Arrays, Records and Pointers

4.1 INTRODUCTION

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

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

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

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

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

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

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

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

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

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

4.2 LINEAR ARRAYS

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.3 REPRESENTATION OF LINEAR ARRAYS IN MEMORY

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

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

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

Base(LA)

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

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

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

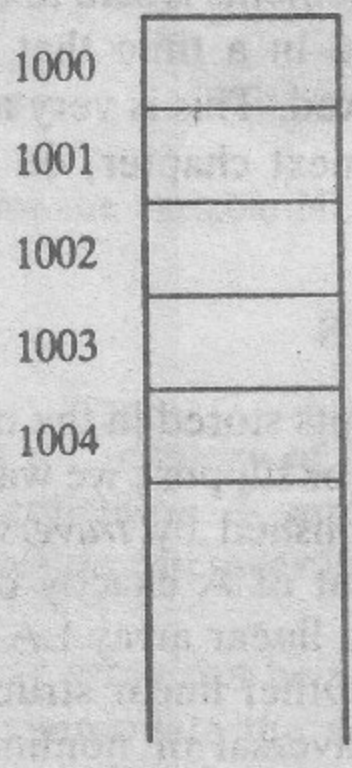


Fig. 4-2 Computer memory.

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

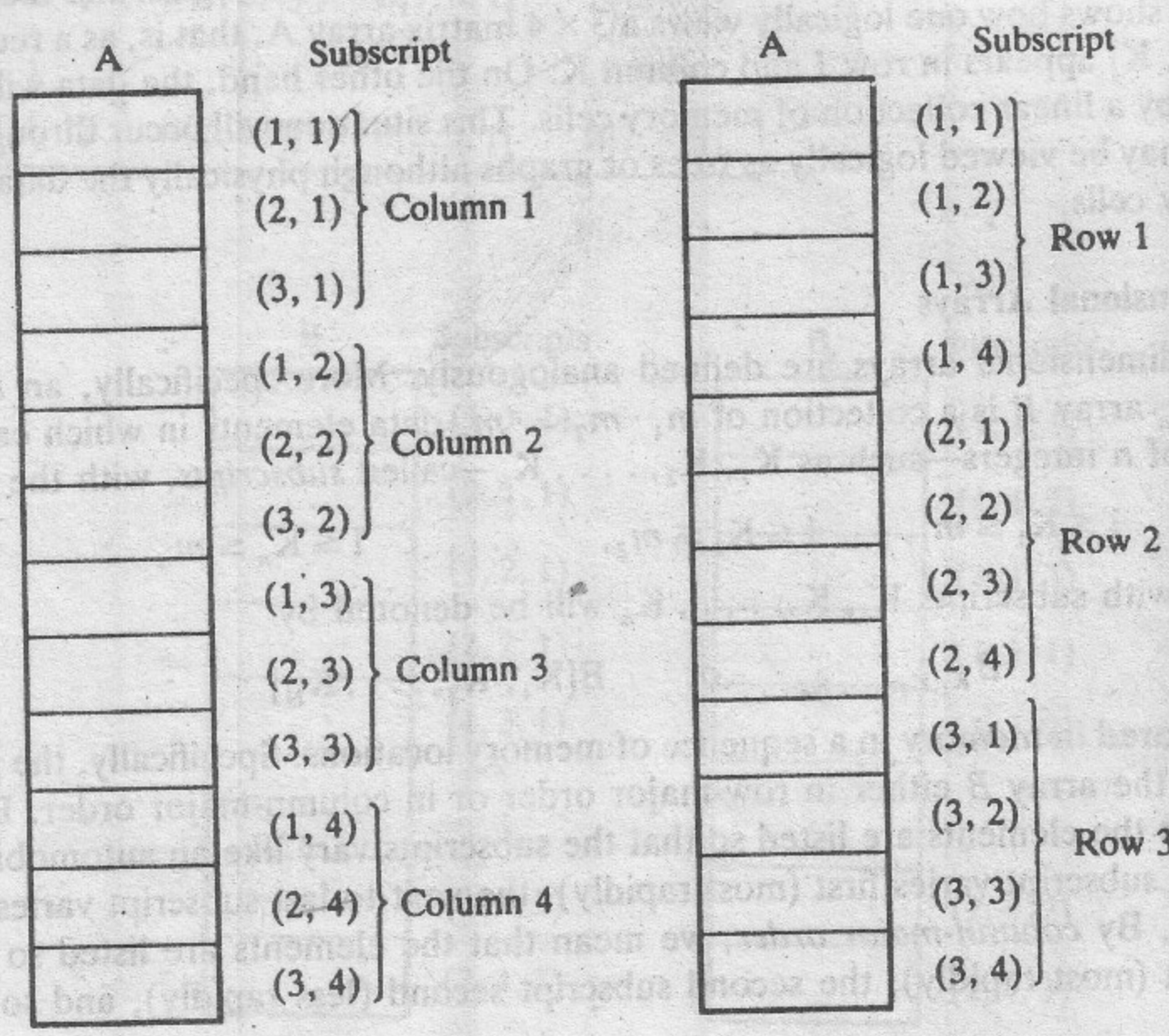
Sorting may seem to be a trivial task. Actually, sorting efficiently may be quite complicated. In fact, there are many, many different sorting algorithms; some of these algorithms are discussed in Chap. 9. Here we present and discuss a very simple sorting algorithm known as the *bubble sort*.

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

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

Representation of Two-Dimensional Arrays in Memory

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



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

Fig. 4-10

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

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

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

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

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

or the formula

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