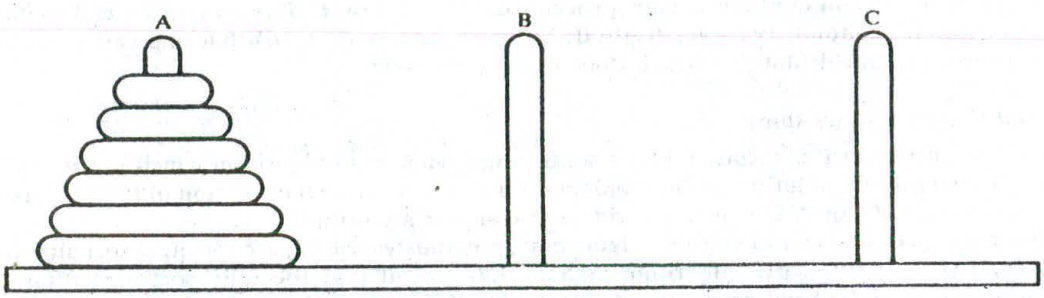


Fig. 6-10 Initial setup of Towers of Hanoi with $n = 6$.

Sometimes we will write X → Y to denote the instruction "Move top disk from peg X to peg Y," where X and Y may be any of the three pegs.

The solution to the Towers of Hanoi problem for $n = 3$ appears in Fig. 6-11. Observe that it consists of the following seven moves:

$n = 3$: Move top disk from peg A to peg C.
Move top disk from peg A to peg B.
Move top disk from peg C to peg B.
Move top disk from peg A to peg C.
Move top disk from peg B to peg A.
Move top disk from peg B to peg C.
Move top disk from peg A to peg C.

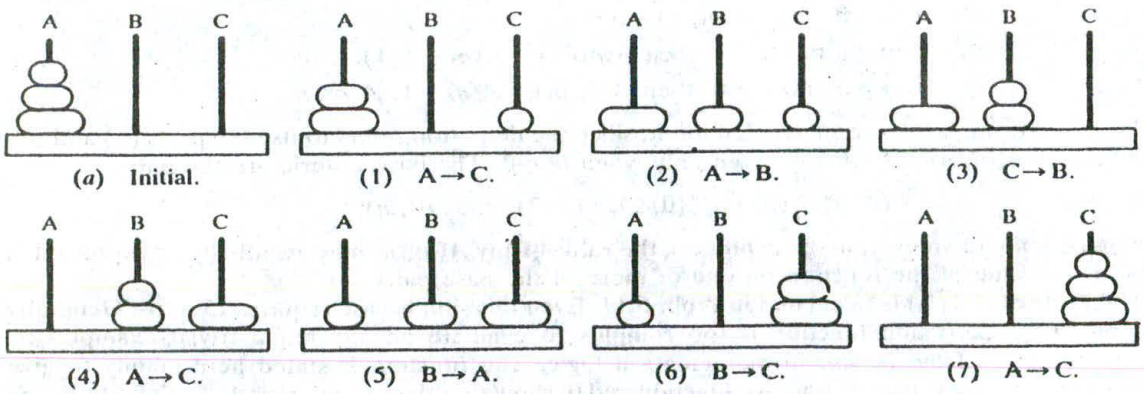


Fig. 6-11

In other words,

$n = 3$: $A \to C$, $A \to B$, $C \to B$, $A \to C$, $B \to A$, $B \to C$, $A \to C$

For completeness, we also give the solution to the Towers of Hanoi problem for $n = 1$ and $n = 2$:

$n = 1$: $A \to C$

$n = 2$: $A \to B$, $A \to C$, $B \to C$

Note that $n = 1$ uses only one move and that $n = 2$ uses three moves.

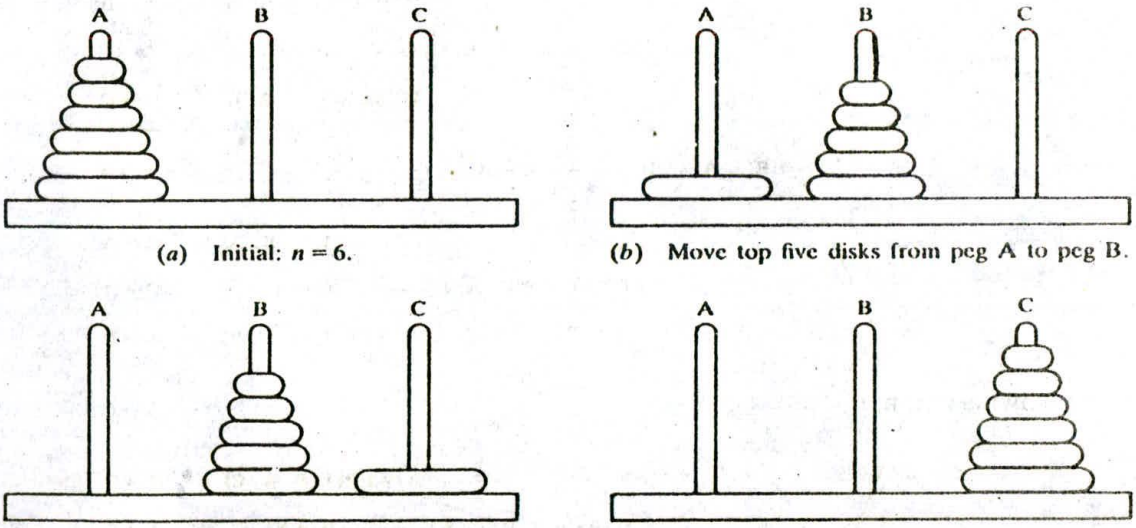
Rather than finding a separate solution for each n, we use the technique of recursion to develop a general solution. First we observe that the solution to the Towers of Hanoi problem for $n > 1$ disks may be reduced to the following subproblems:

(1) Move the top $n - 1$ disks from peg A to peg B.

(2) Move the top disk from peg A to peg C: $A \to C$.

(3) Move the top $n - 1$ disks from peg B to peg C.

This reduction is illustrated in Fig. 6-12 for $n = 6$. That is, first we move the top five disks from peg A to peg B, then we move the large disk from peg A to peg C, and then we move the top five disks from peg B to peg C.



(*a*) Initial: $n = 6$. (*b*) Move top five disks from peg A to peg B.

(*c*) Move top disk from peg A to peg C. (*d*) Move top five disks from peg B to peg C.

Fig. 6-12

Let us now introduce the general notation

TOWER(N, BEG, AUX, END)

to denote a procedure which moves the top n disks from the initial peg BEG to the final peg END using the peg AUX as an auxiliary. When $n = 1$, we have the following obvious solution:

TOWER(1, BEG, AUX, END) consists of the single instruction $BEG \to END$

Furthermore, as discussed above, when $n > 1$, the solution may be reduced to the solution of the following three subproblems:

(1) TOWER(N − 1, BEG, END, AUX)
(2) TOWER(1, BEG, AUX, END) or BEG → END
(3) TOWER(N − 1, AUX, BEG, END)

Observe that each of these three subproblems may be solved directly or is essentially the same as the original problem using fewer disks. Accordingly, this reduction process does yield a recursive solution to the Towers of Hanoi problem.

Figure 6-13 contains a schematic diagram of the above recursive solution for

TOWER(4, A, B, C)

Observe that the recursive solution for $n = 4$ disks consists of the following 15 moves:

A → B	A → C	B → C	A → B	C → A	C → B	A → B	A → C
B → C	B → A	C → A	B → C	A → B	A → C	B → C	

In general, this recursive solution requires $f(n) = 2^n - 1$ moves for n disks.

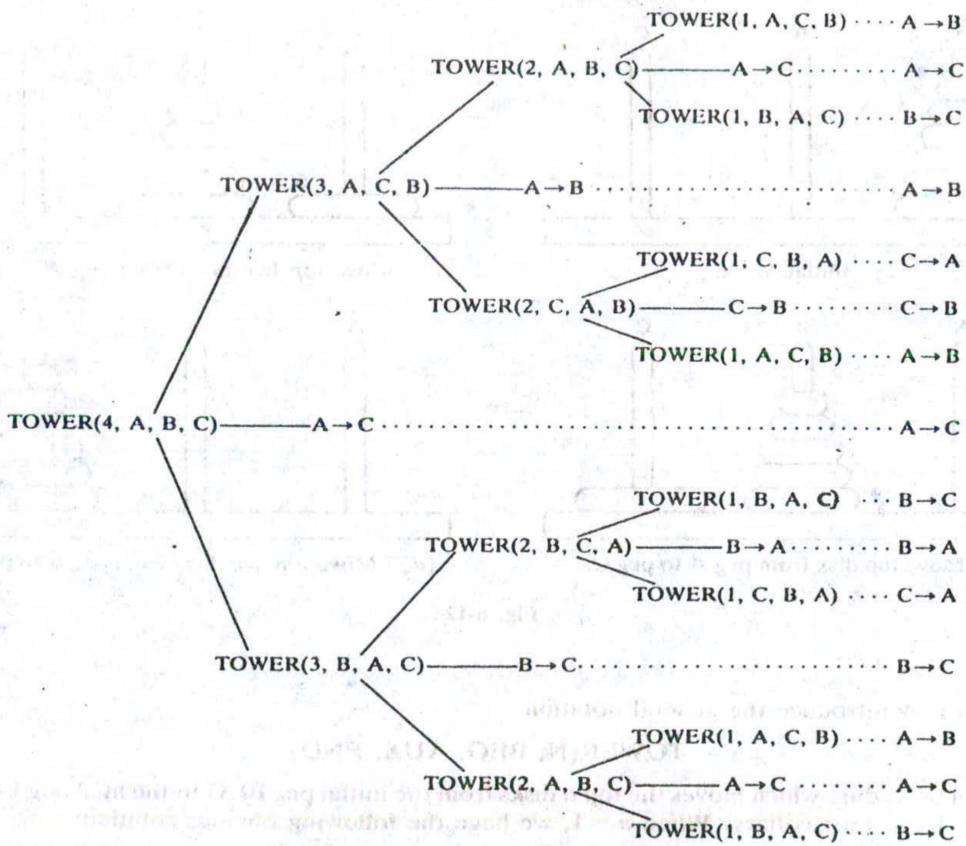


Fig. 6-13 Recursive solution to Towers of Hanoi problem for $n = 4$.

We summarize our investigation with the following formally written procedure.

Procedure 6.9: TOWER(N, BEG, AUX, END)
This procedure gives a recursive solution to the Towers of Hanoi problem for N disks.

1. If N = 1, then:
   (*a*) Write: BEG → END.
   (*b*) Return.
   [End of If structure.]
2. [Move N − 1 disks from peg BEG to peg AUX.]
   Call TOWER(N − 1, BEG, END, AUX).
3. Write: BEG → END.
4. [Move N − 1 disks from peg AUX to peg END.]
   Call TOWER(N − 1, AUX, BEG, END).
5. Return.

One can view this solution as a divide-and-conquer algorithm, since the solution for n disks is reduced to a solution for $n - 1$ disks and a solution for $n = 1$ disk.

6.8 IMPLEMENTATION OF RECURSIVE PROCEDURES BY STACKS

The preceding sections showed how recursion may be a useful tool in developing algorithms for specific problems. This section shows how stacks may be used to implement recursive procedures. It is instructive to first discuss subprograms in general.

Recall that a subprogram can contain both parameters and local variables. The parameters are the variables which receive values from objects in the calling program, called arguments, and which transmit values back to the calling program. Besides the parameters and local variables, the subprogram must also keep track of the return address in the calling program. This return address is essential, since control must be transferred back to its proper place in the calling program. At the time that the subprogram is finished executing and control is transferred back to the calling program, the values of the local variables and the return address are no longer needed.

Suppose our subprogram is a recursive program. Then each level of execution of the subprogram may contain different values for the parameters and local variables and for the return address. Furthermore, if the recursive program does call itself, then these current values must be saved, since they will be used again when the program is reactivated.

Suppose a programmer is using a high-level language which admits recursion, such as Pascal. Then the computer handles the bookkeeping that keeps track of all the values of the parameters, local variables and return addresses. On the other hand, if a programmer is using a high-level language which does not admit recursion, such as FORTRAN, then the programmer must set up the necessary bookkeeping by translating the recursive procedure into a nonrecursive one. This bookkeeping is discussed below.

Translation of a Recursive Procedure into a Nonrecursive Procedure

Suppose P is a recursive procedure. We assume that P is a subroutine subprogram rather than a function subprogram. (This is no loss in generality, since function subprograms can easily be written as subroutine subprograms.) We also assume that a recursive call to P comes only from the procedure P. (The treatment of indirect recursion lies beyond the scope of this text.)

The translation of the recursive procedure P into a nonrecursive procedure works as follows. First of all, one defines:

(1) A stack STPAR for each parameter PAR
(2) A stack STVAR for each local variable VAR
(3) A local variable ADD and a stack STADD to hold return addresses

Each time there is a recursive call to P, the current values of the parameters and local variables are pushed onto the corresponding stacks for future processing, and each time there is a recursive return to P, the values of parameters and local variables for the current execution of P are restored from the stacks. The handling of the return addresses is more complicated; it is done as follows.

Suppose the procedure P contains a recursive Call P in Step K. Then there are two return addresses associated with the execution of this Step K:

(1) There is the current return address of the procedure P, which will be used when the current level of execution of P is finished executing.
(2) There is the new return address K + 1, which is the address of the step following the Call P and which will be used to return to the current level of execution of procedure P.

Some texts push the first of these two addresses, the current return address, onto the return address stack STADD, whereas some texts push the second address, the new return address K + 1, onto STADD. We will choose the latter method, since the translation of P into a nonrecursive procedure will then be simpler. This also means, in particular, that an empty stack STADD will indicate a return to the main program that initially called the recursive procedure P. (The alternative translation which pushes the current return address onto the stack is discussed in Prob. 6.20.)

The algorithm which translates the recursive procedure P into a nonrecursive procedure follows. It consists of three parts: (1) preparation, (2) translating each recursive Call P in procedure P and (3) translating each Return in procedure P.

(1) Preparation.
   (*a*) Define a stack STPAR for each parameter PAR, a stack STVAR for each local variable VAR, and a local variable ADD and a stack STADD to hold return addresses.
   (*b*) Set TOP := NULL.

(2) Translation of "Step K. Call P."
   (*a*) Push the current values of the parameters and local variables onto the appropriate stacks, and push the new return address [Step] K + 1 onto STADD.
   (*b*) Reset the parameters using the new argument values.
   (*c*) Go to Step 1. [The beginning of the procedure P.]

(3) Translation of "Step J. Return."
   (*a*) If STADD is empty, then: Return. [Control is returned to the main program.]
   (*b*) Restore the top values of the stacks. That is, set the parameters and local variables equal to the top values on the stacks, and set ADD equal to the top value on the stack STADD.
   (*c*) Go to Step ADD.

Observe that the translation of "Step K. Call P" does depend on the value of K, but that the translation of "Step J. Return" does not depend on the value of J. Accordingly, one need translate only one Return statement, for example, by using

Step L. Return.

as above and then replace every other Return statement by

Go to Step L.

This will simplify the translation of the procedure.

Towers of Hanoi, Revisited

Consider again the Towers of Hanoi problem. Procedure 6.9 is a recursive solution to the problem for n disks. We translate the procedure into a nonrecursive solution. In order to keep the steps analogous, we label the beginning statement TOP := NULL as Step 0. Also, only the Return statement in Step 5 will be translated, as in (3) on the preceding page.

Procedure 6.10: TOWER(N, BEG, AUX, END)
This is a nonrecursive solution to the Towers of Hanoi problem for N disks which is obtained by translating the recursive solution. Stacks STN, STBEG, STAUX, STEND and STADD will correspond, respectively, to the variables N, BEG, AUX, END and ADD.

0. Set TOP := NULL.
1. If N = 1, then:
   (*a*) Write: BEG → END.
   (*b*) Go to Step 5.
   [End of If structure.]
2. [Translation of "Call TOWER(N − 1, BEG, END, AUX)."]
   (*a*) [Push current values and new return address onto stacks.]
      (i) Set TOP := TOP + 1.
      (ii) Set STN[TOP] := N, STBEG[TOP] := BEG,
         STAUX[TOP] := AUX, STEND[TOP] := END,
         STADD[TOP] := 3.
   (*b*) [Reset parameters.]
      Set N := N − 1, BEG := BEG, AUX := END, END := AUX.
   (*c*) Go to Step 1.
3. Write: BEG → END.
4. [Translation of "Call TOWER(N − 1, AUX, BEG, END)."]
   (*a*) [Push current values and new return address onto stacks.]
      (i) Set TOP := TOP + 1.
      (ii) Set STN[TOP] := N, STBEG[TOP] := BEG,
         STAUX[TOP] := AUX, STEND[TOP] := END,
         STADD[TOP] := 5.
   (*b*) [Reset parameters.]
      Set N := N − 1, BEG := AUX, AUX := BEG, END := END.
   (*c*) Go to Step 1.
5. [Translation of "Return."]
   (*a*) If TOP := NULL, then: Return.
   (*b*) [Restore top values on stacks.]
      (i) Set N := STN[TOP], BEG := STBEG[TOP],
         AUX := STAUX[TOP], STEND[TOP],
         ADD := STADD[TOP].
      (ii) Set TOP := TOP − 1.
   (*c*) Go to Step ADD.

Suppose that a main program does contain the following statement:

Call TOWER(3, A, B, C)

We simulate the execution of the solution of the problem in Procedure 6.10, emphasizing the different levels of execution of the procedure. Each level of execution will begin with an initialization step where the parameters are assigned the argument values from the initial calling statement or from the

	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)	(l)
STN:	3	3, 2	3	3, 2	3		3	3, 2	3	3, 2	3	
STBEG:	A	A, A	A	A, A	A		A	A, B	A	A, B	A	
STAUX:	B	B, C	B	B, C	B		B	B, A	B	B, A	B	
STEND:	C	C, B	C	C, B	C		C	C, C	C	C, C	C	
STADD:	3	3, 3	3	3, 5	3		5	5, 3	5	5, 5	5	

Fig. 6-14 Stacks for TOWER(3, A, B, C).

recursive call in Step 2 or Step 4. (Hence each new return address is either Step 3 or Step 5.) Figure 6-14 shows the different stages of the stacks.

(*a*) (Level 1) The initial Call TOWER(3, A, B, C) assigns the following values to the parameters:

$$N := 3, \quad BEG := A, \quad AUX := B, \quad END := C$$

Step 1. Since $N \neq 1$, control is transferred to Step 2.
Step 2. This is a recursive call. Hence the current values of the variables and the new return address (Step 3) are pushed onto the stacks as pictured in Fig. 6-14(*a*).

(*b*) (Level 2) The Step 2 recursive call [TOWER(N − 1, BEG, END, AUX)] assigns the following values to the parameters:

$$N := N - 1 = 2, \quad BEG := BEG = A, \quad AUX := END = C, \quad END := AUX = B$$

Step 1. Since $N \neq 1$, control is transferred to Step 2.
Step 2. This is a recursive call. Hence the current values of the variables and the new return address (Step 3) are pushed onto the stacks as pictured in Fig. 6-14(*b*).

(*c*) (Level 3) The Step 2 recursive call [TOWER(N − 1, BEG, END, AUX)] assigns the following values to the parameters:

$$N := N - 1 = 1, \quad BEG := BEG = A, \quad AUX := END = B, \quad END := AUX = C$$

Step 1. Now N = 1. The operation BEG → END implements the move

$$A \rightarrow C$$

Then control is transferred to Step 5. [For the Return.]
Step 5. The stacks are not empty, so the top values on the stacks are removed, leaving Fig. 6-14(*c*), and are assigned as follows:

$$N := 2, \quad BEG := A, \quad AUX := C, \quad END := B, \quad ADD := 3$$

Control is transferred to the preceding Level 2 at Step ADD.

(*d*) (Level 2) [Reactivated at Step ADD = 3.]
Step 3. The operation BEG → END implements the move

$$A \rightarrow B$$

Step 4. This is a recursive call. Hence the current values of the variables and the new return address (Step 5) are pushed onto the stacks as pictured in Fig. 6-14(*d*).

(*e*) (Level 3) The Step 4 recursive call [TOWER(N − 1, AUX, BEG, END)] assigns the following values to the parameters:

$$N := N - 1 = 1, \quad BEG := AUX = C, \quad AUX := BEG = A,$$
$$END := END = B$$

Step 1. Now N = 1. The operation BEG → END implements the move

$$C \rightarrow B$$

Then control is transferred to Step 5. [For the Return.]

Step 5. The stacks are not empty; hence the top values on the stacks are removed, leaving Fig. 6-14(*e*), and they are assigned as follows:

$$N := 2, \quad BEG := A, \quad AUX := C, \quad END := B, \quad ADD := 5$$

Control is transferred to the preceding Level 2 at Step ADD.

(*f*) (Level 2) [Reactivation at Step ADD = 5.]

Step 5. The stacks are not empty; hence the top values on the stacks are removed, leaving Fig. 6-14(*f*), and they are assigned as follows:

$$N := 3, \quad BEG := A, \quad AUX := B, \quad END := C, \quad ADD := 3$$

Control is transferred to the preceding Level 1 at Step ADD.

(*g*) (Level 1) [Reactivation at Step ADD = 3.]

Step 3. The operation BEG → END implements the move

$$A \rightarrow C$$

Step 4. This is a recursive call. Hence the current values of the variables and the new return address (Step 5) are pushed onto the stacks as pictured in Fig. 6-14(*g*).

(*h*) (Level 2) The Step 4 recursive call [TOWER(N − 1, AUX, BEG, END)] assigns the following values to the parameters:

$$N := N - 1 = 2, \quad BEG := AUX = B, \quad AUX := BEG = A, \quad END := END = C$$

Step 1. Since N ≠ 1, control is transferred to Step 2.

Step 2. This is a recursive call. Hence the current values of the variables and the new return address (Step 3) are pushed onto the stacks as pictured in Fig. 6-14(*h*).

(*i*) (Level 3) The Step 2 recursive call [TOWER(N − 1, BEG, END, AUX)] assigns the following values to the parameters:

$$N := N - 1 = 1, \quad BEG := BEG = B, \quad AUX := END = C, \quad END := AUX = A$$

Step 1. Now N = 1. The operation BEG → END implements the move

$$B \rightarrow A$$

Then control is transferred to Step 5. [For the Return.]

Step 5. The stacks are not empty; hence the top values on the stacks are removed, leaving Fig. 6-14(*i*), and they are assigned as follows:

$$N := 2, \quad BEG := B, \quad AUX := A, \quad END := C, \quad ADD := 3$$

Control is transferred to the preceding Level 2 at Step ADD.

(*j*) (Level 2) [Reactivation at Step ADD = 3.]

Step 3. The operation BEG → END implements the move

$$B \rightarrow C$$

Step 4. This is a recursive call. Hence the current values of the variables and the new return address (Step 5) are pushed onto the stacks as pictured in Fig. 6-14(*j*).

(*k*) (Level 3) The Step 4 recursive call [TOWER(N − 1, AUX, BEG, END)] assigns the following values to the parameters:

$$N := N - 1 = 1, \quad BEG := AUX = C, \quad AUX := BEG = B, \quad END := END = C$$

Step 1. Now N = 1. The operation BEG → END implements the move

$$A \rightarrow C$$

Then control is transferred to Step 5. [For the Return.]

Step 5. The stacks are not empty; hence the top values on the stacks are removed, leaving Fig. 6-14(*k*), and they are assigned as follows:

$$N := 2, \quad BEG := B, \quad AUX := A, \quad END := C, \quad ADD := 5$$

Control is transferred to the preceding Level 2 at Step ADD.

(*l*) (Level 2) [Reactivation at Step ADD = 5.]

Step 5. The stacks are not empty; hence the top values on the stacks are removed, leaving Fig. 6-14(*l*), and they are assigned as follows:

$$N := 3, \quad BEG := A, \quad AUX := B, \quad END := C, \quad ADD := 5$$

Control is transferred to the preceding Level 1 at Step ADD.

(*m*) (Level 1) [Reactivation at Step ADD = 5.]

Step 5. The stacks are now empty. Accordingly, control is transferred to the original main program containing the statement

Call TOWER(3, A, B, C)

Observe that the output consists of the following seven moves:

$$A \rightarrow C, \quad A \rightarrow B, \quad C \rightarrow B, \quad A \rightarrow C, \quad B \rightarrow A, \quad B \rightarrow C, \quad A \rightarrow C$$

This agrees with the solution in Fig. 6-11.

Summary

The Towers of Hanoi problem illustrates the power of recursion in the solution of various algorithmic problems. This section has shown how to implement recursion by means of stacks when using a programming language—notably FORTRAN or COBOL—which does not allow recursive programs. In fact, even when using a programming language—such as Pascal—which does support recursion, the programmer may want to use the nonrecursive solution, since it may be much less expensive than using the recursive solution.

6.9 QUEUES

A *queue* is a linear list of elements in which deletions can take place only at one end, called the *front*, and insertions can take place only at the other end, called the *rear*. The terms "front" and "rear" are used in describing a linear list only when it is implemented as a queue.

Queues are also called first-in first-out (FIFO) lists, since the first element in a queue will be the first element out of the queue. In other words, the order in which elements enter a queue is the order in which they leave. This contrasts with stacks, which are last-in first-out (LIFO) lists.

Queues abound in everyday life. The automobiles waiting to pass through an intersection form a queue, in which the first car in line is the first car through; the people waiting in line at a bank form a queue, where the first person in line is the first person to be waited on; and so on. An important example of a queue in computer science occurs in a timesharing system, in which programs with the same priority form a queue while waiting to be executed. (Another structure, called a priority queue, is discussed in Sec. 6.11.)

EXAMPLE 6.9

Figure 6-15(*a*) is a schematic diagram of a queue with 4 elements, where AAA is the front element and DDD is the rear element. Observe that the front and rear elements of the queue are also, respectively, the first and last elements of the list. Suppose an element is deleted from the queue. Then it must be AAA. This yields the queue in Fig. 6-15(*b*), where BBB is now the front element. Next, suppose EEE is added to the queue and then FFF is added to the queue. Then they must be added at the rear of the queue, as pictured in Fig. 6-15(*c*). Note that FFF is now the rear element. Now suppose another element is deleted from the queue; then it must be BBB, to yield the queue in Fig. 6-15(*d*). And so on. Observe that in such a data structure, EEE will be deleted before FFF because it has been placed in the queue before FFF. However, EEE will have to wait until CCC and DDD are deleted.

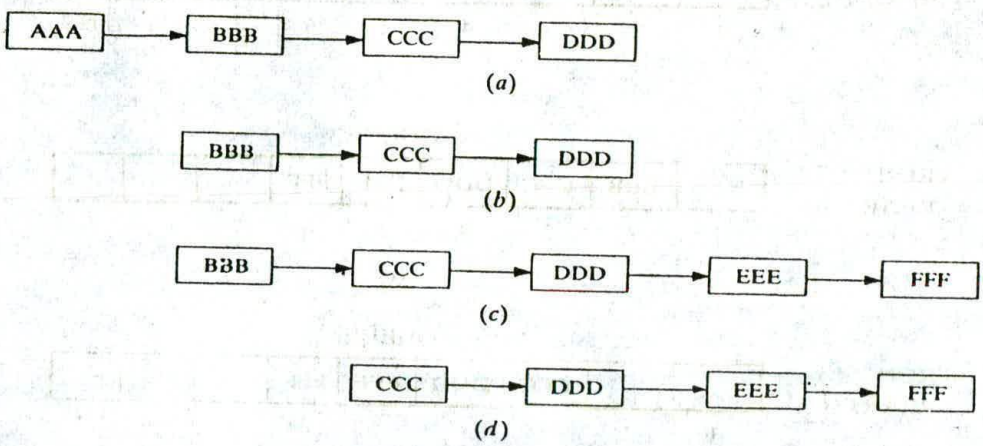


Fig. 6-15

Representation of Queues

Queues may be represented in the computer in various ways, usually by means of one-way lists or linear arrays. Unless otherwise stated or implied, each of our queues will be maintained by a linear array QUEUE and two pointer variables: FRONT, containing the location of the front element of the queue; and REAR, containing the location of the rear element of the queue. The condition FRONT = NULL will indicate that the queue is empty.

Figure 6-16 shows the way the array in Fig. 6-15 will be stored in memory using an array QUEUE with N elements. Figure 6-16 also indicates the way elements will be deleted from the queue and the way new elements will be added to the queue. Observe that whenever an element is deleted from the queue, the value of FRONT is increased by 1; this can be implemented by the assignment

$$\text{FRONT} := \text{FRONT} + 1$$

Similarly, whenever an element is added to the queue, the value of REAR is increased by 1; this can be implemented by the assignment

$$\text{REAR} := \text{REAR} + 1$$

This means that after N insertions, the rear element of the queue will occupy QUEUE[N] or, in other words, eventually the queue will occupy the last part of the array. This occurs even though the queue itself may not contain many elements.

Suppose we want to insert an element ITEM into a queue at the time the queue does occupy the last part of the array, i.e., when REAR = N. One way to do this is to simply move the entire queue to the beginning of the array, changing FRONT and REAR accordingly, and then inserting ITEM as above. This procedure may be very expensive. The procedure we adopt is to assume that the array

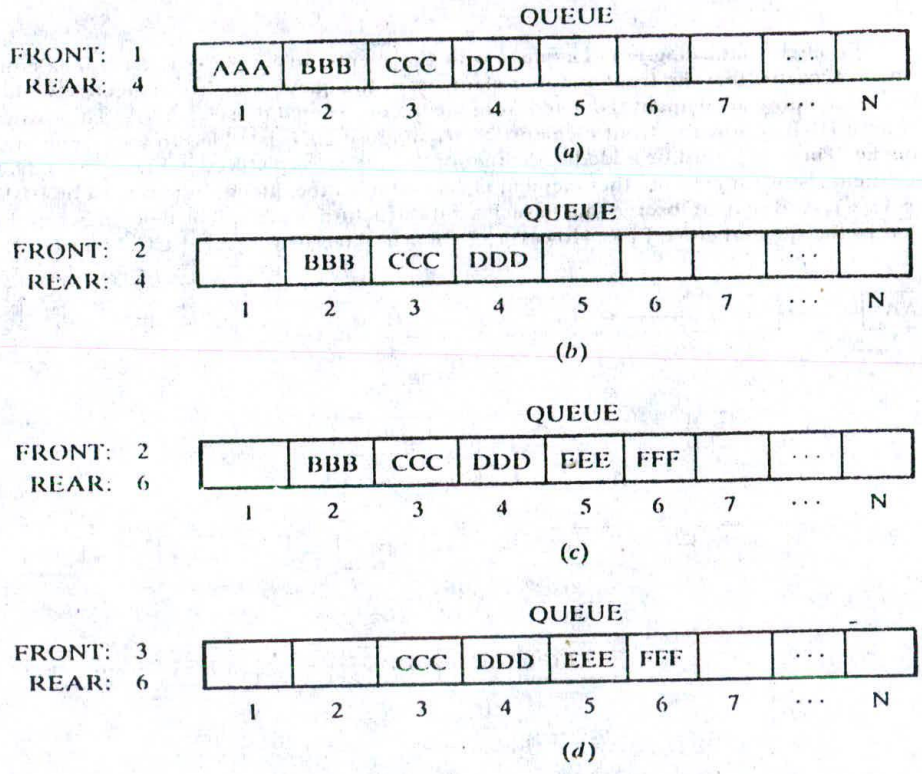


Fig. 6-16. Array representation of a queue.

QUEUE is circular, that is, that QUEUE[1] comes after QUEUE[N] in the array. With this assumption, we insert ITEM into the queue by assigning ITEM to QUEUE[1]. Specifically, instead of increasing REAR to N + 1, we reset REAR = 1 and then assign

$$\text{QUEUE[REAR]} := \text{ITEM}$$

Similarly, if FRONT = N and an element of QUEUE is deleted, we reset FRONT = 1 instead of increasing FRONT to N + 1. (Some readers may recognize this as modular arithmetic, discussed in Sec. 2.2.)

Suppose that our queue contains only one element, i.e., suppose that

$$\text{FRONT} = \text{REAR} \neq \text{NULL}$$

and suppose that the element is deleted. Then we assign

$$\text{FRONT} := \text{NULL} \quad \text{and} \quad \text{REAR} := \text{NULL}$$

to indicate that the queue is empty.

EXAMPLE 6.10

Figure 6-17 shows how a queue may be maintained by a circular array QUEUE with N = 5 memory locations. Observe that the queue always occupies consecutive locations except when it occupies locations at the beginning and at the end of the array. If the queue is viewed as a circular array, this means that it still occupies consecutive locations. Also, as indicated by Fig. 6-17(*m*), the queue will be empty only when FRONT = REAR and an element is deleted. For this reason, NULL is assigned to FRONT and REAR in Fig. 6-17(*m*).

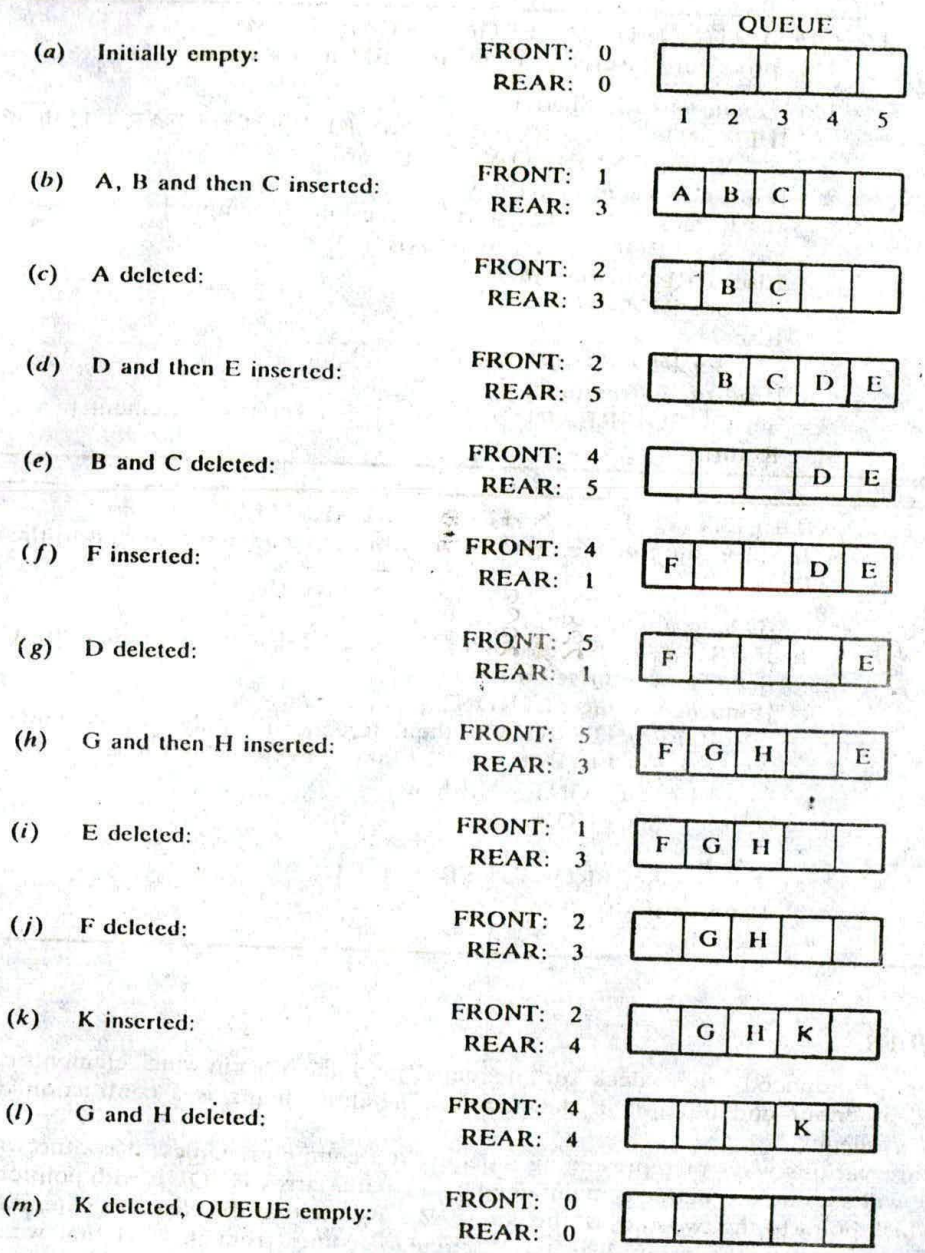


Fig. 6-17

We are now prepared to formally state our procedure QINSERT (Procedure 6.11), which inserts a data ITEM into a queue. The first thing we do in the procedure is to test for overflow, that is, to test whether or not the queue is filled.

Next we give a procedure QDELETE (Procedure 6.12), which deletes the first element from a queue, assigning it to the variable ITEM. The first thing we do is to test for underflow, i.e., to test whether or not the queue is empty.

Procedure 6.11: QINSERT(QUEUE, N, FRONT, REAR, ITEM)
This procedure inserts an element ITEM into a queue.

```
1. [Queue already filled?]
   If FRONT = 1 and REAR = N, or if FRONT = REAR + 1, then:
      Write: OVERFLOW, and Return.
2. [Find new value of REAR.]
   If FRONT := NULL, then: [Queue initially empty.]
      Set FRONT := 1 and REAR := 1.
   Else if REAR = N, then:
      Set REAR := 1.
   Else:
      Set REAR := REAR + 1.
   [End of If structure.]
3. Set QUEUE[REAR] := ITEM. [This inserts new element.]
4. Return.
```

Procedure 6.12: QDELETE(QUEUE, N, FRONT, REAR, ITEM)
This procedure deletes an element from a queue and assigns it to the variable ITEM.

```
1. [Queue already empty?]
   If FRONT := NULL, then: Write: UNDERFLOW, and Return.
2. Set ITEM := QUEUE[FRONT].
3. [Find new value of FRONT.]
      If FRONT = REAR, then: [Queue has only one element to start.]
         Set FRONT := NULL and REAR := NULL.
      Else if FRONT = N, then:
         Set FRONT := 1.
      Else:
         Set FRONT := FRONT + 1.
   [End of If structure.]
4. Return.
```

6.10 DEQUES

A *deque* (pronounced either "deck" or "dequeue") is a linear list in which elements can be added or removed at either end but not in the middle. The term deque is a contraction of the name *double-ended queue*.

There are various ways of representing a deque in a computer. Unless it is otherwise stated or implied, we will assume our deque is maintained by a circular array DEQUE with pointers LEFT and RIGHT, which point to the two ends of the deque. We assume that the elements extend from the left end to the right end in the array. The term "circular" comes from the fact that we assume that DEQUE[1] comes after DEQUE[N] in the array. Figure 6-18 pictures two deques, each with 4 elements maintained in an array with N = 8 memory locations. The condition LEFT = NULL will be used to indicate that a deque is empty.

There are two variations of a deque—namely, an input-restricted deque and an output-restricted deque—which are intermediate between a deque and a queue. Specifically, an *input-restricted deque* is a deque which allows insertions at only one end of the list but allows deletions at both ends of the list; and an *output-restricted deque* is a deque which allows deletions at only one end of the list but allows insertions at both ends of the list.

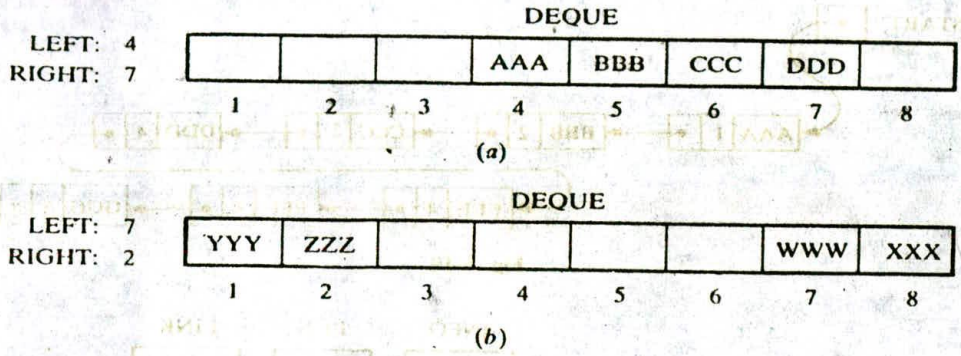


Fig. 6-18

The procedures which insert and delete elements in deques and the variations on those procedures are given as supplementary problems. As with queues, a complication may arise (*a*) when there is overflow, that is, when an element is to be inserted into a deque which is already full, or (*b*) when there is underflow, that is, when an element is to be deleted from a deque which is empty. The procedures must consider these possibilities.

6.11 PRIORITY QUEUES

A *priority queue* is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

(1) An element of higher priority is processed before any element of lower priority.

(2) Two elements with the same priority are processed according to the order in which they were added to the queue.

A prototype of a priority queue is a timesharing system: programs of high priority are processed first, and programs with the same priority form a standard queue.

There are various ways of maintaining a priority queue in memory. We discuss two of them here: one uses a one-way list, and the other uses multiple queues. The ease or difficulty in adding elements to or deleting them from a priority queue clearly depends on the representation that one chooses.

One-Way List Representation of a Priority Queue

One way to maintain a priority queue in memory is by means of a one-way list, as follows:

(*a*) Each node in the list will contain three items of information: an information field INFO, a priority number PRN and a link number LINK.

(*b*) A node X precedes a node Y in the list (1) when X has higher priority than Y or (2) when both have the same priority but X was added to the list before Y. This means that the order in the one-way list corresponds to the order of the priority queue.

Priority numbers will operate in the usual way: the lower the priority number, the higher the priority.

EXAMPLE 6.11

Figure 6-19 shows a schematic diagram of a priority queue with 7 elements. The diagram does not tell us whether BBB was added to the list before or after DDD. On the other hand, the diagram does tell us that BBB was inserted before CCC, because BBB and CCC have the same priority number and BBB appears before CCC in the list. Figure 6-20 shows the way the priority queue may appear in memory using linear arrays INFO, PRN and LINK. (See Sec. 5.2.)

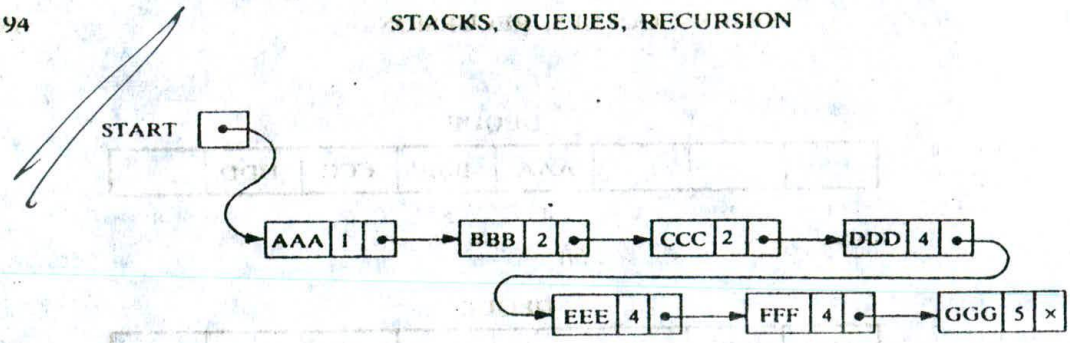


Fig. 6-19

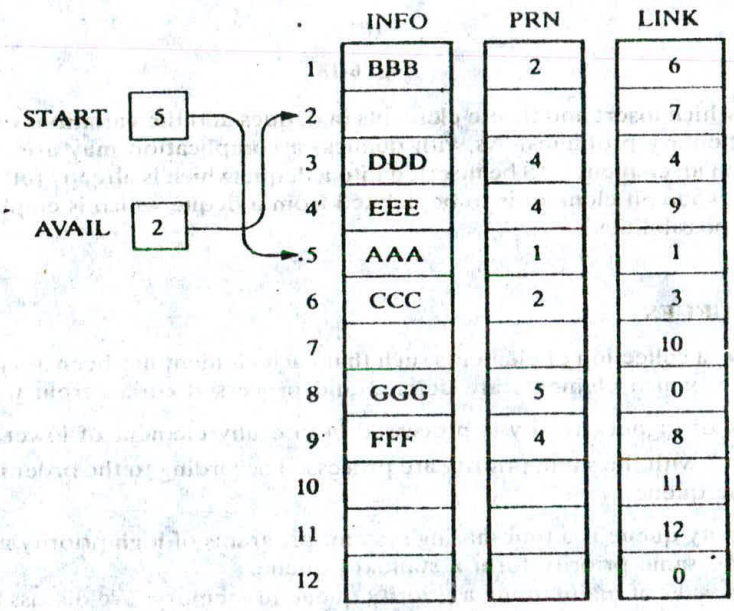


Fig. 6-20

The main property of the one-way list representation of a priority queue is that the element in the queue that should be processed first always appears at the beginning of the one-way list. Accordingly, it is a very simple matter to delete and process an element from our priority queue. The outline of the algorithm follows.

Algorithm 6.13: This algorithm deletes and processes the first element in a priority queue which appears in memory as a one-way list.

1. Set ITEM := INFO[START]. [This saves the data in the first node.]
2. Delete first node from the list.
3. Process ITEM.
4. Exit.

The details of the algorithm, including the possibility of underflow, are left as an exercise.

Adding an element to our priority queue is much more complicated than deleting an element from the queue, because we need to find the correct place to insert the element. An outline of the algorithm follows.

Algorithm 6.14: This algorithm adds an ITEM with priority number N to a priority queue which is maintained in memory as a one-way list.

(*a*) Traverse the one-way list until finding a node X whose priority number exceeds N. Insert ITEM in front of node X.

(*b*) If no such node is found, insert ITEM as the last element of the list.

The above insertion algorithm may be pictured as a weighted object "sinking" through layers of elements until it meets an element with a heavier weight.

The details of the above algorithm are left as an exercise. The main difficulty in the algorithm comes from the fact that ITEM is inserted before node X. This means that, while traversing the list, one must also keep track of the address of the node preceding the node being accessed.

EXAMPLE 6.12

Consider the priority queue in Fig. 6-19. Suppose an item XXX with priority number 2 is to be inserted into the queue. We traverse the list, comparing priority numbers. Observe that DDD is the first element in the list whose priority number exceeds that of XXX. Hence XXX is inserted in the list in front of DDD, as pictured in Fig. 6-21. Observe that XXX comes after BBB and CCC, which have the same priority as XXX. Suppose now that an element is to be deleted from the queue. It will be AAA, the first element in the list. Assuming no other insertions, the next element to be deleted will be BBB, then CCC, then XXX, and so on.

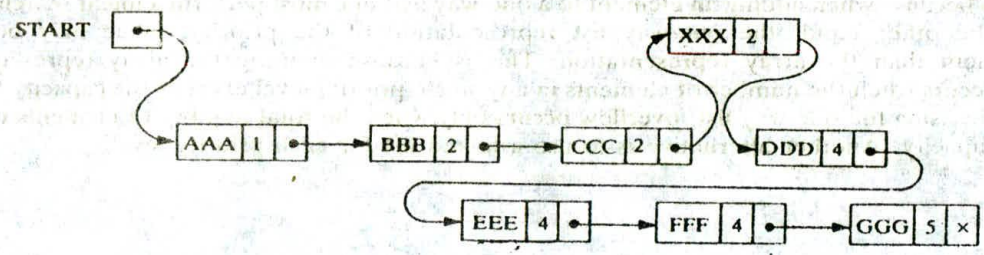


Fig. 6-21

Array Representation of a Priority Queue

Another way to maintain a priority queue in memory is to use a separate queue for each level of priority (or for each priority number). Each such queue will appear in its own circular array and must have its own pair of pointers, FRONT and REAR. In fact, if each queue is allocated the same amount of space, a two-dimensional array QUEUE can be used instead of the linear arrays. Figure 6-22 indicates this representation for the priority queue in Fig. 6-21. Observe that FRONT[K] and REAR[K] contain, respectively, the front and rear elements of row K of QUEUE, the row that maintains the queue of elements with priority number K.

	FRONT	REAR
1	2	2
2	1	3
3	0	0
4	5	1
5	4	4

	1	2	3	4	5	6
1		AAA				
2	BBB	CCC	XXX			
3						
4	FFF				DDD	EEE
5				GGG		

Fig. 6-22

The following are outlines of algorithms for deleting and inserting elements in a priority queue that is maintained in memory by a two-dimensional array QUEUE, as above. The details of the algorithms are left as exercises.

Algorithm 6.15: This algorithm deletes and processes the first element in a priority queue maintained by a two-dimensional array QUEUE.

1. [Find the first nonempty queue.]
   Find the smallest K such that FRONT[K] ≠ NULL.
2. Delete and process the front element in row K of QUEUE.
3. Exit.

Algorithm 6.16: This algorithm adds an ITEM with priority number M to a priority queue maintained by a two-dimensional array QUEUE.

1. Insert ITEM as the rear element in row M of QUEUE.
2. Exit.

Summary

Once again we see the time-space tradeoff when choosing between different data structures for a given problem. The array representation of a priority queue is more time-efficient than the one-way list. This is because when adding an element to a one-way list, one must perform a linear search on the list. On the other hand, the one-way list representation of the priority queue may be more space-efficient than the array representation. This is because in using the array representation, overflow occurs when the number of elements in any single priority level exceeds the capacity for that level, but in using the one-way list, overflow occurs only when the total number of elements exceeds the *total* capacity. Another alternative is to use a linked list for each priority level.

Solved Problems

STACKS

6.1 Consider the following stack of characters, where STACK is allocated N = 8 memory cells:

STACK: A, C, D, F, K, __, __, __

(For notational convenience, we use "__" to denote an empty memory cell.) Describe the stack as the following operations take place:

(*a*) POP(STACK, ITEM)
(*b*) POP(STACK, ITEM)
(*c*) PUSH(STACK, L)
(*d*) PUSH(STACK, P)
(*e*) POP(STACK, ITEM)
(*f*) PUSH(STACK, R)
(*g*) PUSH(STACK, S)
(*h*) POP(STACK, ITEM)

The POP procedure always deletes the top element from the stack, and the PUSH procedure always adds the new element to the top of the stack. Accordingly:

(*a*) STACK: A, C, D, F, __, __, __, __
(*b*) STACK: A, C, D, __, __, __, __, __
(*c*) STACK: A, C, D, L, __, __, __, __
(*d*) STACK: A, C, D, L, P, __, __, __
(*e*) STACK: A, C, D, L, __, __, __, __
(*f*) STACK: A, C, D, L, R, __, __, __
(*g*) STACK: A, C, D, L, R, S, __, __
(*h*) STACK: A, C, D, L, R, __, __, __

6.2 Consider the data in Prob. 6.1. (*a*) When will overflow occur? (*b*) When will C be deleted before D?

(*a*) Since STACK has been allocated N = 8 memory cells, overflow will occur when STACK contains 8 elements and there is a PUSH operation to add another element to STACK.

(*b*) Since STACK is implemented as a stack, C will never be deleted before D.

6.3 Consider the following stack, where STACK is allocated N = 6 memory cells:

STACK: AAA, DDD, EEE, FFF, GGG, ___ __

Describe the stack as the following operations take place: (*a*) PUSH(STACK, KKK), (*b*) POP(STACK, ITEM), (*c*) PUSH(STACK, LLL), (*d*) PUSH(STACK, SSS), (*e*) POP(STACK, ITEM) and (*f*) PUSH(STACK, TTT).

(*a*) KKK is added to the top of STACK, yielding

STACK: AAA, DDD, EEE, FFF, GGG, KKK

(*b*) The top element is removed from STACK, yielding

STACK: AAA, DDD, EEE, FFF, GGG, _____

(*c*) LLL is added to the top of STACK, yielding

STACK: AAA, DDD, EEE, FFF, GGG, LLL

(*d*) Overflow occurs, since STACK is full and another element SSS is to be added to STACK.

No further operations can take place until the overflow is resolved—by adding additional space for STACK, for example.

6.4 Suppose STACK is allocated N = 6 memory cells and initially STACK is empty, or, in other words, TOP = 0. Find the output of the following module:

```
1. Set AAA := 2 and BBB := 5.
2. Call PUSH(STACK, AAA).
   Call PUSH(STACK, 4).
   Call PUSH(STACK, BBB + 2).
   Call PUSH(STACK, 9).
   Call PUSH(STACK, AAA + BBB).
3. Repeat while TOP ≠ 0:
       Call POP(STACK, ITEM).
       Write: ITEM.
   [End of loop.]
4. Return.
```

Step 1. Sets AAA = 2 and BBB = 5.

Step 2. Pushes AAA = 2, 4, BBB + 2 = 7, 9 and AAA + BBB = 7 onto STACK, yielding

STACK: 2, 4, 7, 9, 7, __

Step 3. Pops and prints the elements of STACK until STACK is empty. Since the top element is always popped, the output consists of the following sequence:

7, 9, 7, 4, 2

Observe that this is the reverse of the order in which the elements were added to STACK.

6.5 Suppose a given space S of N contiguous memory cells is allocated to K = 6 stacks. Describe ways that the stacks may be maintained in S.

Suppose no prior data indicate that any one stack will grow more rapidly than any of the other stacks. Then one may reserve N/K cells for each stack, as in Fig. 6-23(*a*), where $B_1, B_2, \ldots, B_6$ denote, respectively, the bottoms of the stacks. Alternatively, one can partition the stack into pairs and reserve 2N/K cells for each pair of stacks, as in Fig. 6-23(*b*). The second method may decrease the number of times overflow will occur.

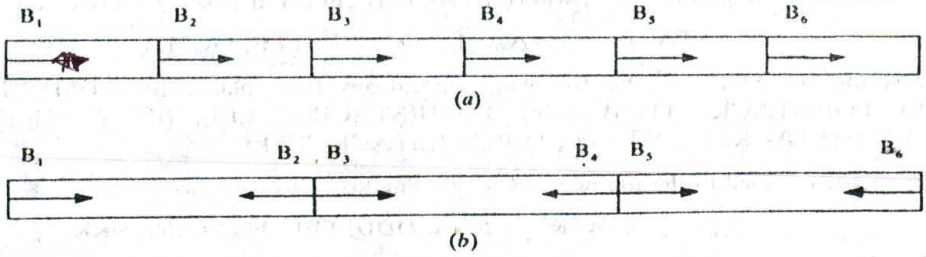


Fig. 6-23

POLISH NOTATION

6.6 Translate, by inspection and hand, each infix expression into its equivalent postfix expression:

(*a*) $(A - B) * (D/E)$ (*b*) $(A + B \uparrow D)/(E - F) + G$

(*c*) $A * (B + D)/E - F * (G + H/K)$

Using the order in which the operators are executed, translate each operator from infix to postfix notation. (We use brackets [ ] to denote a partial translation.)

(*a*) $(A - B) * (D/E) = [AB-] * [DE/] = AB-DE/*$

(*b*) $(A + B \uparrow D)/(E - F) + G = (A + [BD\uparrow])/[EF-] + G = [ABD\uparrow +]/[EF-] + G$

$= [ABD\uparrow + EF - /] + G = ABD\uparrow + EF - /G +$

(*c*) $A * (B + D)/E - F * (G + H/K) = A * [BD+]/E - F * (G + [HK/])$

$= [ABD+*]/E - F * [GHK/+]$

$= [ABD+*E/] - [FGHK/+*]$

$= ABD+*E/FGHK/+*-$

Observe that we did translate more than one operator in a single step when the operands did not overlap.

6.7 Consider the following arithmetic expression P, written in postfix notation:

P: 12, 7, 3, −, /, 2, 1, 5, +, *, +

(*a*) Translate P, by inspection and hand, into its equivalent infix expression.

(*b*) Evaluate the infix expression.

(*a*) Scanning from left to right, translate each operator from postfix to infix notation. (We use brackets [ ] to denote a partial translation.)

$P = 12, [7 - 3], /, 2, 1, 5, +, *, +$

$= [12/(7 - 3)], 2, 1, 5, +, *, +$

$= [12/(7 - 3)], 2, [1 + 5], *, +$

$= [12/(7 - 3)], [2 * (1 + 5)], +$

$= 12/(7 - 3) + 2 * (1 + 5)$

(*b*) Using the infix expression, we obtain:

$$P = 12/(7 - 3) + 2*(1 + 5) = 12/4 + 2*6 = 3 + 12 = 15$$

6.8 Consider the postfix expression P in Prob. 6.7. Evaluate P using Algorithm 6.3.

First add a sentinel right parenthesis at the end of P to obtain:

P: 12, 7, 3, −, /, 2, 1, 5, +, *, +,)

Scan P from left to right. If a constant is encountered, put it on a stack, but if an operator is encountered, evaluate the two top constants on the stack. Figure 6-24 shows the contents of STACK as each element of P is scanned. The final number, 15, in STACK, when the sentinel right parenthesis is scanned, is the value of P. This agrees with the result in Prob. 6.7(*b*).

Symbol	STACK
12	12
7	12, 7
3	12, 7, 3
−	12, 4
/	3
2	3, 2
1	3, 2, 1
5	3, 2, 1, 5
+	3, 2, 6
*	3, 12
+	15
)	15

Fig. 6-24

6.9 Consider the following infix expression Q:

$$Q: \quad ((A + B)*D) \uparrow (E - F)$$

Use Algorithm 6.4 to translate Q into its equivalent postfix expression P.

First push a left parenthesis onto STACK, and then add a right parenthesis to the end of Q to obtain

Q: ((A + B) * D) ↑ (E − F))

(Note that Q now contains 16 elements.) Scan Q from left to right. Recall that (1) if a constant is encountered, it is added to P; (2) if a left parenthesis is encountered, it is put on the stack; (3) if an operator is encountered, it "sinks" to its own level; and (4) if a right parenthesis is encountered, it "sinks" to the first left parenthesis. Figure 6-25 shows pictures of STACK and the string P as each element of Q is scanned. When STACK is empty, the final right parenthesis has been scanned and the result is

P: A B + D * E F − ↑

which is the required postfix equivalent of Q.

6.10 Translate, by inspection and hand, each infix expression into its equivalent prefix expression:

(*a*) $(A - B)*(D / E)$

(*b*) $(A + B \uparrow D)/(E - F) + G$

Symbol	STACK	Expression P
(	((	
(	(((	
A	(((	A
+	(((+	A
B	(((+	A B
)	((	A B +
*	((*	A B +
D	((*	A B + D
)	(	A B + D *
↑	(↑	A B + D *
(	(↑ (	A B + D *
E	(↑ (	A B + D * E
−	(↑ (−	A B + D * E
F	(↑ (−	A B + D * E F
)	(↑	A B + D * E F −
)		A B + D * E F −

Fig. 6-25

Is there any relationship between the prefix expressions and the equivalent postfix expressions obtained in Prob. 6.6.

Using the order in which the operators are executed, translate each operator from infix to prefix notation.

(*a*) $(A - B) * (D/E) = [-AB] * [/DE] = * - A B / D E$

(*b*)
$$(A + B \uparrow D)/(E - F) + G = (A + [\uparrow BD])/[-EF] + G$$
$$= [+A\uparrow BD]/[-EF] + G$$
$$= [/+A\uparrow BD - EF] + G$$
$$= + / + A \uparrow B D - E F G$$

The prefix expression is not the reverse of the postfix expression. However, the order of the operands—A, B, D and E in part (*a*) and A, B, D, E, F and G in part (*b*)—is the same for all three expressions, infix, postfix and prefix.

QUICKSORT

6.11 Suppose S is the following list of 14 alphabetic characters:

(D) A T A S T R U C T U R E (S)

Suppose the characters in S are to be sorted alphabetically. Use the quicksort algorithm to find the final position of the first character D.

Beginning with the last character S, scan the list from right to left until finding a character which precedes D alphabetically. It is C. Interchange D and C to obtain the list:

(C) A T A S T R U (D) T U R E S

Beginning with this C, scan the list toward D, i.e., from left to right, until finding a character which succeeds D alphabetically. It is T. Interchange D and T to obtain the list:

C A (D) A S (T) R U T T U R E S

Beginning with this T, scan the list toward D until finding a character which precedes D. It is A. Interchange D and A to obtain the list:

C A (A) (D) S T R U T T U R E S

Beginning with this A, scan the list toward D until finding a character which succeeds D. There is no such letter. This means D is in its final position. Furthermore, the letters before D form a sublist consisting of all letters preceding D alphabetically, and the letters after D form a sublist consisting of all the letters succeeding D alphabetically, as follows:

C A A (D) S T R U T T U R E S

Sublist Sublist

Sorting S is now reduced to sorting each sublist.

6.12 Suppose S consists of the following $n = 5$ letters:

(A) B C D (E)

Find the number C of comparisons to sort S using quicksort. What general conclusion can one make, if any?

Beginning with E, it takes $n - 1 = 4$ comparisons to recognize that the first letter A is already in its correct position. Sorting S is now reduced to sorting the following sublist with $n - 1 = 4$ letters:

A (B) C D (E)

Beginning with E, it takes $n - 2 = 3$ comparisons to recognize that the first letter B in the sublist is already in its correct position. Sorting S is now reduced to sorting the following sublist with $n - 2 = 3$ letters:

A B (C) D (E)

Similarly, it takes $n - 3 = 2$ comparisons to recognize that the letter C is in its correct position, and it takes $n - 4 = 1$ comparison to recognize that the letter D is in its correct position. Since only one letter is left, the list is now known to be sorted. Altogether we have:

$$C = 4 + 3 + 2 + 1 = 10 \text{ comparisons}$$

Similarly, using quicksort, it takes

$$C = (n - 1) + (n - 2) + \cdots + 2 + 1 = \frac{n(n-1)}{2} = \frac{n^2}{2} + 0(n) = O(n^2)$$

comparisons to sort a list with n elements when the list is already sorted. (This can be shown to be the worst case for quicksort.)

6.13 Consider the quicksort algorithm. (*a*) Can the arrays LOWER and UPPER be implemented as queues rather than as stacks? Why? (*b*) How much extra space is needed for the quicksort algorithm, or, in other words, what is the space complexity of the algorithm?

(*a*) Since the order in which the subsets are sorted does not matter, LOWER and UPPER can be implemented as queues, or even deques, rather than as stacks.

(*b*) Quicksort algorithm is an "in-place" algorithm; that is, the elements remain in their places except for interchanges. The extra space is required mainly for the stacks LOWER and UPPER. On the average, the extra space required for the algorithm is proportional to $\log n$, where n is the number of elements to be sorted.

RECURSION

6.14 Let a and b denote positive integers. Suppose a function Q is defined recursively as follows:

$$Q(a, b) = \begin{cases} 0 & \text{if } a < b \\ Q(a - b, b) + 1 & \text{if } b \le a \end{cases}$$

(*a*) Find the value of $Q(2, 3)$ and $Q(14, 3)$.

(*b*) What does this function do? Find $Q(5861, 7)$.

(*a*)
$$\begin{aligned} Q(2, 3) &= 0 \quad \text{since} \quad 2 < 3 \\ Q(14, 3) &= Q(11, 3) + 1 \\ &= [Q(8, 3) + 1] + 1 = Q(8, 3) + 2 \\ &= [Q(5, 3) + 1] + 2 = Q(5, 3) + 3 \\ &= [Q(2, 3) + 1] + 3 = Q(2, 3) + 4 \\ &= 0 + 4 = 4 \end{aligned}$$

(*b*) Each time b is subtracted from a, the values of Q is increased by 1. Hence $Q(a, b)$ finds the quotient when a is divided by b. Thus,

$$Q(5861, 7) = 837$$

6.15 Let n denote a positive integer. Suppose a function L is defined recursively as follows:

$$L(n) = \begin{cases} 0 & \text{if } n = 1 \\ L(\lfloor n/2 \rfloor) + 1 & \text{if } n > 1 \end{cases}$$

(Here $\lfloor k \rfloor$ denotes the "floor" of k, that is, the greatest integer which does not exceed k. See Sec. 2.2.)

(*a*) Find $L(25)$.

(*b*) What does this function do?

(*a*)
$$\begin{aligned} L(25) &= L(12) + 1 \\ &= [L(6) + 1] + 1 = L(6) + 2 \\ &= [L(3) + 1] + 2 = L(3) + 3 \\ &= [L(1) + 1] + 3 = L(1) + 4 \\ &= 0 + 4 = 4 \end{aligned}$$

(*b*) Each time n is divided by 2, the value of L is increased by 1. Hence L is the greatest integer such that

$$2^L \le n$$

Accordingly, this function finds

$$L = \lfloor \log_2 n \rfloor$$

6.16 Suppose the Fibonacci numbers $F_{11} = 89$ and $F_{12} = 144$ are given.

(*a*) Should one use recursion or iteration to obtain F_{16}? Find F_{16}.

(*b*) Write an iterative procedure to obtain the first N Fibonacci numbers F[1], F[2], . . . , F[N], where $N > 2$. (Compare this with the recursive Procedure 6.8.)

(*a*) The Fibonacci numbers should be evaluated by using iteration (that is, by evaluating from the bottom up), rather than by using recursion (that is, evaluating from the top down).

Recall that each Fibonacci number is the sum of the two preceding Fibonacci numbers. Beginning with F_{11} and F_{12} we have

$$F_{13} = 89 + 144 = 233, \qquad F_{14} = 144 + 233 = 377, \qquad F_{15} = 233 + 377 = 610$$

and hence

$$F_{16} = 377 + 610 = 987$$

(*b*) **Procedure P6.16:** FIBONACCI(F, N)
This procedure finds the first N Fibonacci numbers and assigns them to an array F.

1. Set F[1] := 1 and F[2] := 1.
2. Repeat for L = 3 to N:
   Set F[L] := F[L − 1] + F[L − 2].
   [End of loop.]
3. Return.

(We emphasize that this iterative procedure is much more efficient than the recursive Procedure 6.8.)

6.17 Use the definition of the Ackermann function (Definition 6.3) to find $A(1, 3)$.

We have the following 15 steps:

(1) $A(1, 3) = A(0, A(1, 2))$
(2) $A(1, 2) = A(0, A(1, 1))$
(3) $A(1, 1) = A(0, A(1, 0))$
(4) $A(1, 0) = A(0, 1)$
(5) $A(0, 1) = 1 + 1 = 2$
(6) $A(1, 0) = 2$
(7) $A(1, 1) = A(0, 2)$
(8) $A(0, 2) = 2 + 1 = 3$
(9) $A(1, 1) = 3$
(10) $A(1, 2) = A(0, 3)$
(11) $A(0, 3) = 3 + 1 = 4$
(12) $A(1, 2) = 4$
(13) $A(1, 3) = A(0, 4)$
(14) $A(0, 4) = 4 + 1 = 5$
(15) $A(1, 3) = 5$

The forward indention indicates that we are postponing an evaluation and are recalling the definition, and the backward indention indicates that we are backtracking.

Observe that the first formula in Definition 6.3 is used in Steps 5, 8, 11 and 14, the second formula in Step 4 and the third formula in Steps 1, 2 and 3. In the other Steps we are backtracking with substitutions.

6.18 Suppose a recursive procedure P contains only one recursive call:

Step K. Call P.

Indicate the reason that the stack STADD (for the return addresses) is not necessary.

Since there is only one recursive call, control will always be transferred to Step K + 1 on a Return, except for the final Return to the main program. Accordingly, instead of maintaining the stack STADD

(and the local variable ADD), we simply write

(*c*) Go to Step K + 1

instead of

(*c*) Go to Step ADD

in the translation of "Step J. Return." (See Sec. 6.8.)

6.19 Rewrite the solution to the Towers of Hanoi problem so it uses only one recursive call instead of two.

One may view the pegs A and B symmetrically. That is, we apply the steps

Move N − 1 disks from A to B, and then apply A → C
Move N − 2 disks from B to A, and then apply B → C
Move N − 3 disks from A to B, and then apply A → C
Move N − 4 disks from B to A, and then apply B → C

and so on. Accordingly, we can iterate a single recursive call, interchanging BEG and AUX after each iteration, as follows:

Procedure P6.19: TOWER(N, BEG, AUX, END)

1. If N = 0, then: Return.
2. Repeat Steps 3 to 5 for K = N, N − 1, N − 2, . . . , 1.
3. Call TOWER(K − 1, BEG, END, AUX).
4. Write: BEG → END.
5. [Interchange BEG and AUX.]
   Set TEMP := BEG, BEG := AUX, AUX := TEMP.
   [End of Step 2 loop.]
6. Return.

Observe that we use N = 0 as a base value for the recursion instead of N = 1. Either one may be used to yield a solution.

6.20 Consider the stack implementation algorithm in Sec. 6.8 for translating a recursive procedure into a nonrecursive procedure. Recall that, at the time of a recursive call, we pushed the new return address rather than the current return address onto the stack STADD. Suppose we decide to push the current return address onto the stack STADD. (Many texts do this.) What changes must then take place in the translation algorithm?

The main change is that, at the time of a Return to the preceding execution level, the current value of ADD determines the location of the Return, not the value of ADD after the stack values have been popped. Accordingly, the value of ADD must be saved, by setting SAVE := ADD, then the stack values are popped, and then control is transferred to Step SAVE. Another change is that one must initially assign ADD := Main and then Return to the main calling program when ADD = Main, not when the stacks are empty. The formal algorithm follows.

(1) Preparation.
   (*a*) Define a stack STPAR for each parameter PAR, a stack STVAR for each local variable VAR and a local variable ADD and a stack STADD to hold return addresses.
   (*b*) Set TOP := NULL and ADD := Main.

(2) Translation of "Step K. Call P."
   (*a*) Push the current values of the parameters and local variables and the current return address ADD onto the appropriate stacks.
   (*b*) Reset the parameters using the new argument values, and set ADD := [Step] K + 1.
   (*c*) Go to Step 1. [The beginning of the procedure P.]

(3) Translation of "Step J. Return."
   (*a*) If ADD = Main, then: Return. [Control is transferred to the main program.]
   (*b*) Set SAVE := ADD.
   (*c*) Restore the top values of the stacks. That is, set the parameters and local variables equal to the top values on the stacks, and set ADD equal to the top value on the stack STADD.
   (*d*) Go to Step SAVE.

(Compare this translation algorithm with the algorithm in Sec. 6.8.)

QUEUES, DEQUES

6.21 Consider the following queue of characters, where QUEUE is a circular array which is allocated six memory cells:

FRONT = 2, REAR = 4 QUEUE: __, A, C, D, __, __

(For notational convenience, we use "__" to denote an empty memory cell.) Describe the queue as the following operations take place:

(*a*) F is added to the queue.
(*b*) two letters are deleted.
(*c*) K, L and M are added to the queue.
(*d*) two letters are deleted.
(*e*) R is added to the queue.
(*f*) two letters are deleted.
(*g*) S is added to the queue.
(*h*) two letters are deleted.
(*i*) one letter is deleted.
(*j*) one letter is deleted.

(*a*) F is added to the rear of the queue, yielding

FRONT = 2, REAR = 5 QUEUE: __, A, C, D, F, __

Note that REAR is increased by 1.

(*b*) The two letters, A and C, are deleted, leaving

FRONT = 4, REAR = 5 QUEUE: __, __, __, D, F, __

Note that FRONT is increased by 2.

(*c*) K, L and M are added to the rear of the queue. Since K is placed in the last memory cell of QUEUE, L and M are placed in the first two memory cells. This yields

FRONT = 4, REAR = 2 QUEUE: L, M, __, D, F, K

Note that REAR is increased by 3 but the arithmetic is modulo 6:

$$REAR = 5 + 3 = 8 = 2 \pmod 6$$

(*d*) The two front letters, D and F are deleted, leaving

FRONT = 6, REAR = 2 QUEUE: L, M, __, __, __, K

(*e*) R is added to the rear of the queue, yielding

FRONT = 6, REAR = 3 QUEUE: L, M, R, __, __, K

(*f*) The two front letters, K and L, are deleted, leaving

FRONT = 2, REAR = 3 QUEUE: __, M, R, __, __, __

Note that FRONT is increased by 2 but the arithmetic is modulo 6:

$$FRONT = 6 + 2 = 8 = 2 \pmod 6$$

(*g*) S is added to the rear of the queue, yielding

FRONT = 2, REAR = 4 QUEUE: __, M, R, S, __, __

(*h*) The two front letters, M and R, are deleted, leaving

FRONT = 4, REAR = 4 QUEUE: __, __, __, S, __.

(*i*) The front letter S is deleted. Since FRONT = REAR, this means that the queue is empty; hence we assign NULL to FRONT and REAR. Thus

FRONT = 0, REAR = 0 QUEUE: __, __, __, __, __, __

(*j*) Since FRONT = NULL, no deletion can take place. That is, underflow has occurred.

6.22 Suppose each data structure is stored in a circular array with N memory cells.

(*a*) Find the number NUMB of elements in a queue in terms of FRONT and REAR.

(*b*) Find the number NUMB of elements in a deque in terms of LEFT and RIGHT.

(*c*) When will the array be filled?

(*a*) If FRONT ≤ REAR, then NUMB = REAR − FRONT + 1. For example, consider the following queue with N = 12:

FRONT = 3, REAR = 9 QUEUE: __, __, *, *, *, *, *, *, *, *, __, __, __

Then NUMB = 9 − 3 + 1 = 7, as pictured.

If REAR < FRONT, then FRONT − REAR − 1 is the number of empty cells, so

NUMB = N − (FRONT − REAR − 1) = N + REAR − FRONT + 1

For example, consider the following queue with N = 12:

FRONT = 9, REAR = 4 QUEUE: *, *, *, *, __, __, __, __ *, *, *

Then NUMB = 12 + 4 − 9 + 1 = 8, as pictured.

Using arithmetic modulo N, we need only one formula, as follows:

NUMB = REAR − FRONT + 1 (mod N)

(*b*) The same result holds for deques except that FRONT is replaced by RIGHT. That is,

NUMB = RIGHT − LEFT + 1 (mod N)

(*c*) With a queue, the array is full when

(i) FRONT = 1 and REAR = N or (ii) FRONT = REAR + 1

Similarly, with a deque, the array is full when

(i) LEFT = 1 and RIGHT = N or (ii) LEFT = RIGHT + 1

Each of these conditions implies NUMB = N.

6.23 Consider the following deque of characters where DEQUE is a circular array which is allocated six memory cells:

LEFT = 2, RIGHT = 4 DEQUE: __, A, C, D, __, __

Describe the deque while the following operations take place.

(*a*) F is added to the right of the deque.

(*b*) Two letters on the right are deleted.

(*c*) K, L and M are added to the left of the deque.

(*d*) One letter on the left is deleted.

(*e*) R is added to the left of the deque.

(*f*) S is added to the right of the deque.

(*g*) T is added to the right of the deque.

(*a*) F is added on the right, yielding

LEFT = 2, RIGHT = 5 DEQUE: __, A, C, D, F, __

Note that RIGHT is increased by 1.

(*b*) The two right letters, F and D, are deleted, yielding

LEFT = 2, RIGHT = 3 DEQUE: __, A, C, __, __, __

Note that RIGHT is decreased by 2.

(*c*) K, L and M are added on the left. Since K is placed in the first memory cell, L is placed in the last memory cell and M is placed in the next-to-last memory cell. This yields

LEFT = 5, RIGHT = 3 DEQUE: K, A, C, __, M, L

Note that LEFT is decreased by 3 but the arithmetic is modulo 6:

$$\text{LEFT} = 2 - 3 = -1 = 5 \pmod 6$$

(*d*) The left letter, M, is deleted, leaving

LEFT = 6, RIGHT = 3 DEQUE: K, A, C, _, __, L

Note that LEFT is increased by 1.

(*e*) R is added on the left, yielding

LEFT = 5, RIGHT = 3 DEQUE: K, A, C, __, R, L

Note that LEFT is decreased by 1.

(*f*) S is added on the right, yielding

LEFT = 5, RIGHT = 4 DEQUE: K, A, C, S, R, L

(*g*) Since LEFT = RIGHT + 1, the array is full, and hence T cannot be added to the deque. That is, overflow has occurred.

6.24 Consider a deque maintained by a circular array with N memory cells.

(*a*) Suppose an element is added to the deque. How is LEFT or RIGHT changed?

(*b*) Suppose an element is deleted. How is LEFT or RIGHT changed?

(*a*) If the element is added on the left, then LEFT is decreased by 1 (mod N). On the other hand, if the element is added on the right, then RIGHT is increased by 1 (mod N).

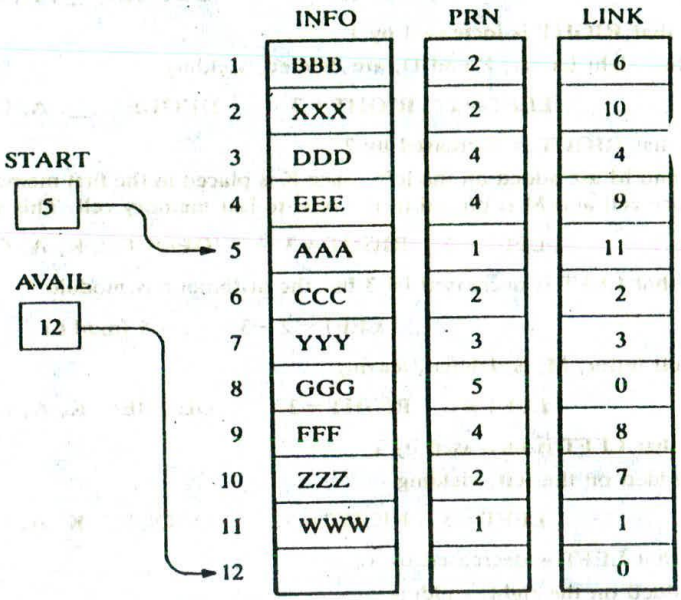
(*b*) If the element is deleted from the left, then LEFT is increased by 1 (mod N). However if the element is deleted from the right, then RIGHT is decreased by 1 (mod N). In the case that LEFT = RIGHT before the deletion (that is, when the deque has only one element), then LEFT and RIGHT are both assigned NULL to indicate that the deque is empty.

PRIORITY QUEUES

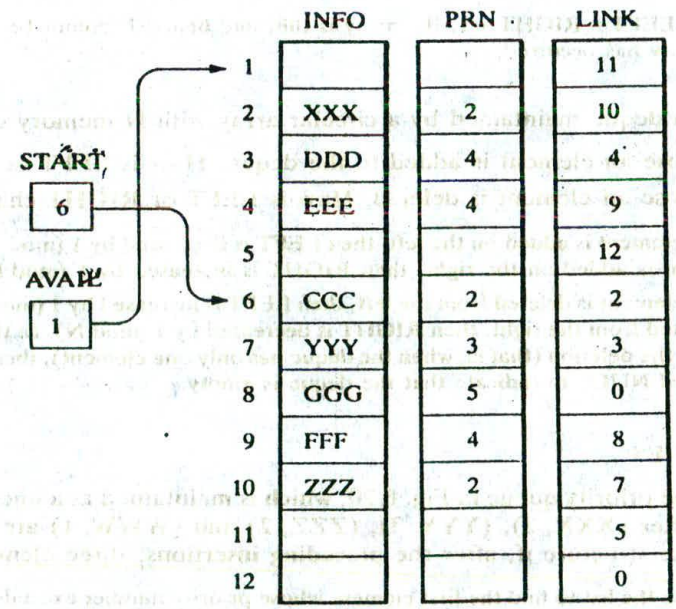
6.25 Consider the priority queue in Fig. 6-20, which is maintained as a one-way list. (*a*) Describe the structure after (XXX, 2), (YYY, 3), (ZZZ, 2) and (WWW, 1) are added to the queue. (*b*) Describe the structure if, after the preceding insertions, three elements are deleted.

(*a*) Traverse the list to find the first element whose priority number exceeds that of XXX. It is DDD, so insert XXX before DDD (after CCC) in the first empty cell, INFO[2]. Then traverse the list to find the first element whose priority number exceeds that of YYY. Again it is DDD. Hence insert YYY before DDD (after XXX) in the next empty cell, INFO[7]. Then traverse the list to find the first

element whose priority number exceeds that of ZZZ. It is YYY. Hence insert ZZZ before YYY (after XXX) in the next empty cell, INFO[10]. Last, traverse the list to find the first element whose priority number exceeds that of WWW. It is BBB. Hence insert WWW before BBB (after AAA) in the next empty cell, INFO[11]. This finally yields the structure in Fig. 6-26(*a*).



(*a*)



(*b*)

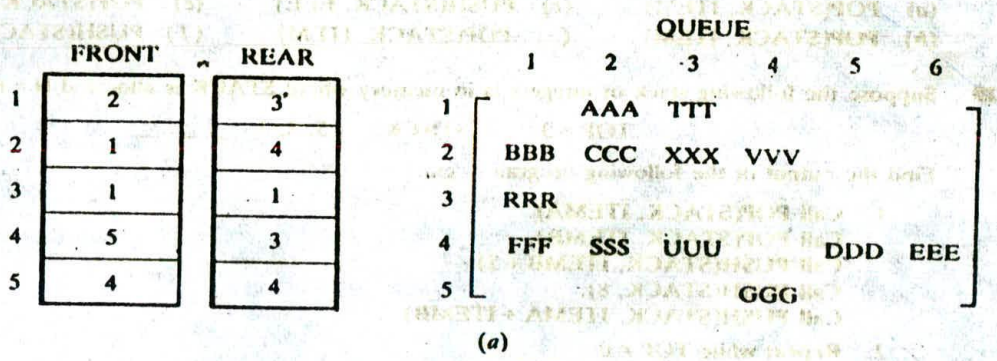
Fig. 6-26

(*b*) The first three elements in the one-way list are deleted. Specifically, first AAA is deleted and its memory cell INFO[5] is added to the AVAIL list. Then WWW is deleted and its memory cell INFO[11] is added to the AVAIL list. Last, BBB is deleted and its memory cell INFO[1] is added to the AVAIL list. This finally yields the structure in Fig. 6-26(*b*).

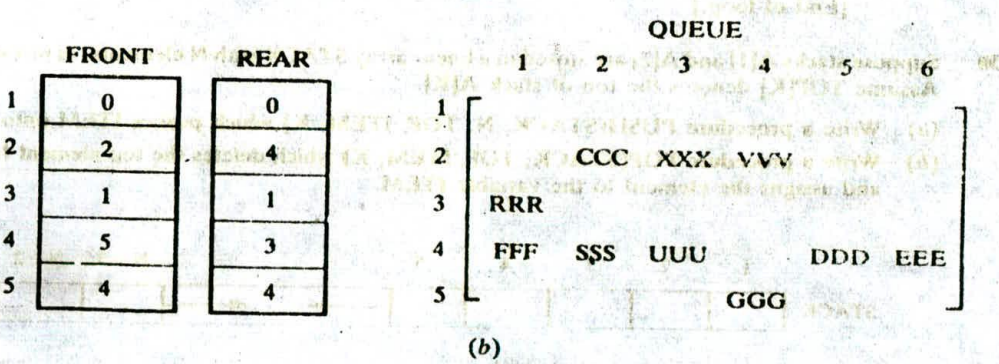
Remark: Observe that START and AVAIL are changed accordingly.

6.26 Consider the priority queue in Fig. 6-22, which is maintained by a two-dimensional array QUEUE. (*a*) Describe the structure after (RRR, 3), (SSS, 4), (TTT, 1), (UUU, 4) and (VVV, 2) are added to the queue. (*b*) Describe the structure if, after the preceding insertions, three elements are deleted.

(*a*) Insert each element in its priority row. That is, add RRR as the rear element in row 3, add SSS as the rear element in row 4, add TTT as the rear element in row 1, add UUU as the rear element in row 4 and add VVV as the rear element in row 2. This yields the structure in Fig. 6-27(*a*). (As noted previously, insertions with this array representation are usually simpler than insertions with the one-way list representation.)



(*a*)



(*b*)

Fig. 6-27

(*b*) First delete the elements with the highest priority in row 1. Since row 1 contains only two elements, AAA and TTT, then the front element in row 2, BBB, must also be deleted. This finally leaves the structure in Fig. 6-27(*b*).

Remark: Observe that, in both cases, FRONT and REAR are changed accordingly.

Supplementary Problems

STACKS

6.27 Consider the following stack of city names:

STACK: London, Berlin, Rome, Paris, _____, _____

(*a*) Describe the stack as the following operations take place:

(i) PUSH(STACK, Athens), (iii) POP(STACK, ITEM) (v) PUSH(STACK, Moscow)
(ii) POP(STACK, ITEM) (iv) PUSH(STACK, Madrid) (vi) POP(STACK, ITEM)

(*b*) Describe the stack if the operation POP(STACK, ITEM) deletes London.

6.28 Consider the following stack where STACK is allocated N = 4 memory cells:

STACK: AAA, BBB, _____, _____

Describe the stack as the following operations take place:

(*a*) POP(STACK, ITEM) (*c*) PUSH(STACK, EEE) (*e*) POP(STACK, ITEM)
(*b*) POP(STACK, ITEM) (*d*) POP(STACK, ITEM) (*f*) PUSH(STACK, GGG)

6.29 Suppose the following stack of integers is in memory where STACK is allocated N = 6 memory cells:

TOP = 3 STACK: 5, 2, 3, __, __, __

Find the output of the following program segment:

1. Call POP(STACK, ITEMA).
   Call POP(STACK, ITEMB).
   Call PUSH(STACK, ITEMB + 2).
   Call PUSH(STACK, 8).
   Call PUSH(STACK, ITEMA + ITEMB).
2. Repeat while TOP ≠ 0:
   Call POP(STACK, ITEM).
   Write: ITEM.
   [End of loop.]

6.30 Suppose stacks A[1] and A[2] are stored in a linear array STACK with N elements, as pictured in Fig. 6-28. Assume TOP[K] denotes the top of stack A[K].

(*a*) Write a procedure PUSH(STACK, N, TOP, ITEM, K) which pushes ITEM onto stack A[K].

(*b*) Write a procedure POP(STACK, TOP, ITEM, K) which deletes the top element from stack A[K] and assigns the element to the variable ITEM.

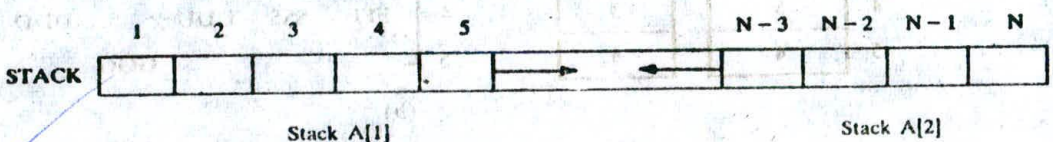


Fig. 6-28

ARITHMETIC EXPRESSIONS; POLISH EXPRESSIONS

6.31 Translate, by inspection and hand, each infix expression into its equivalent postfix expression:

(*a*) (A − B)/(D + E) * F) (*b*) ((A + B)/D) ↑ ((E − F) * G)

6.32 Translate, by inspection and hand, each infix expression in Prob. 6.31 into its equivalent prefix expression.

6.33 Evaluate each of the following parenthesis-free arithmetic expressions:

(*a*) 5 + 3 ↑ 2 − 8 / 4 * 3 + 6

(*b*) 6 + 2 ↑ 3 + 9 / 3 − 4 * 5

6.34 Consider the following parenthesis-free arithmetic expression:

E: 6 + 2 ↑ 3 ↑ 2 − 4 * 5

Evaluate the expression E, (*a*) assuming that exponentiation is performed from left to right, as are the other operations, and (*b*) assuming that exponentiation is performed from right to left.

6.35 Consider each of the following postfix expressions:

P_1: 5, 3, +, 2, *, 6, 9, 7, −, /, −

P_2: 3, 5, +, 6, 4, −, *, 4, 1, −, 2, ↑, +

P_3: 3, 1, +, 2, ↑, 7, 4, −, 2, *, +, 5, −

Translate, by inspection and hand, each expression into infix notation and then evaluate.

6.36 Evaluate each postfix expression in Prob. 6.35, using Algorithm 6.3.

6.37 Use Algorithm 6.4 to translate each infix expression into its equivalent postfix expression:

(*a*) (A − B)/((D + E) * F) (*b*) ((A + B)/D)↑((E − F) * G)

(Compare with Prob. 6.31.)

RECURSION

6.38 Let J and K be integers and suppose Q(J, K) is recursively defined by

$$Q(J, K) = \begin{cases} 5 & \text{if } J < K \\ Q(J - K, K + 2) + J & \text{if } J \geq K \end{cases}$$

Find Q(2, 7), Q(5, 3) and Q(15, 2).

6.39 Let A and B be nonnegative integers. Suppose a function GCD is recursively defined as follows:

$$GCD(A, B) = \begin{cases} GCD(B, A) & \text{if } A < B \\ A & \text{if } B = 0 \\ GCD(B, MOD(A, B)) & \text{otherwise} \end{cases}$$

(Here MOD(A, B), read "A modulo B," denotes the remainder when A is divided by B.) (*a*) Find GCD(6, 15), GCD(20, 28) and GCD(540, 168). (*b*) What does this function do?

6.40 Let N be an integer and suppose H(N) is recursively defined by

$$H(N) = \begin{cases} 3 * N & \text{if } N < 5 \\ 2 * H(N - 5) + 7 & \text{otherwise} \end{cases}$$

(*a*) Find the base criteria of H and (*b*) find H(2), H(8) and H(24).

6.41 Use Definition 6.3 (of the Ackermann function) to find $A(2, 2)$.

6.42 Let M and N be integers and suppose F(M, N) is recursively defined by

$$F(M, N) = \begin{cases} 1 & \text{if } M = 0 \text{ or } M \geq N \geq 1 \\ F(M-1, N) + F(M-1, N-1) & \text{otherwise} \end{cases}$$

(*a*) Find F(4, 2), F(1, 5) and F(2, 4). (*b*) When is F(M, N) undefined?

6.43 Let A be an integer array with N elements. Suppose X is an integer function defined by

$$X(K) = X(A, N, K) = \begin{cases} 0 & \text{if } K = 0 \\ X(K-1) + A(K) & \text{if } 0 < K \leq N \\ X(K-1) & \text{if } K > N \end{cases}$$

Find X(5) for each of the following arrays:

(*a*) N = 8, A: 3, 7, −2, 5, 6, −4, 2, 7 (*b*) N = 3, A: 2, 7, −4

What does this function do?

6.44 Show that the recursive solution to the Towers of Hanoi problem in Sec. 6.7 requires $f(n) = 2^n - 1$ moves for n disks. Show that no other solution uses fewer than $f(n)$ moves.

6.45 Suppose S is a string with N characters. Let SUB(S, J, L) denote the substring of S beginning in the position J and having length L. Let A//B denote the concatenation of strings A and B. Suppose REV(S, N) is recursively defined by

$$REV(S, N) = \begin{cases} S & \text{if } N = 1 \\ SUB(S, N, 1)//REV(SUB(S, 1, N-1), N-1) & \text{otherwise} \end{cases}$$

(*a*) Find REV(S, N) when (i) N = 3, S = *abc* and (ii) N = 5, S = *ababc*. (*b*) What does this function do?

QUEUES; DEQUES

6.46 Consider the following queue where QUEUE is allocated 6 memory cells:

FRONT = 2, REAR = 5 QUEUE: _____, London, Berlin, Rome, Paris, _____

Describe the queue, including FRONT and REAR, as the following operations take place: (*a*) Athens is added, (*b*) two cities are deleted, (*c*) Madrid is added, (*d*) Moscow is added, (*e*) three cities are deleted and (*f*) Oslo is added.

6.47 Consider the following deque where DEQUE is allocated 6 memory cells:

LEFT = 2, RIGHT = 5 DEQUE: _____, London, Berlin, Rome, Paris, _____

Describe the deque, including LEFT and RIGHT, as the following operations take place:

(*a*) Athens is added on the left.
(*b*) Two cities are deleted from the right.
(*c*) Madrid is added on the left.
(*d*) Moscow is added on the right.
(*e*) Two cities are deleted from the right.
(*f*) A city is deleted from the left.
(*g*) Oslo is added on the left.

6.48 Suppose a queue is maintained by a circular array QUEUE with N = 12 memory cells. Find the number of elements in QUEUE if (*a*) FRONT = 4, REAR = 8; (*b*) FRONT = 10, REAR = 3; and (*c*) FRONT = 5, REAR = 6 and then two elements are deleted.

6.49 Consider the priority queue in Fig. 6-26(*b*), which is maintained as a one-way list.

(*a*) Describe the structure if two elements are deleted.
(*b*) Describe the structure if, after the preceding deletions, the elements (RRR, 3), (SSS, 1), (TTT, 3) and (UUU, 2) are added to the queue.
(*c*) Describe the structure if, after the preceding insertions, three elements are deleted.

6.50 Consider the priority queue in Fig. 6-27(*b*), which is maintained by a two-dimensional array QUEUE.

(*a*) Describe the structure if two elements are deleted.
(*b*) Describe the structure if, after the preceding deletions, the elements (JJJ, 3), (KKK, 1), (LLL, 4) and (MMM, 5) are added to the queue.
(*c*) Describe the structure if, after the preceding insertions, six elements are deleted.

Programming Problems

6.51 Translate Quicksort into a subprogram QUICK(A, N) which sorts the array A with N elements. Test the program using

(*a*) 44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66
(*b*) D, A, T, A, S, T, R, U, C, T, U, R, E, S

6.52 Write a program which gives the solution to the Towers of Hanoi problem for n disks. Test the program using (*a*) $n = 3$ and (*b*) $n = 4$.

6.53 Translate Algorithm 6.4 into a subprogram POLISH(Q, P) which transforms an infix expression Q into its equivalent postfix expression P. Assume each operand is a single alphabetic character, and use the usual symbols for addition (+), subtraction (−), multiplication (*) and division (/), but use the symbol ↑ or $ for exponentiation. (Some programming languages do not accept ↑.) Test the program using
(*a*) ((A + B) * D) $ (E − F) (*b*) A + (B * C − (D / E $ F) * G) * H

6.54 Suppose a priority queue is maintained as a one-way list as illustrated in Fig. 6-20.

(*a*) Write a procedure

INSPQL(INFO, PRN, LINK, START, AVAIL, ITEM, N)

which adds an ITEM with priority number N to the queue. (See Algorithm 6.14.)

(*b*) Write a procedure

DELPQL(INFO, PRN, LINK, START, AVAIL, ITEM)

which removes an element from the queue and assigns the element to the variable ITEM. (See Algorithm 6.13.)

Test the procedures, using the data in Prob. 6.25.

6.55 Suppose a priority queue is maintained by a two-dimensional array as illustrated in Fig. 6-22.

(*a*) Write a procedure

INSPQA(QUEUE, FRONT, REAR, ITEM, M)

which adds an ITEM with priority number M to the queue. (See Algorithm 6.16.)

(*b*) Write a procedure

DELPQA(QUEUE, FRONT, REAR, ITEM)

which removes an element from the queue and assigns the element to the variable ITEM. (See Algorithm 6.15.)

Test the procedures, using the data in Prob. 6.26. (Assume that QUEUE has ROW number of rows and COL number of columns, where ROW and COL are global variables.)