Chapter 4

Arrays, Records and Pointers

4.1 INTRODUCTION

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

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

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

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

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

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

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

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

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

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

4.2 LINEAR ARRAYS

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

(*a*) The elements of the array are referenced respectively by an *index set* consisting of 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 wi assume the index set consists of the integers $1, 2, \ldots, n$. In general, the length or the number of dat elements of the array can be obtained from the index set by the formula

$$\text{Length} = \text{UB} - \text{LB} + 1 \qquad (4.$$

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

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

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

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

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

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

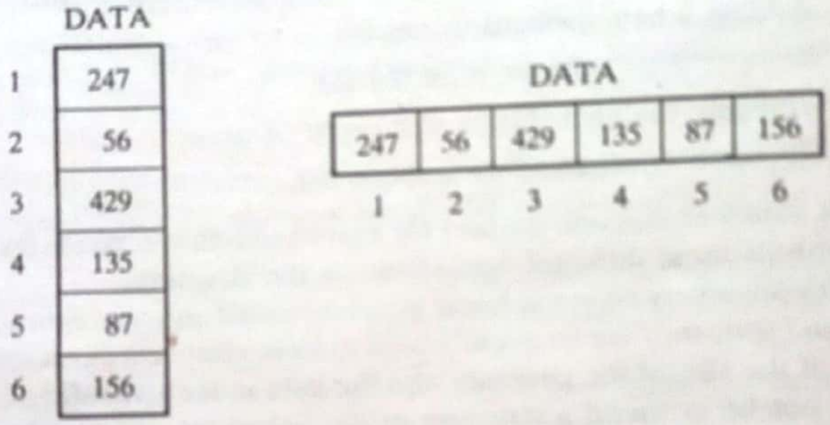
(*a*) Let DATA be a 6-element linear array of integers such that

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*).



(*a*)   (*b*)

Fig. 4-1

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

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

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

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

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.

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.

	NAME (*a*)	NAME (*b*)	NAME (*c*)	NAME (*d*)
1	Brown	Brown	Brown	Brown
2	Davis	Davis	Davis	Ford
3	Johnson	Ford	Ford	Johnson
4	Smith	Johnson	Johnson	Smith
5	Wagner	Smith	Smith	Taylor
6		Wagner	Taylor	Wagner
7			Wagner	
8				

Fig. 4-4

The following algorithm inserts a data element ITEM into the Kth position in a linear array LA with N elements. The first four steps create space in LA by moving downward one location each element from the Kth position on. We emphasize that these elements are moved in reverse order—i.e., first LA[N], then LA[N − 1], . . . , and last LA[K]; otherwise data might be erased. (See Prob. 4.3.) In more detail, we first set J := N and then, using J as a counter, decrease J each time the loop is executed until J reaches K. The next step, Step 5, inserts ITEM into the array in the space just created. Before the exit from the algorithm, the number N of elements in LA is increased by 1 to account for the new element.

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

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

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

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

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

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

4.6 SORTING; BUBBLE SORT

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

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

For example, suppose A originally is the list

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

After sorting, A is the list

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

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

Remark: The above definition of sorting refers to arranging numerical data in increasing order; this restriction is only for notational convenience. Clearly, sorting may also mean arranging numerical data in decreasing order or arranging nonnumerical data in alphabetical order. Actually, A is frequently a file of records, and sorting A refers to rearranging the records of A so that the values of a given key are ordered.

Bubble Sort

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

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

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

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

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

. . . .

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

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

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

EXAMPLE 4.7

Suppose the following numbers are stored in an array A:

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

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

Pass 1. We have the following comparisons:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

27, 33, 51, 23, 66, 13, 57, 85

27, 33, 51, 23, 13, 66, 57, 85

27, 33, 51, 23, 13, 57, 66, 85

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

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

27, 33, 23, 13, 51, 57, 66, 85

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

27, 23, 13, 33, 51, 57, 66, 85

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

23, 13, 27, 33, 51, 57, 66, 85

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

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

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

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

We now formally state the bubble sort algorithm.

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

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

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

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:

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

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:

$$DATA[BEG], DATA[BEG+1], \ldots, DATA[MID-1]$$

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:

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

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

$$\text{END} < \text{BEG}$$

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.