Chapter Four

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 Chapter 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.

4.2

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 Chapter 5.

4.2 LINEAR ARRAYS, one dimensional array

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, ..., 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)

$$A(1), A(2), \ldots, A(N)$$

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).

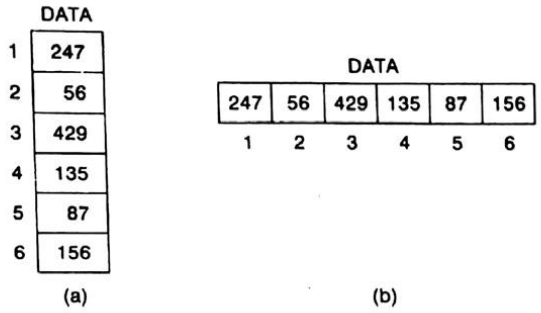


Fig. 4.1

(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),

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

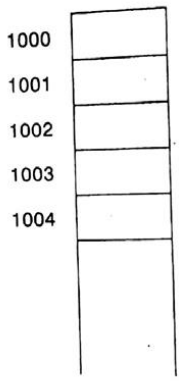


Fig. 4.2 *Computer Memory*

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

$$Base(\text{LA})$$

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

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

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.

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):

$$\begin{aligned} \text{LOC(AUTO[1965])} &= \textit{Base}(\text{AUTO}) + w(1965 - \text{lower bound}) \\ &= 200 + 4(1965 - 1932) = 332 \end{aligned}$$

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

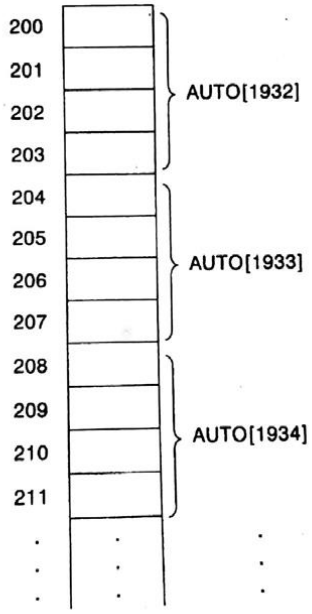


Fig. 4.3

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 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

TEST[4] := X

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

TEST[5] := 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.

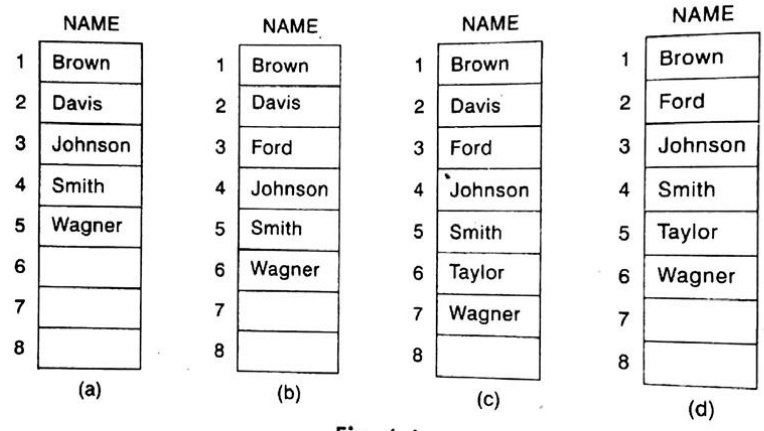


Fig. 4.4

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 Solved Problem 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 ≤ 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,

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

For example, suppose A originaily 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 Chapter 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 non-numerical 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[l], A[2], … ,A[N] is in memory. The bubble sort algorithm works as follows:

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 ≤ 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,

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

For example, suppose A originaily 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 Chapter 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 non-numerical 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[l], 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 51 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

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

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 Chapter 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

$$LOC = N + 1$$

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:

Repeat while LOC ≤ N and DATA[LOC] ≠ 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 $4 \le n$), we know that Susan is in the original array.

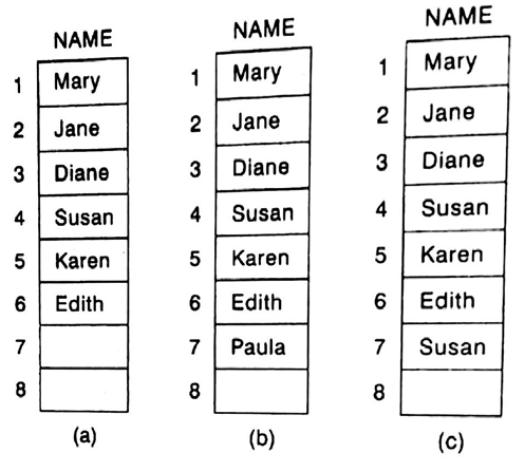


Fig. 4.5

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 $p_1 + p_2 + \ldots + p_n + q = 1$.) Since the algorithm uses k comparisons when ITEM appears in DATA[K], the average number of comparisons is given by

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

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,

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

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

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:

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

MID = INT((BEG + 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:

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], ..., 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 and so DATA[MID] = 55

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

   MID = INT[(1 + 6)/2] = 3 and so DATA[MID] = 30

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

   MID = INT[(4 + 6)/2] = 5 and so 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 and so DATA[MID] = 77

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

   MID = INT[(11 + 13)/2] = 12 and so DATA[MID] = 88

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

   MID = INT[(11 + 11)/2] = 11 and so 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 whe DATA contains n elements. Observe that each comparison reduces the sample size in half. Hen 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 sho that the running time for the average case is approximately equal to the running time for the wors 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.

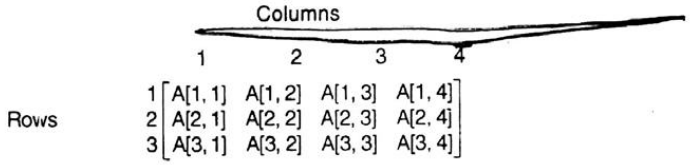


Fig. 4.8 *Two-Dimensional 3 × 4 Array A*

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, ..., m, with *lower bound* 1 and *upper bound* m; and the second dimension of A contains the *index set* 1, 2, ..., 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

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

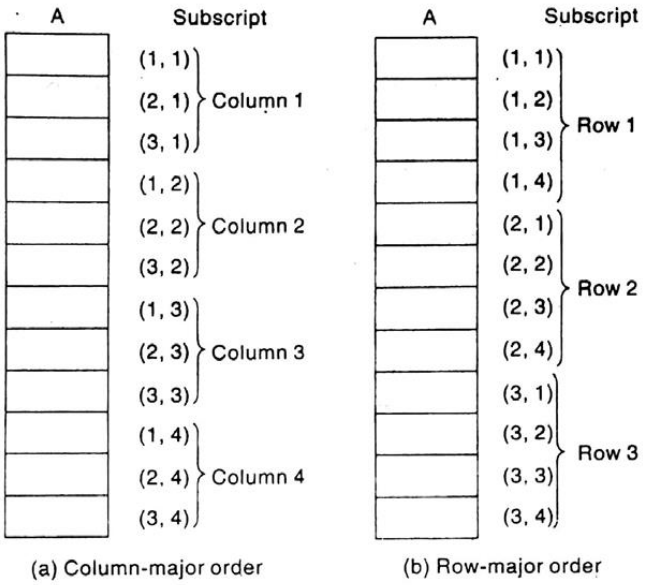
By Eq. (4.3), the length of the first dimension is equal to $5 - 2 + 1 = 4$, and the length of the second dimension is equal to $1 - (-3) + 1 = 5$. Thus NUMB contains $4 \cdot 5 = 20$ elements.

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.

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

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



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

Fig. 4.10

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

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

or the formula

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

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 $m_1 \times m_2 \times \ldots \times m_n$ array B is a collection of $m_1 \cdot m_2 \ldots m_n$ 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

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

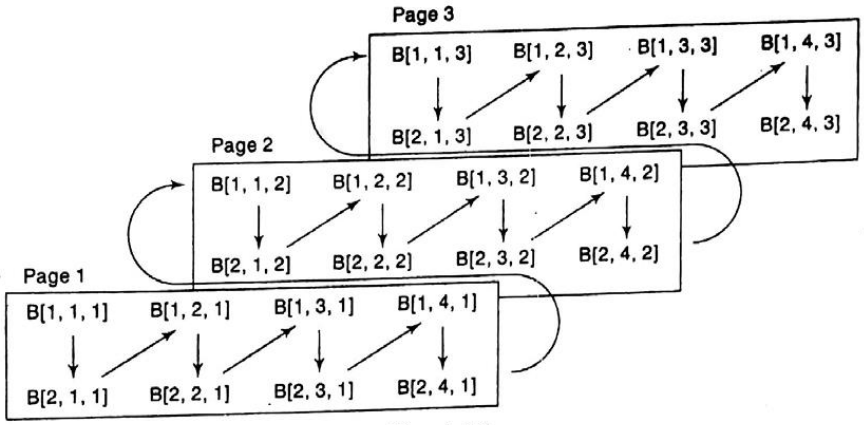


Fig. 4.11

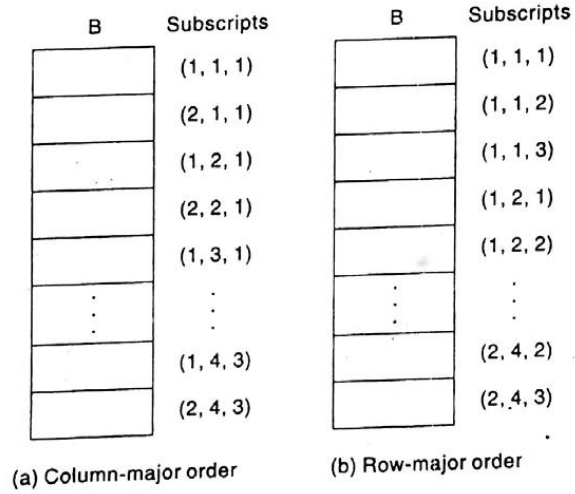


Fig. 4.12

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

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

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

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

or from the formula

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

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,

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

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*(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,

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

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,

$$LOC(MAZE[5, -1, 8]) = 200 + 4(107) = 200 + 428 = 628$$

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

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

Chapter Five

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).

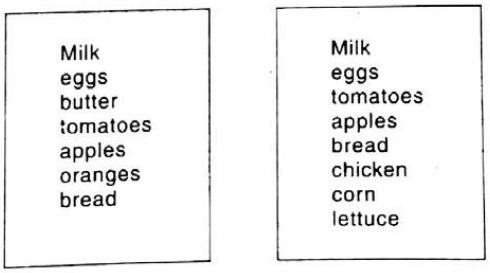


Fig. 5.1

Data processing frequently involves storing and processing data organized into lists. One way to store such data is by means of arrays, discussed in Chapter 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.

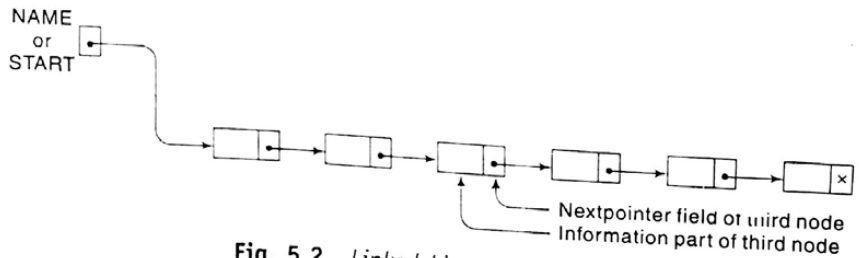


Fig. 5.2 *Linked List with 6 Nodes*

(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.)

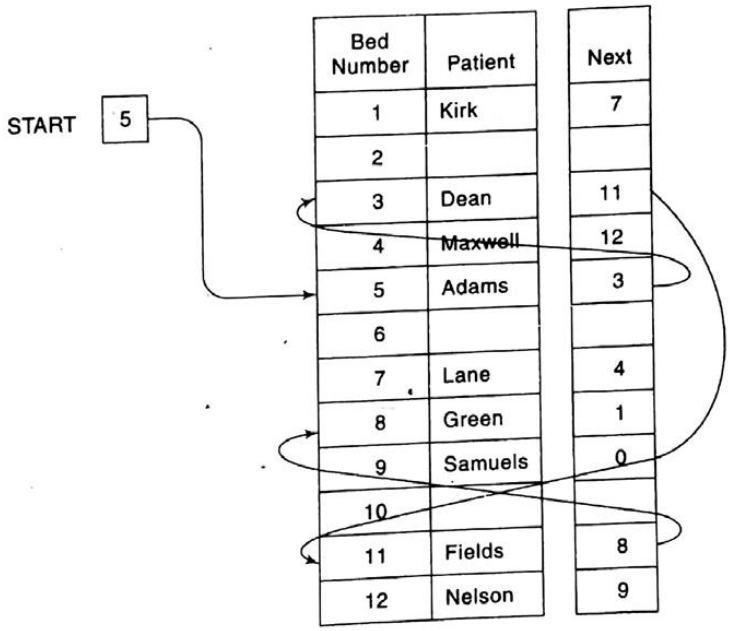


Fig. 5.3

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.

5.4

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

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.

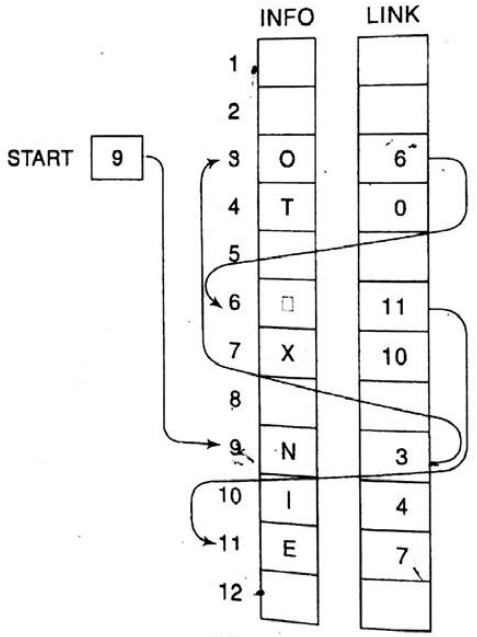


Fig. 5.4

Example 5.3

Figure 5.5 pictures show 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

and GEOM consists of the test scores

84, 62, 74, 100, 74, 78

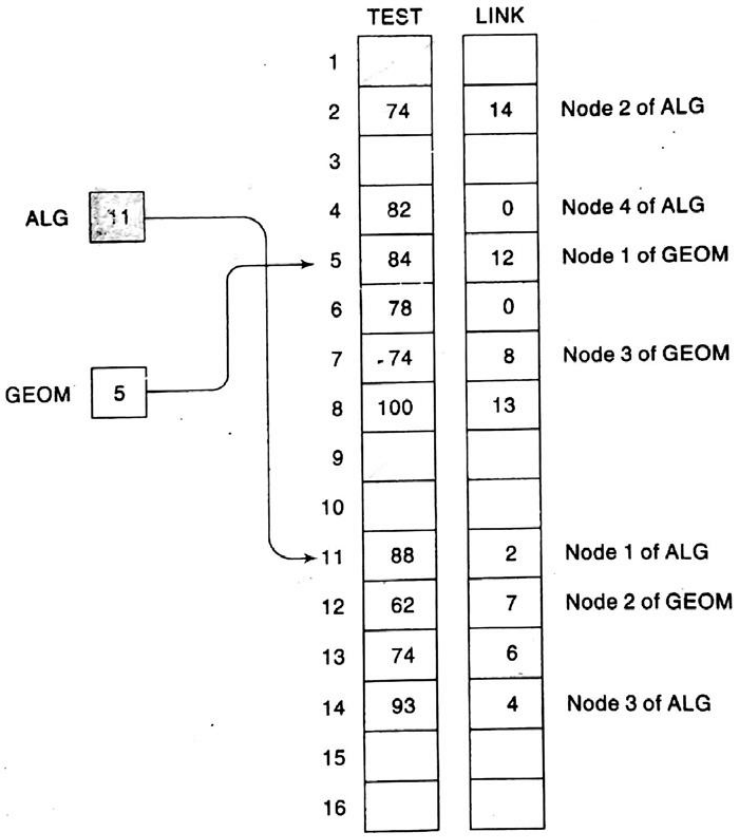


Fig. 5.5

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

5.6

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].

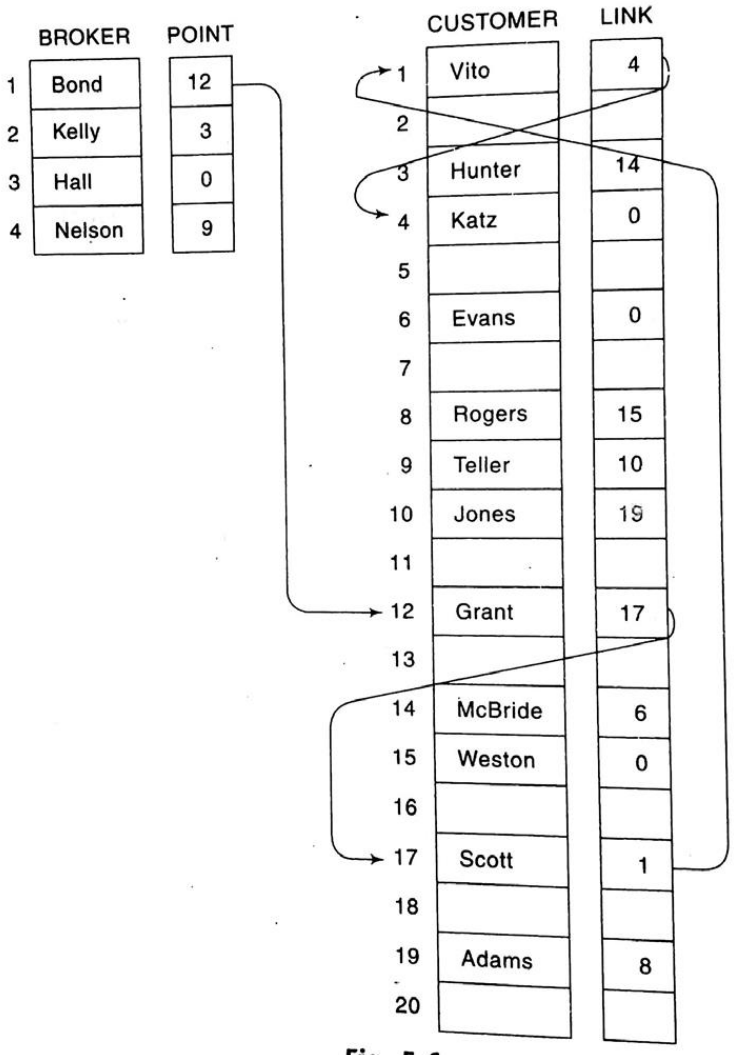


Fig. 5.6

Accordingly, Bond's list of customers, as indicated by the arrows, consists of Grant, Scott, Vito, Katz

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.

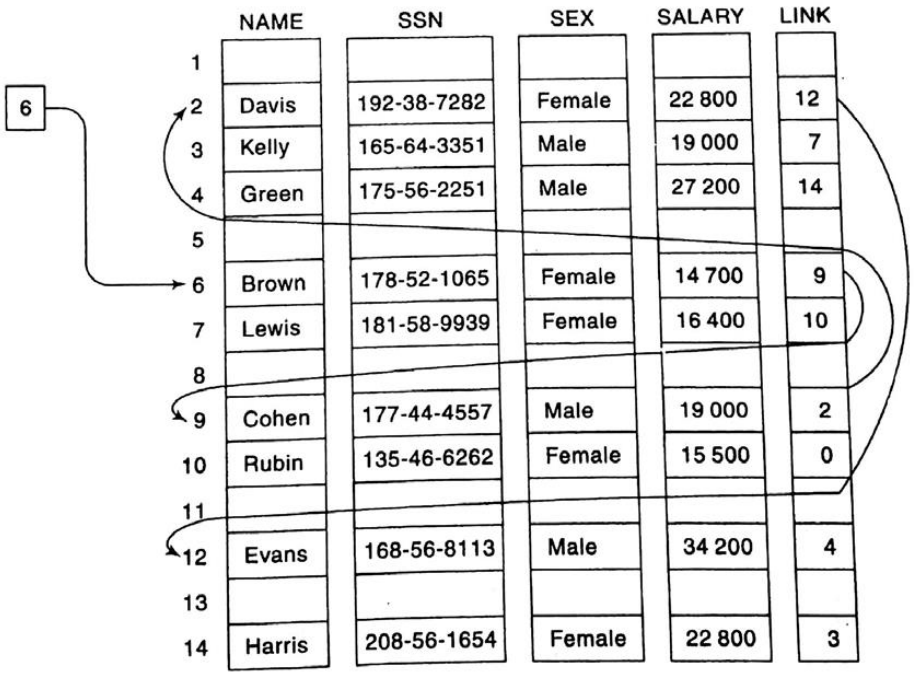


Fig. 5.7

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.

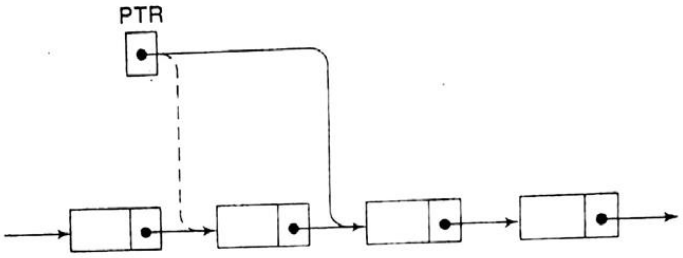


Fig. 5.8 *PTR : = LINK[PTR]*

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 Solved Problem 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.

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.
```

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 $n/2$.

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.

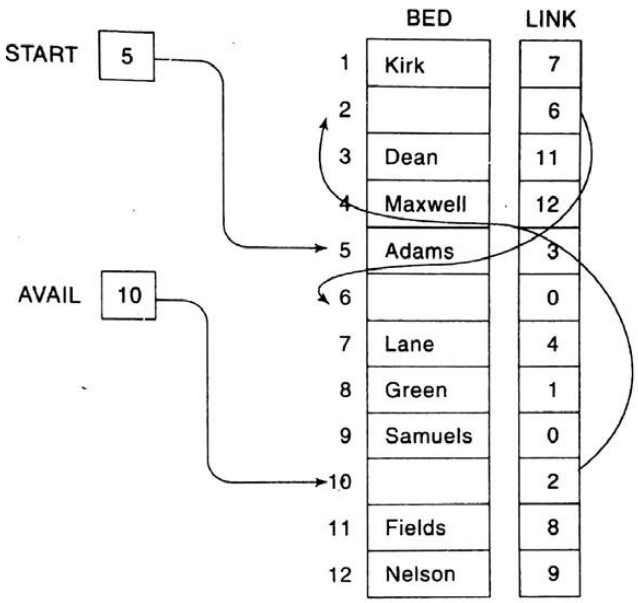


Fig. 5.9

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.

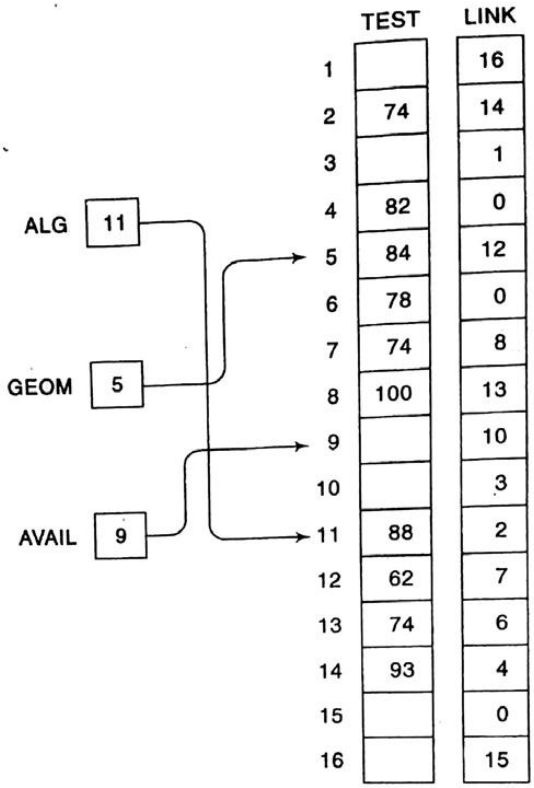


Fig. 5.10

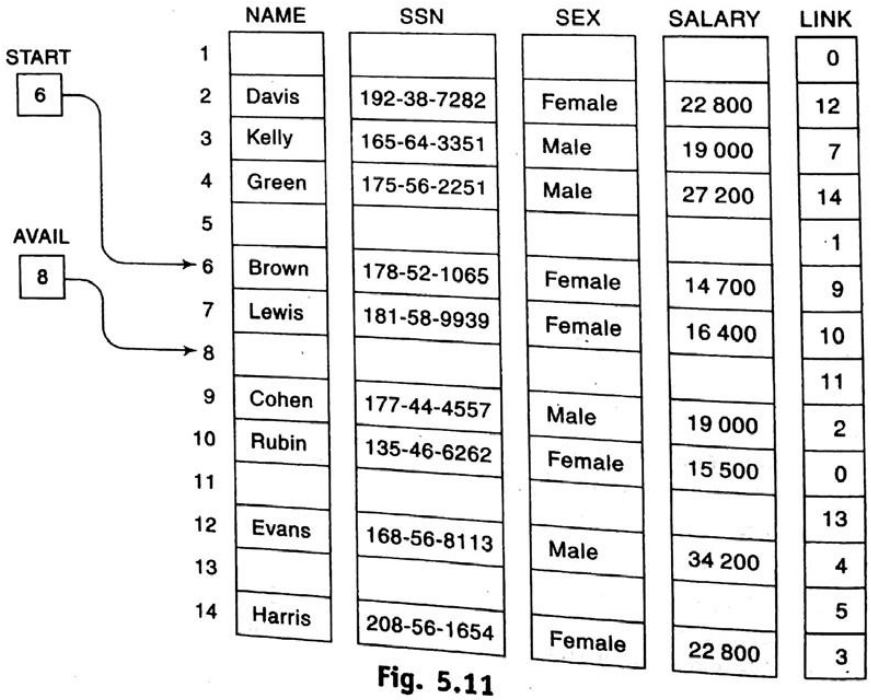


Fig. 5.11

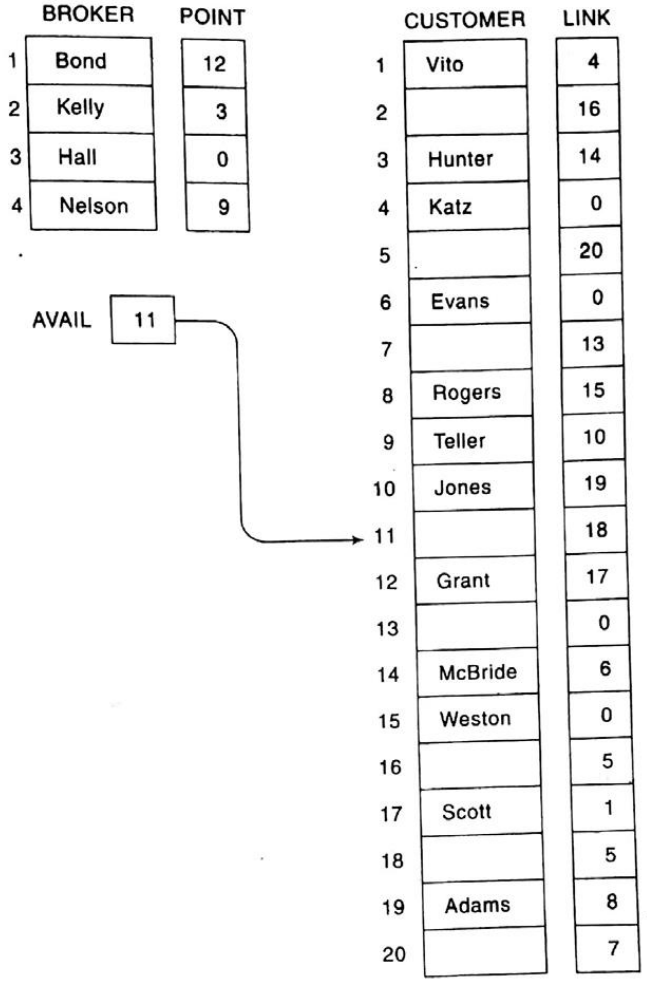


Fig. 5.12

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.

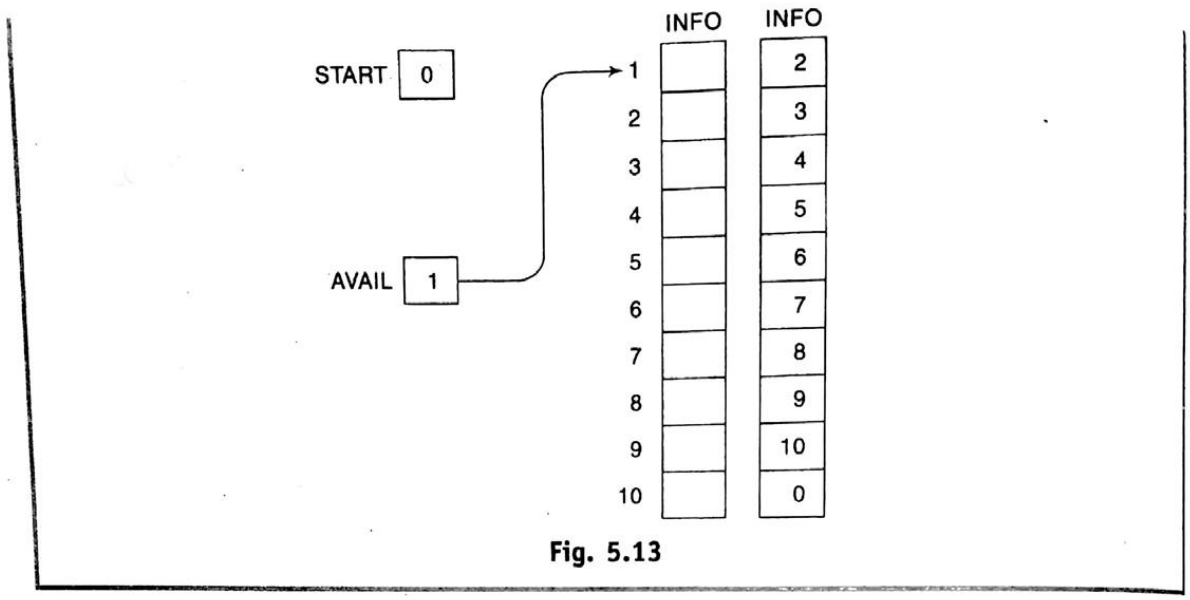


Fig. 5.13

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 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.

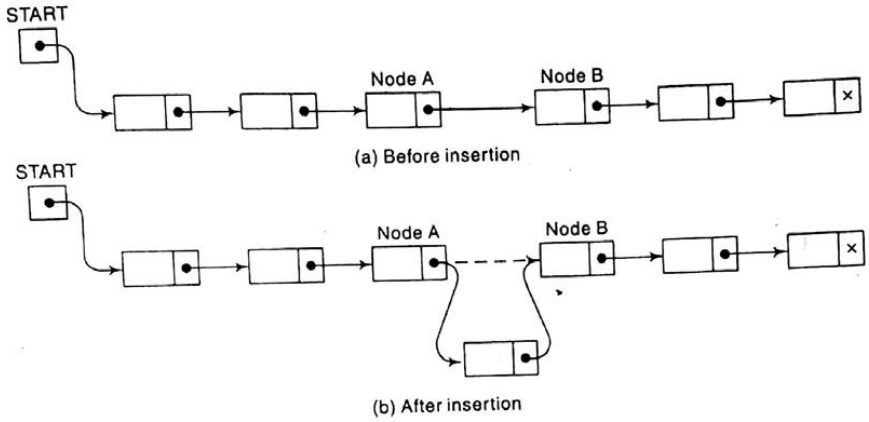


Fig. 5.14

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 thy 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.

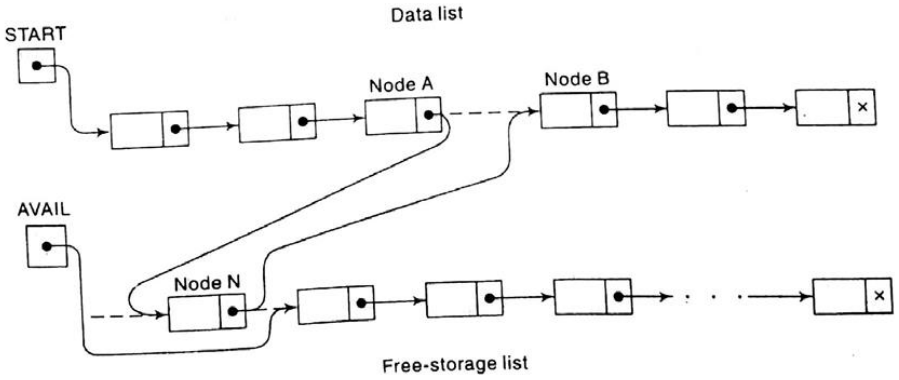


Fig. 5.15

(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

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 $n = 10$.)

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.

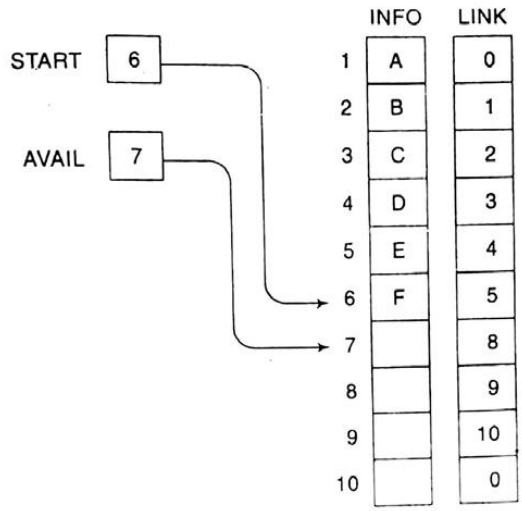


Fig. 5.16

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)

NEW := AVAIL, AVAIL := LINK[AVAIL]

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

INFO[NEW] := ITEM

The schematic diagram of the latter two steps is pictured in Fig. 5.17.

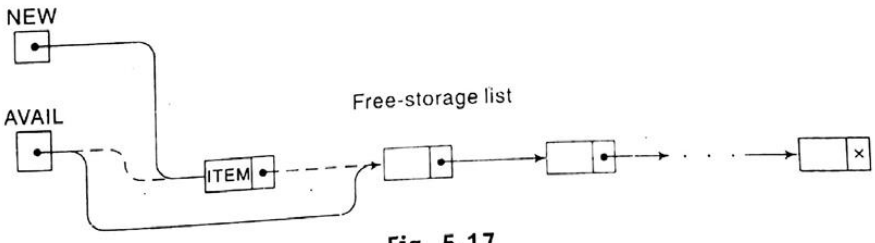


Fig. 5.17

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 beginning 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.

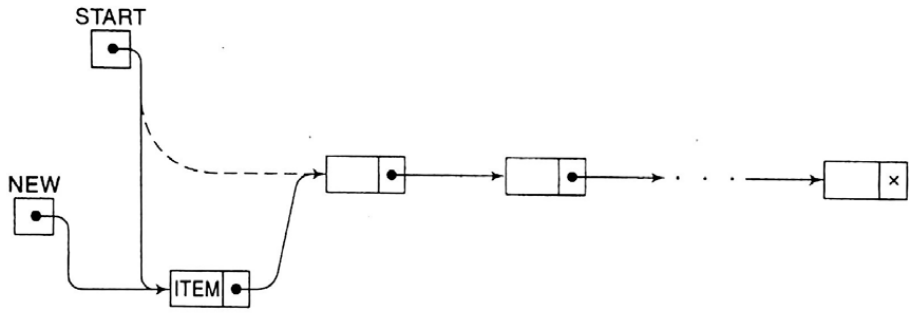


Fig. 5.18 *Insertion at the Beginning of a List*

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].

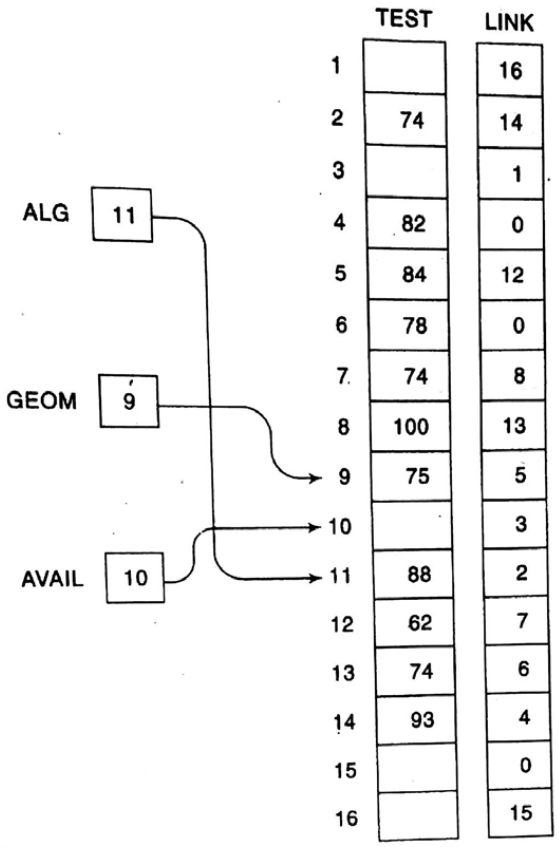


Fig. 5.19

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

LINK[NEW] := LINK[LOC]

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

LINK[LOC] := 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

SAVE := PTR and PTR := LINK[PTR]

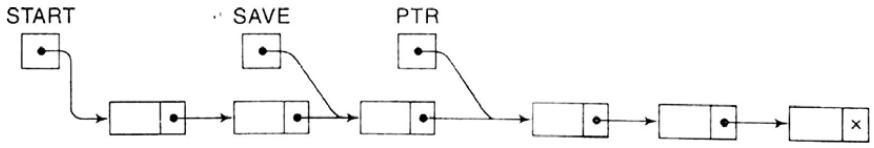


Fig. 5.20

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.

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: INSERT(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 ≠ 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 ≠ NULL, control is transferred to Step 2.
2. NEW = 10 and AVAIL = LINK[10] = 2.
3. BED[10] = Jones.

4. Since LOC ≠ 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].

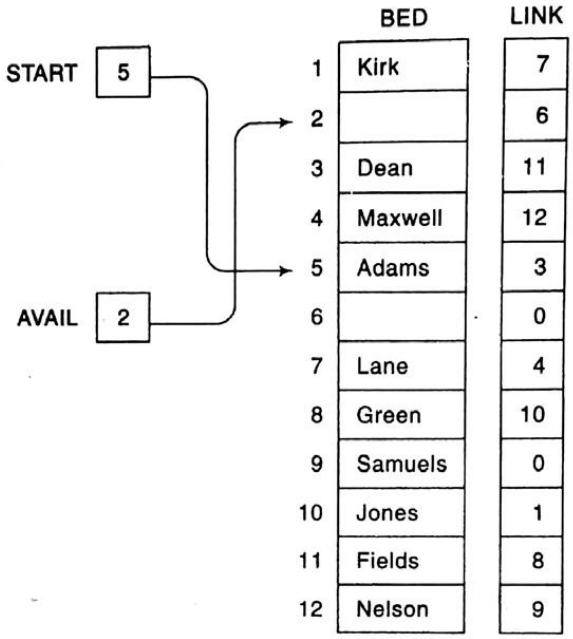


Fig. 5.21

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.

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)

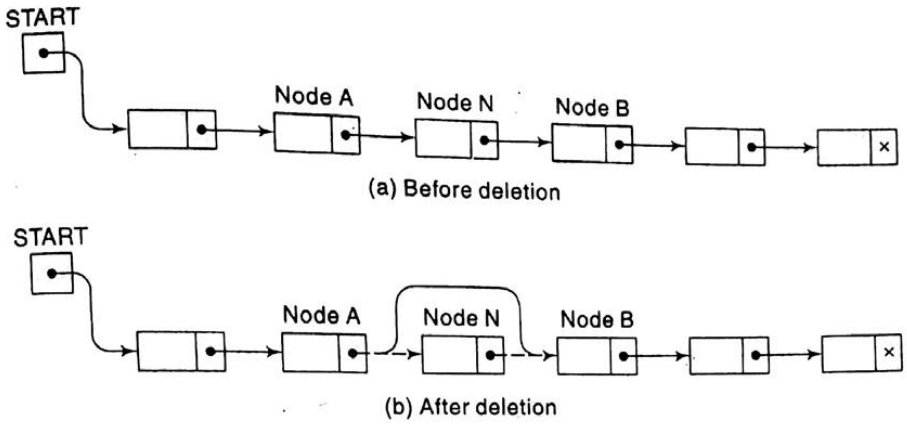


Fig. 5.22

Figure 5.22 does not take into account the fact that, when a node N is deleted from our list, we will 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.

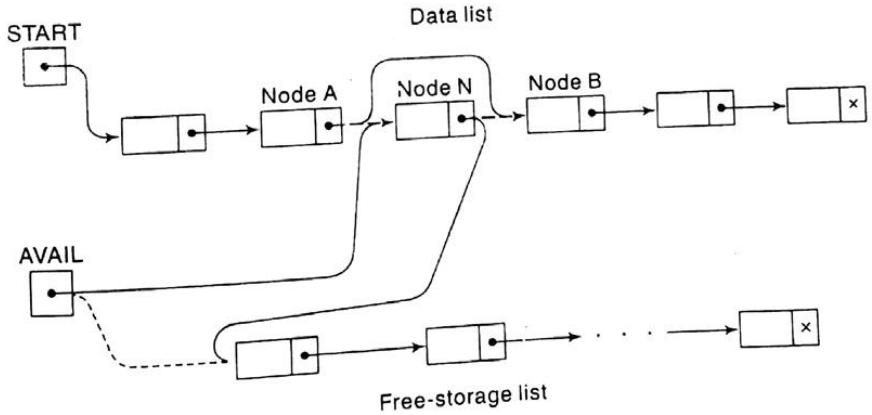


Fig. 5.23

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.

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

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.

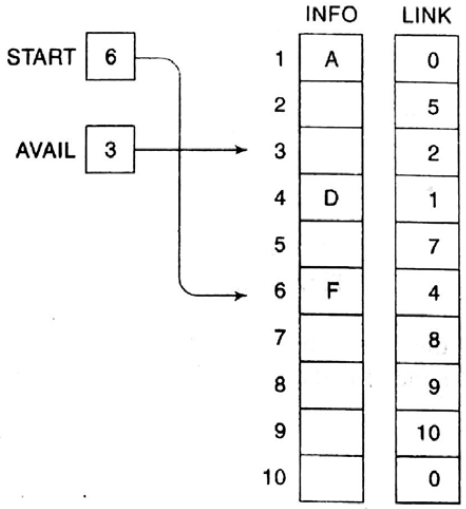


Fig. 5.24

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:

LINK[LOC] := AVAIL and then AVAIL := LOC

These two operations are pictured in Fig. 5.25.

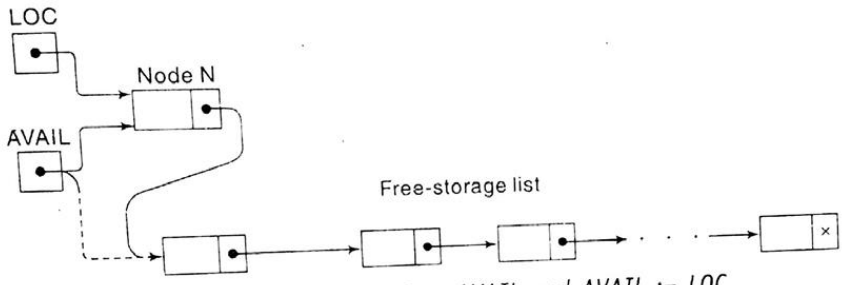


Fig. 5.25 *LINK[LOC] := AVAIL and AVAIL := LOC*

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.

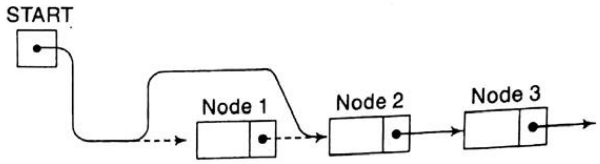


Fig. 5.26 *START := LINK[START]*

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.

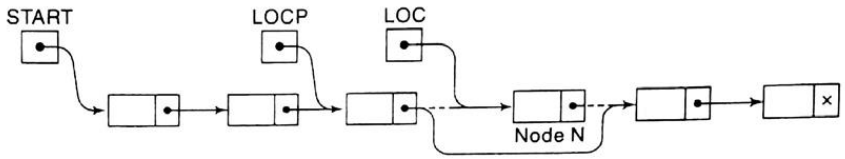


Fig. 5.27 *LINK[LOCP] := LINK[LOC]*

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.

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

SAVE := PTR and PTR := 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 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.

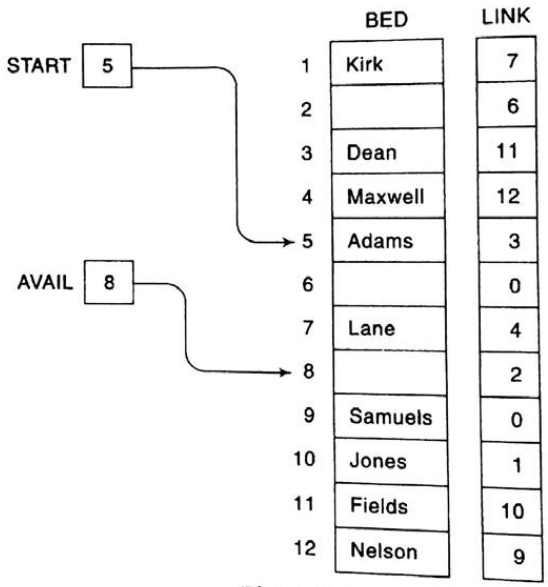


Fig. 5.28

(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.

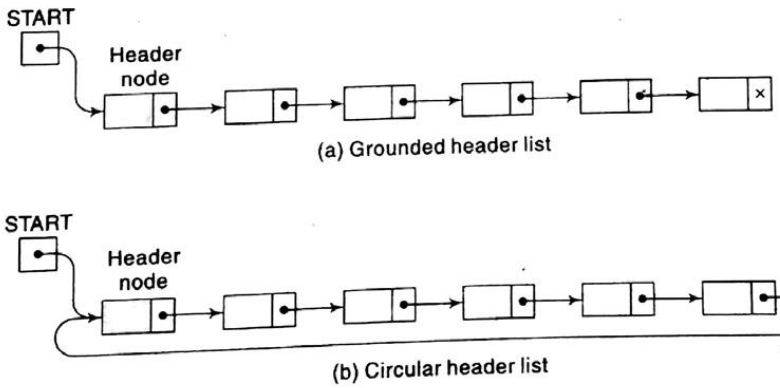


Fig. 5.29

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.

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.

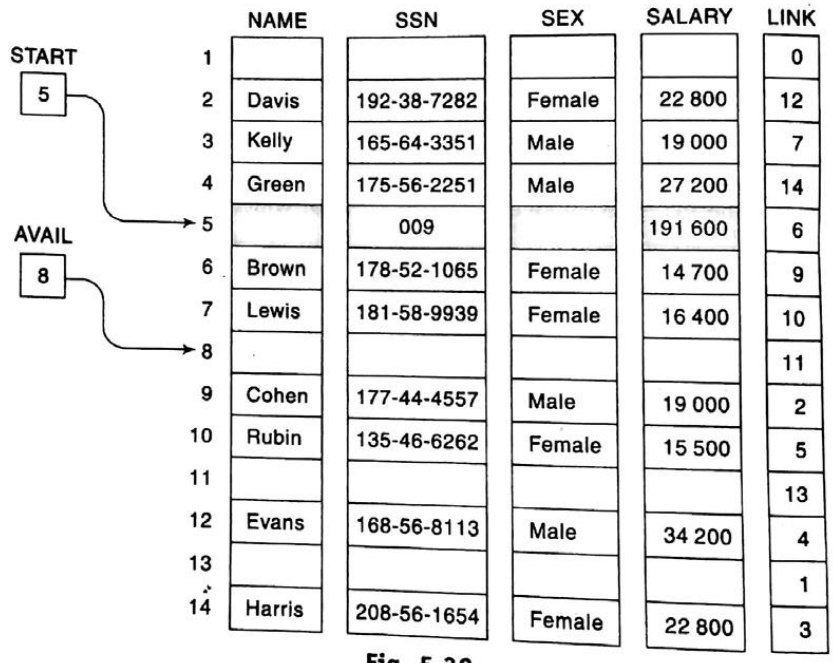


Fig. 5.30

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.

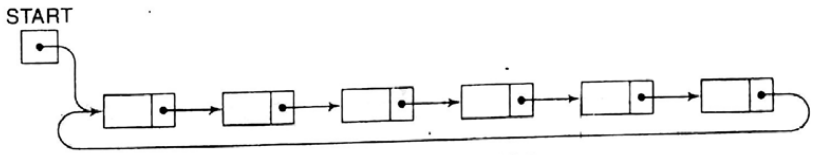
Again we did not have to consider the special case when ITEM appears in the first node, as we did in Algorithm 5.10.

Remark: There are two other variations of linked lists which sometimes appear in the literature:

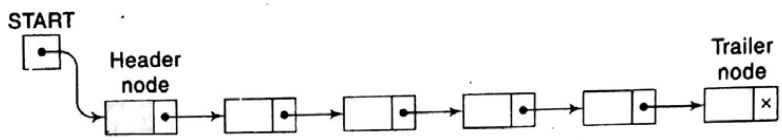
(1) A linked list whose last node points back to the first node instead of containing the null pointer, called a *circular list*

(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).

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