# 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.

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), ..., A(N)

or by the bracket notation (used in Pascal)

A[1], A[2], A[3], ..., 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).

DATA

| | |
|---|---|
| 1 | 247 |
| 2 | 56 |
| 3 | 429 |
| 4 | 135 |
| 5 | 87 |
| 6 | 156 |

(a)

DATA

| 247 | 56 | 429 | 135 | 87 | 156 |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |

(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

$$\text{LOC(LA[K])} = \text{address of the element LA[K] of the array LA}$$

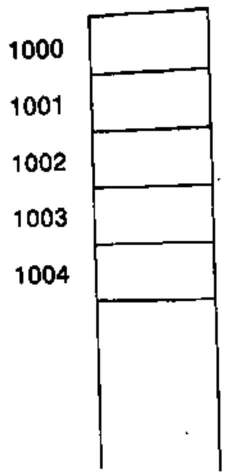



**Fig. 4.2** *Computer Memory*

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*(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.

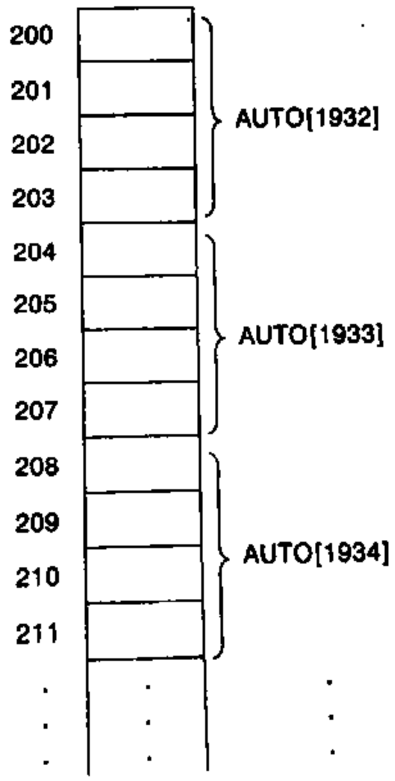



Fig. 4.3

*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.

| | (a) NAME | (b) NAME | (c) NAME | (d) NAME |
|---|---|---|---|---|
| 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 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≤ 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 ≥ 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,

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

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 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[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[l] < 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 $n$th position or "sinks" to the $n$th 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[l] with A[2] and arrange them so that A[l] < 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

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

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.

| | NAME (a) | NAME (b) | NAME (c) |
|---|---|---|---|
| 1 | Mary | Mary | Mary |
| 2 | Jane | Jane | Jane |
| 3 | Diane | Diane | Diane |
| 4 | Susan | Susan | Susan |
| 5 | Karen | Karen | Karen |
| 6 | Edith | Edith | Edith |
| 7 | | Paula | Susan |
| 8 | | | |

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

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

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, $\boxed{55,}$ 60, 66, 77, 80, 88, 99

(2) 11, 22, $\boxed{30,}$ 33, 40, 44, 55, 60, 66, 77, 80, 88, 99

(3) 11, 22, 30, 33, $\boxed{40,}$ 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, $\boxed{55,}$ 60, 66, 77, 80, 88, 99

(2) 11, 22, 30, 33, 40, 44, 55, 60, 66, $\boxed{77,}$ 80, 88, 99

(3) 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, $\boxed{88,}$ 99

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

**Fig. 4.7** *Binary Search for ITEM = 85*

## Complexity of the Binary Search Algorithm

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

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

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

**Example 4.10**

Suppose DATA contains 1 000 000 elements. Observe that

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

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

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

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.

$$\begin{array}{cc} & \overbrace{\begin{array}{cccc} \quad 1 \quad & \quad 2 \quad & \quad 3 \quad & \quad 4 \quad \end{array}}^{\text{Columns}} \\ \text{Rows} \quad \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} A[1,1] & A[1,2] & A[1,3] & A[1,4] \\ A[2,1] & A[2,2] & A[2,3] & A[2,4] \\ A[3,1] & A[3,2] & A[3,3] & A[3,4] \end{bmatrix} \end{array}$$

**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 $4 \times 8$ 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

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

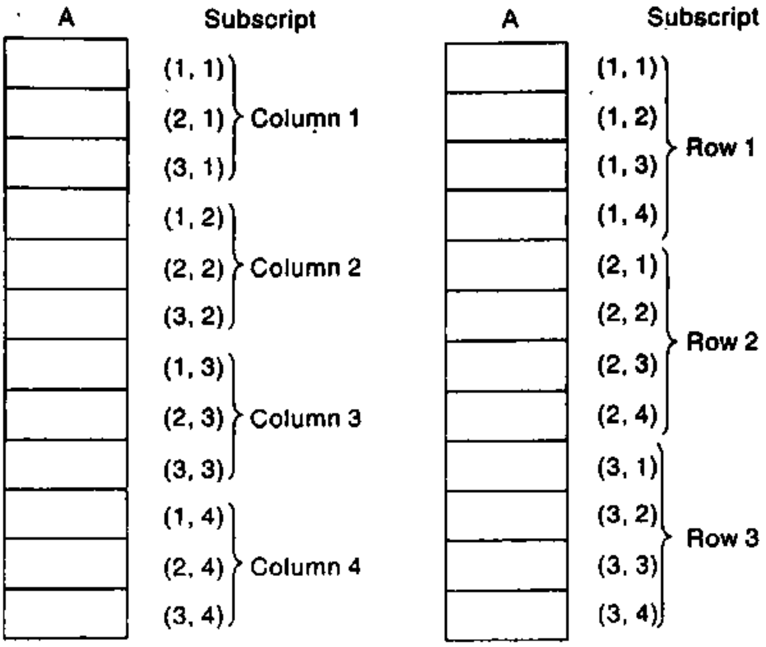
By Eq. (4.3), the length of the first dimension is equal to $5 - 2 + 1 = 4$, and the length of the second dimension is equal to $1 - (-3) + 1 = 5$. Thus NUMB contains $4 \cdot 5 = 20$ elements.

## 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 $3 \times 4$ 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

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




(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

(Column-major order) LOC(A[J, K]) = *Base*(A) + $w$[M(K − 1) + (J − 1)] (4.4)

or the formula

(Row-major order) LOC(A[J, K]) = *Base*(A) + $w$[N(J − 1) + (K − 1)] (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:

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 $3 \times 4$ 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 $2 \times 4$ 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

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

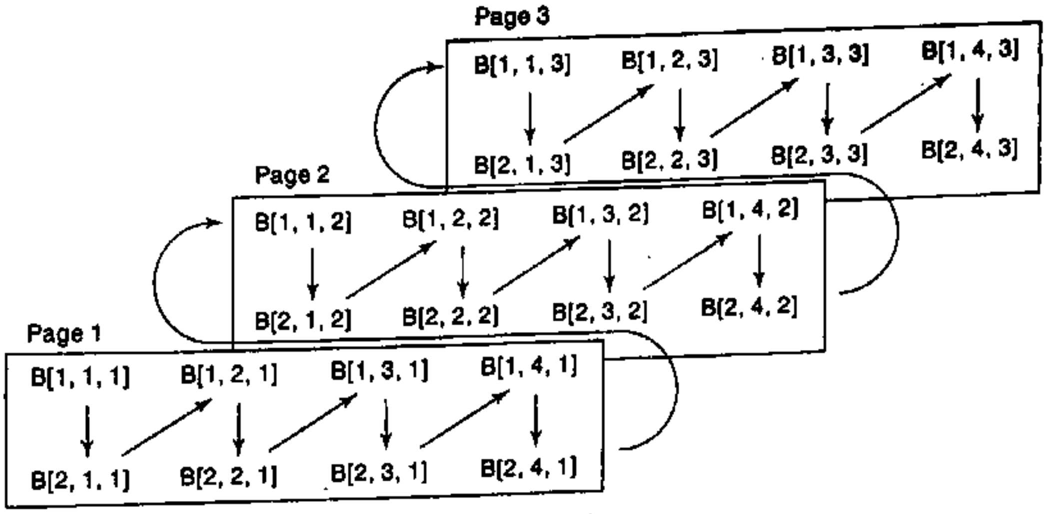
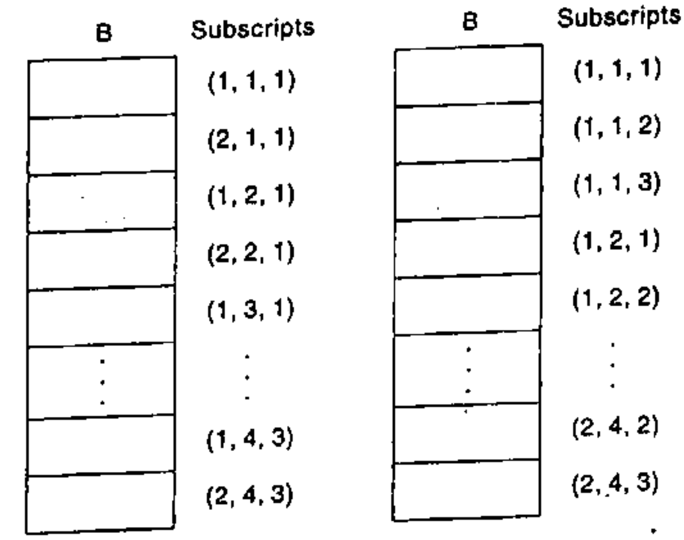



**Fig. 4.11**




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

**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

such a listing. Observe that there are 21 people and the groups contain 4, 9, 2 and 6 people, respectively.

Suppose the membership list is to be stored in memory keeping track of the different groups. One way to do this is to use a two-dimensional $4 \times n$ array where each row contains a group, or to use a two-dimensional $n \times 4$ array where each column contains a group. Although this data structure does allow us to access each individual group, much space will be wasted when the groups vary greatly in size. Specifically, the data in Fig. 4.13 will require at least a 36-element $4 \times 9$ or $9 \times 4$ array to store the 21 names, which is almost twice the space that is necessary. Figure 4.14 shows the representation of the $4 \times 9$ array; the asterisks denote data elements and the zeros denote unused storage locations. (Arrays whose rows—or columns—begin with different numbers of data elements and end with unused storage locations are said to be *jagged*.)

$$\begin{pmatrix} * & * & * & * & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & * & * & * & * & * \\ * & * & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & * & * & 0 & 0 & 0 \end{pmatrix}$$

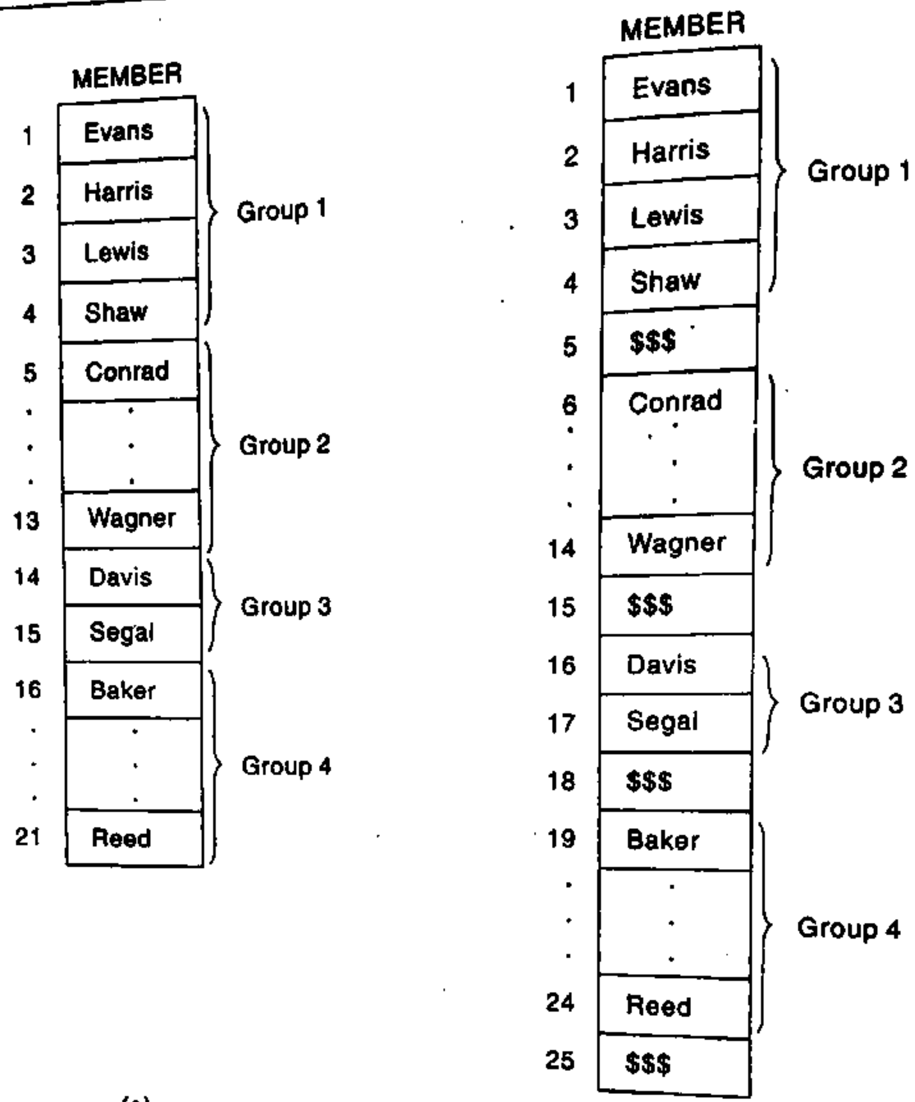
**Fig. 4.14** *Jagged Array*

Another way the membership list can be stored in memory is pictured in Fig. 4.15(a). That is, the list is placed in a linear array, one group after another. Clearly, this method is space-efficient. Also, the entire list can easily be processed—one can easily print all the names on the list, for example. On the other hand, there is no way to access any particular group; e.g., there is no way to find and print only the names in the third group.

A modified version of the above method is pictured in Fig. 4.15(b). That is, the names are listed in a linear array, group by group, except now some sentinel or marker, such as the three dollar signs used here, will indicate the end of a group. This method uses only a few extra memory cells—one for each group—but now one can access any particular group. For example, a programmer can now find and print those names in the third group by locating those names which appear after the second sentinel and before the third sentinel. The main drawback of this representation is that the list still must be traversed from the beginning in order to recognize the third group. In other words, the different groups are not indexed with this representation.

## Pointer Arrays

The two space-efficient data structures in Fig. 4.15 can be easily modified so that the individual groups can be indexed. This is accomplished by using a pointer array (here, GROUP) which contains the locations of the different groups or, more specifically, the locations of the first elements in the different groups. Figure 4.16 shows how Fig. 4.15(a) is modified. Observe that GROUP[L] and GROUP[L + 1] – 1 contain, respectively, the first and last elements in group L. (Observe that GROUP[5] points to the sentinel of the list and that GROUP[5] – 1 gives us the location of the last element in Group 4.)




(a) (b)

**Fig. 4.15**

## Example 4.15

Suppose one wants to print only the names in the Lth group in Fig. 4.16, where the value of L is part of the input. Since GROUP[L] and GROUP[L + 1] − 1 contain, respectively, the locations of the first and last name in the Lth group, the following module accomplishes our task:

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

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

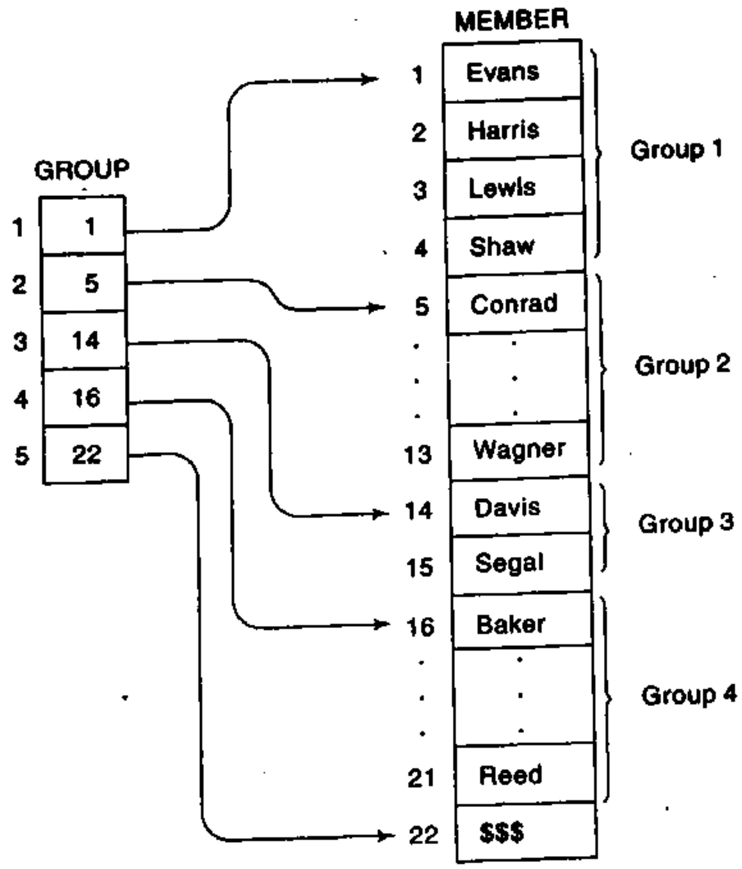



**Fig. 4.16**

A slight variation of the data structure in Fig. 4.16 is pictured in Fig. 4.17, where unused memory cells are indicated by the shading. Observe that now there are some empty cells between the groups. Accordingly, a new element may be inserted in a group without necessarily moving the elements in any other group. Using this data structure, one requires an array NUMB which gives the number of elements in each group. Observe that GROUP[K + 1] – GROUP[K] is the total amount of space available for Group K; hence

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

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

## Example 4.16

Suppose, again, one wants to print only the names in the Lth group, where L is part of the input, but now the groups are stored as in Fig. 4.17. Observe that

GROUP[L] and GROUP[L] + NUMB[L] - 1

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

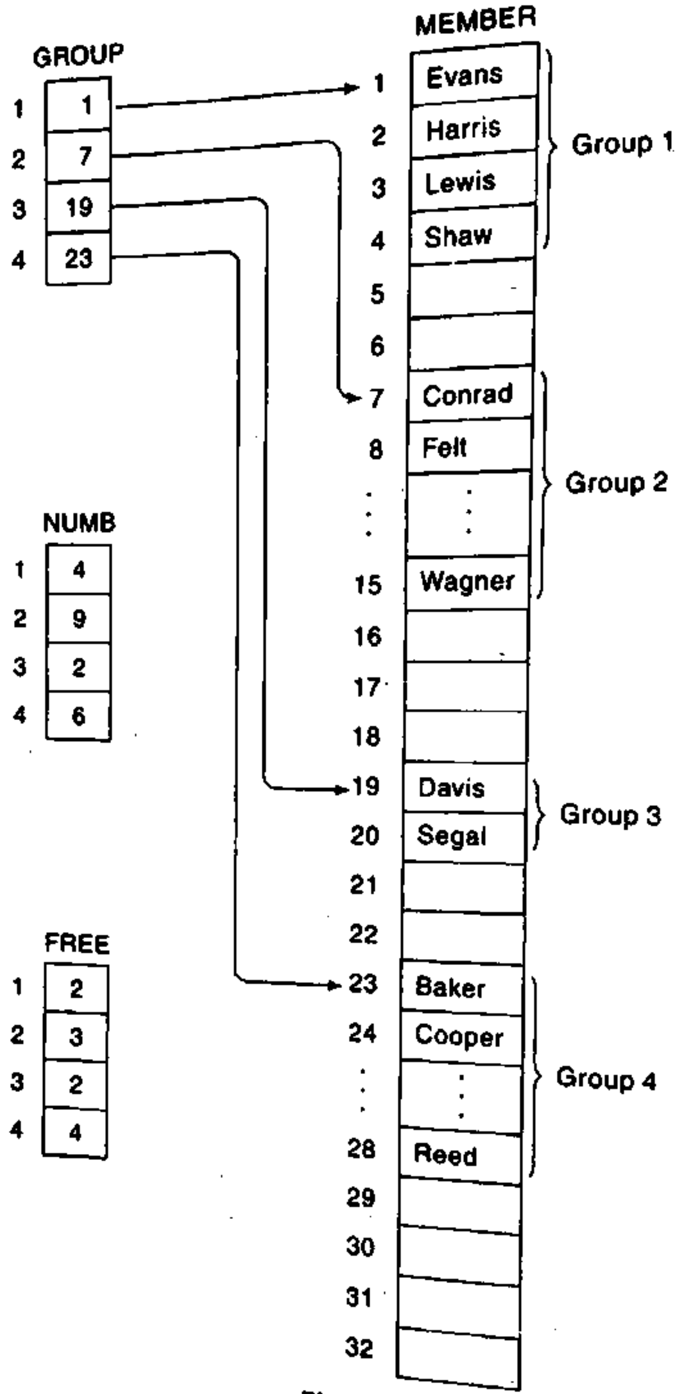



**Fig. 4.17**

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

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

## 4.11 RECORDS; RECORD STRUCTURES

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

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

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

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

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

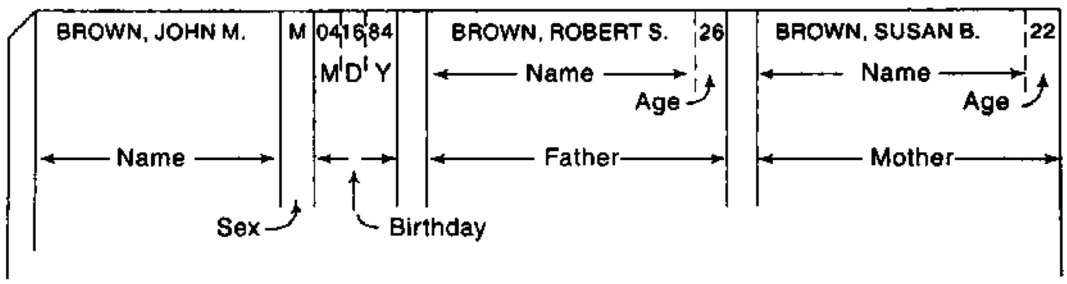



**Fig. 4.18**

### Example 4.17

Suppose a hospital keeps a record on each newborn baby which contains the following data items: Name, Sex, Birthday, Father, Mother. Suppose further that Birthday is a group item with subitems Month, Day and Year, and Father and Mother are group items, each with subitems Name and Age. Figure 4.18 shows how such a record could appear.

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

```
1 Newborn
    2 Name
    2 Sex
    2 Birthday
        3 Month
        3 Day
        3 Year
```

2 Father
  3 Name
  3 Age
2 Mother
  3 Name
  3 Age

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

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

1 Newborn(20)

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

$\text{Newborn}_1$, $\text{Newborn}_2$, $\text{Newborn}_3$,...

or Newborn[1], Newborn[2], Newborn[3],...

to denote different records in the file.

**Example 4.18**

A class of student records may be organized as follows:

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

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

## Indexing Items in a Record

Suppose we want to access some data item in a record. In some cases, we cannot simply write the data name of the item since the same name may appear in different places in the record. For example, Age appears in two places in the record in Example 4.17. Accordingly, in order to specify a particular item, we may have to *qualify* the name by using appropriate group item names in the structure. This *qualification* is indicated by using decimal points (periods) to separate group items from subitems.

**Example 4.19**

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

Newborn.Father.Age     or simply     Father.Age

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

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

1 Newborn(20)

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

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

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

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

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

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

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

Test[3, 1]

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

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

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

and

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

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

## 4.12 REPRESENTATION OF RECORDS IN MEMORY; PARALLEL ARRAYS

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

**Example 4.20**

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

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

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

FATHER.AGE    or    MOTHER. AGE

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

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

**Example 4.21**

Suppose a membership list contains the name, age, sex and telephone number of each member. One can store the file in four parallel arrays, NAME, AGE, SEX and PHONE, as pictured in Fig. 4.19; that is, for a given subscript K, the elements NAME[K], AGE[K], SEX[K] and PHONE[K] belong to the same record.

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

Fig. 4.19

**Example 4.22**

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

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

one array for each elementary data item. Here we must use different variable names for the name and age of the father and mother, which was not necessary in the previous example. Again, we assume that the arrays are parallel, i.e., that for a fixed subscript K, the elements

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

belong to the same record.

## Records with Variable Lengths

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

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

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

The above is an example of a variable-length record. since the data element Siblings can contain zero or more names. One way of storing the file in arrays is pictured in Fig. 4.20, where there are linear arrays NAME, PHONE, FATHER and MOTHER taking care of the first four data items in the records, and arrays NUMB and PTR giving, respectively, the number and location of siblings in an array SIBLING.

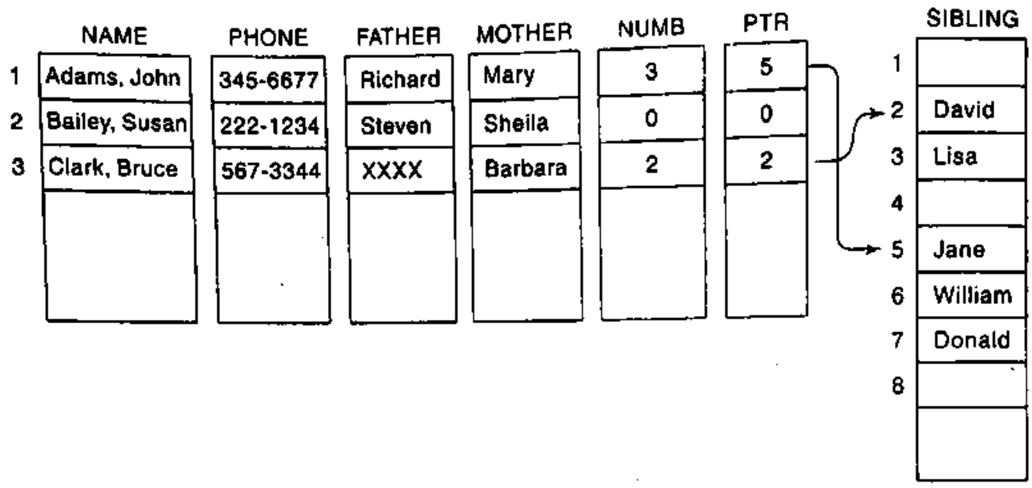



Fig. 4.20

## 4.13 MATRICES

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

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

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

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

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

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

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

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

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

## Algebra of Matrices

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

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

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

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

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

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

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

**Example 4.23**

**(a)** Suppose

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

Then:

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

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

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

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

$$U \cdot W = 1.3 + (-3) \cdot (-5) + 4 \cdot 2 + 5 \cdot (-1) = 3 + 15 + 8 - 5 = 21$$

**(c)** Suppose

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

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

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

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

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

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

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

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

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

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

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

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

**Example 4.24**

Suppose $A$ and $B$ are $2 \times 2$ matrices. We have:

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

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

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

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

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

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

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

## 4.14 SPARSE MATRICES

Matrices with a relatively high proportion of zero entries are called *sparse matrices.* Two general types of $n$-square sparse matrices, which occur in various applications, are pictured in Fig. 4.21. (It is sometimes customary to omit blocks of zeros in a matrix as in Fig. 4.21.) The first matrix, where all entries above the main diagonal are zero or, equivalently, where nonzero entries can only occur on or below the main diagonal, is called a *(lower) triangular matrix.* The second matrix, where nonzero entries can only occur on the diagonal or on elements immediately above or below the diagonal, is called a *tridiagonal matrix.*

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

(a) Triangular matrix

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

(b) Tridiagonal matrix

Fig. 4.21

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

### Example 4.25

Suppose we want to place in memory the triangular array $A$ in Fig. 4.22. Clearly it would be wasteful to store those entries above the main diagonal of $A$, since we know they are all zero; hence we store only the other entries of $A$ in a linear array $B$ as indicated by the arrows. That is, we let

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

Observe first that $B$ will contain only

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

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

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

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

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

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

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

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

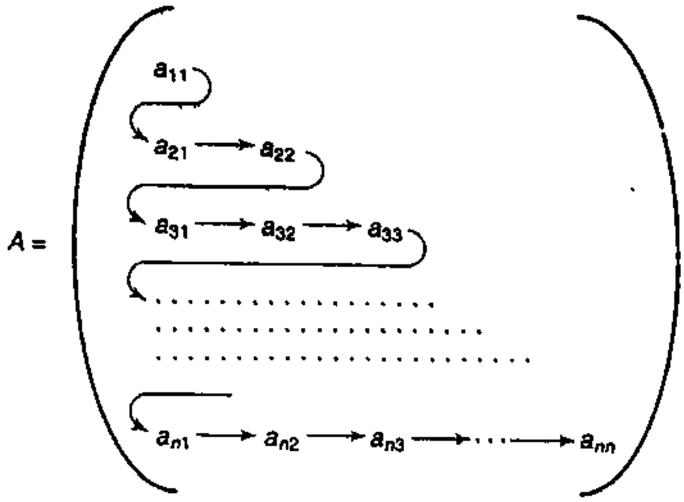


Fig. 4.22

## Solved Problems

# Linear Arrays

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

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

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

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

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

137

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

**(b)** Use the formula

$$\text{LOC(AAA[K])} = Base(\text{AAA}) + w(\text{K} - \text{LB})$$

Hence:
$$\text{LOC(AAA[15])} = 300 + 4(15 - 5) = 340$$
$$\text{LOC(AAA[35])} = 300 + 4(35 - 5) = 420$$

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

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

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

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

Each module traverses the array.

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

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

(c) We want the number of employees born in 1934 or earlier.
  1. Set N50 := 0.
  2. Repeat for K = 1920 to 1934:
    Set N50 := N50 + YEAR[K].
  [End of loop.]
  3. Return.

(d) We want the number of employees born in year 1984 – L or earlier.
  1. Set NL := 0 and LLL := 1984 – L
  2. Repeat for K = 1920 to LLL:
    Set NL := NL + YEAR[K].
  [End of loop.]
  3. Return.

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

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

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

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

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

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

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

**4.4** Consider the alphabetized linear array NAME in Fig. 4.23.

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

**Fig. 4.23**

(a) Find the number of elements that must be moved if Brown, Johnson and Peters are inserted into NAME at three different times.
(b) How many elements are moved if the three names are inserted at the same time?
(c) How does the telephone company handle insertions in a telephone directory?

(a) Inserting Brown requires 13 elements to be moved, inserting Johnson requires 7 elements to be moved and inserting Peters requires 4 elements to be moved. Hence 24 elements are moved.
(b) If the elements are inserted at the same time, then 13 elements need be moved, each only once (with the obvious algorithm).
(c) The telephone company keeps a running list of new numbers and then updates the telephone directory once a year.

## Searching, Sorting

**4.5** Consider the alphabetized linear array NAME in Fig. 4.23.

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

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

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

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

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

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

**4.6** Suppose the binary search algorithm, Algorithm 4.6, is applied to the array NAME in Fig. 4.23 to find the location of Goodman. Find the ends BEG and END and the middle MID for the test segment in each step of the algorithm.

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

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

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

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

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

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

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

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

Steps 1 through 4. Same as in Algorithm 4.6.
5. If ITEM < DATA[MID), then:
    Set LOC := MID.
  Else:
    Set LOC := MID + 1.
  [End of If structure.]
6. Insert ITEM into DATA[LOC] using Algorithm 4.2.
7. Exit.

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

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

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

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

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

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

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

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

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

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

Pass 1. $\boxed{\text{P E}}$O P L E, E$\boxed{\text{P O}}$PLE, E O P P L E
E O P$\boxed{\text{P L}}$E E O P L$\boxed{\text{P E}}$ E O P L E P

Pass 2. E O P L E P, E O P L E P, E O$\boxed{\text{P L}}$E P
E O L$\boxed{\text{P E}}$P, E O L E P P

Pass 3. E O L E P P, E$\boxed{\text{O L}}$E P P, E L$\boxed{\text{O E}}$P P
E L E O P P

Pass 4. E L E O P P, E$\boxed{\text{L E}}$O P P, E E L O P P

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

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

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

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

Writing the sum $S$ forward and backward, we obtain:

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

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

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

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

## Multidimensional Arrays; Matrices

**4.11** Suppose multidimensional arrays A and B are declared using

A(–2:2, 2:22) and B(1:8, –5:5, –10:5)

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

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

Length = upper bound – lower bound + 1

Hence the lengths $L_i$ of the dimensions of A are:

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

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

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

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

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

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

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

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

Therefore, LOC(B[3, 3, 3]) = 400 + 4(1210) = 400 + 4840 = 5240

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

**(a)** Finds the number NUM of nonzero elements in A
**(b)** Finds the SUM of the elements above the diagonal, i.e., elements A[I, J] where I < J
**(c)** Finds the product PROD of the diagonal elements $(a_{11}, a_{22}, \ldots, a_{nn})$

```
(a) 1. Set NUM := 0.
    2. Repeat for I = 1 to N:
    3.      Repeat for J = 1 to N:
                 If A[I, J] ≠ 0, then: Set NUM := NUM + 1.
            [End of inner loop.]
       [End of outer loop.]
    4. Return.
(b) 1. Set SUM := 0.
    2. Repeat for J = 2 to N:                                   143
    3.      Repeat for I = 1 to J - 1:
                 Set SUM := SUM + A[I, J].
            [End of inner Step 3 loop.]
    4. Return.
(c) 1. Set PROD := 1. [This is analogous to setting SUM = 0.]
    2. Repeat for K = 1 to N:
          Set PROD := PROD*A[K, K].
       [End of loop.]
    3. Return.
```

**4.13** Consider an $n$-square tridiagonal array A as shown in Fig. 4.24. Note that A has $n$ elements on the diagonal and $n - 1$ elements above and $n - 1$ elements below the diagonal. Hence A contains at most $3n - 2$ nonzero elements. Suppose we want to store A in a linear array B as indicated by the arrows in Fig. 4.24; i.e.,

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

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

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

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

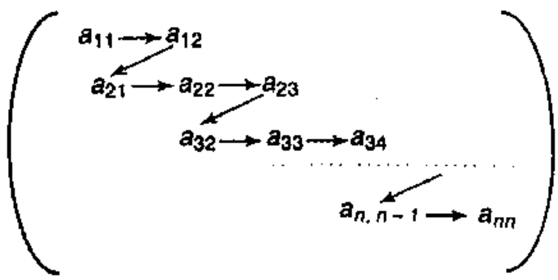



**Fig. 4.24** *Tridiagonal Array*

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

Hence

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

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

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

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

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

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

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

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

**(c)** First note that, for a symmetric matrix, we need store only either those elements on or below the diagonal or those on or above the diagonal. Therefore, A and B can be stored in an $n \times (n + 1)$ array $C$ as pictured in Fig. 4.25, where C[J, K] = A[J, K] when J ≥ K but C[J, K] = B[J, K –1] when J < K.

$$\begin{pmatrix} a_{11} & b_{11} & b_{12} & b_{13} & \cdots & b_{1,n-1} & b_{1n} \\ a_{21} & a_{22} & b_{22} & b_{23} & \cdots & b_{2,n-1} & b_{2n} \\ a_{31} & a_{32} & a_{33} & b_{33} & \cdots & b_{3,n-1} & b_{3n} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & a_{n3} & a_{n4} & \cdots & a_{nn} & b_{nn} \end{pmatrix}$$

**Fig. 4.25**

## Pointer Arrays; Record Structures

**4.15** Three lawyers, Davis, Levine and Nelson, share the same office. Each lawyer has his own clients. Figure 4.26 shows three ways of organizing the data.

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

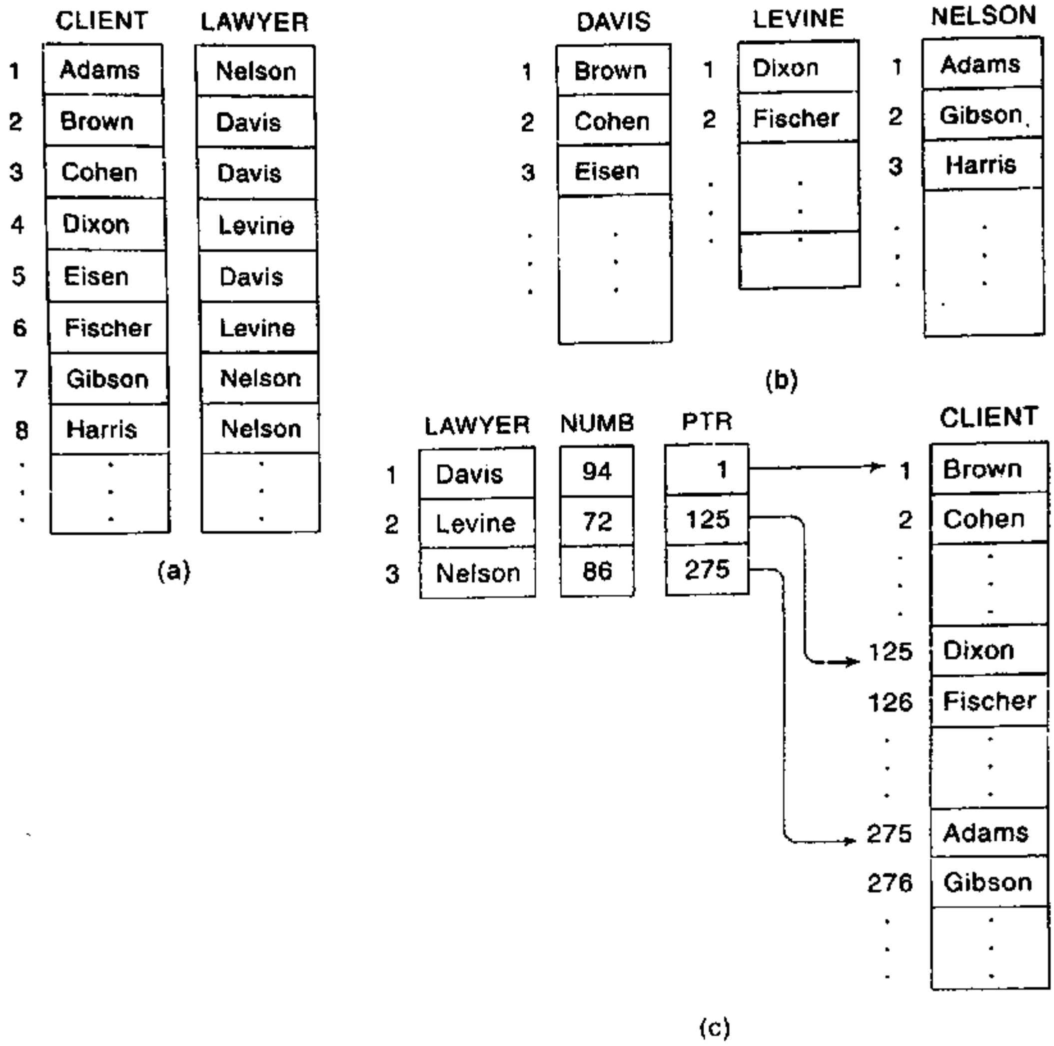



**Fig. 4.26**

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

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

Which data structure is most useful? Why?

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

Suppose there are only one secretary and one telephone number, and suppose there is a single monthly billing of the clients. Also, suppose clients frequently change from one lawyer to another. Then Fig. 4.26(a) would probably be the most useful data structure.

Suppose the lawyers operate completely independently: each lawyer has his own secretary and his own telephone number and bills his clients differently. Then Fig. 4.26(b) would likely be the most useful data structure.

Suppose the office processes all the clients frequently and each lawyer has to process his own clients frequently. Then Fig. 4.26(c) would likely be the most useful data structure.

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

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

**(a)** Draw the corresponding hierarchical structure.
**(b)** Which of the items are elementary items?

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

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

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

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

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

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

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

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

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

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

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

**(a)** How many elementary items are there in the file?
**(b)** How does one access (i) the major of the eighth student and (ii) the sophomore GPA of the forty-fifth student?
**(c)** Find each output:
- **(i)** Write: Name[15]
- **(ii)** Write: CUM
- **(iii)** Write: GPA[2].
- **(iv)** Write: GPA[1, 3].

**(a)** Since GPA is counted 4 times per student, there are 11 elementary items per student, so there are altogether 2200 elementary items.
**(b)** (i) Student.Major[8] or simply MAJOR[8]. (ii) GPA[45, 2].
**(c)** **(i)** Here Name[15] refers to the name of the fifteenth student. But Name is a group item. Hence LAST[15], First[15] and MI[15] are printed.
**(ii)** Here CUM refers to all the CUM values. That is,

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

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

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

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

**4.19** An automobile dealership keeps track of the serial number and price of each of its automobiles in arrays AUTO and PRICE, respectively. In addition, it uses the data structure in Fig. 4.27, which combines a record structure with pointer variables. The new Chevys, new Buicks, new Oldsmobiles, and used cars are listed together in AUTO. The

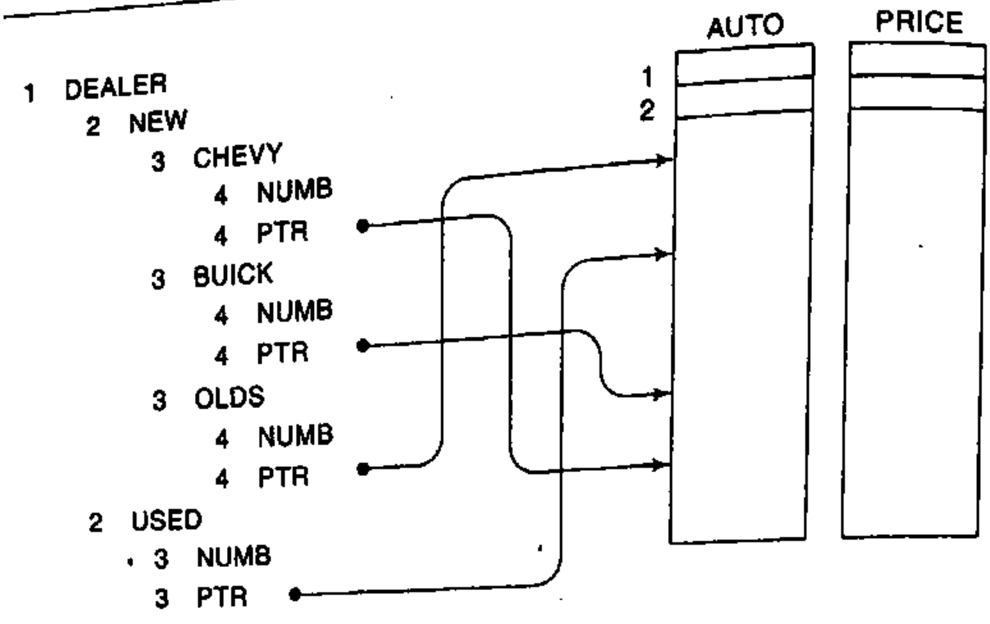



Fig. 4.27

variables NUMB and PTR under USED give, respectively, the number and location of the list of used automobiles.

**(a)** How does one index the location of the list of new Buicks in AUTO?
**(b)** Write a procedure to print serial numbers of all new Buicks under \$10 000.

**(a)** Since PTR appears more than once in the record structure, one must use BUICK.PTR to reference the location of the list of new Buicks in AUTO.
**(b)** One must traverse the list of new Buicks but print out only those Buicks whose price is less than \$10 000. The procedure follows:

**Procedure P4.19:** The data are stored in the structure in Fig. 4.27. This procedure outputs those new Buicks whose price is less than \$10 000.

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

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

This can be accomplished as in Fig. 4.28. That is, besides AUTO and PRICE, there is an array POINTER such that POINTER[K] gives the location in an array ACCESSORIES of the list of accessories (with sentinel '\$\$\$') of AUTO[K].

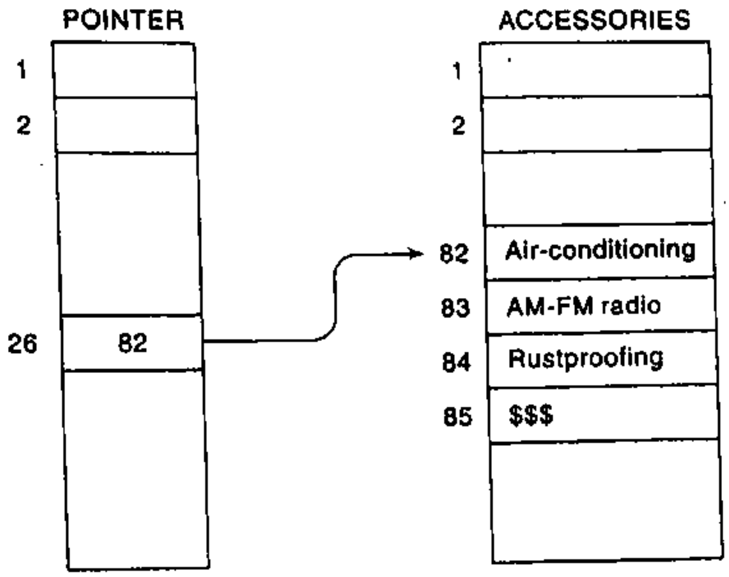



Fig. 4.28

## SUPPLEMENTARY PROBLEMS

### Arrays

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

**4.2** Consider the following multidimensional arrays:

X(–5:5, 3:33)    Y(3:10, 1:15, 10:20)

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

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

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

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

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

**4.4** Suppose A is a linear array with $n$ numeric values. Write a procedure

MEAN(A, N, AVE)

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

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

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

(a) Finds the average grade for each test
(b) Finds the final grade for each student where the final grade is the average of the student's five highest test scores
(c) Finds the number NUM of students who have failed, i.e. whose final grade is less than 60
(d) Finds the average of the final grades

## Pointer Arrays; Record Structures

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

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

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

(a) Draw the corresponding hierarchical structure.
(b) Which of the items are elementary items?
(c) Describe a record structure—for example, a PL/1 structure or a Pascal record—to store the data.

**4.8** Consider the data structure in Fig. 4.27. Write a procedure to carry out each of the following:

(a) Finding the number of new Oldsmobiles selling for under \$10 000.
(b) Finding the number of new automobiles selling for under \$10 000.
(c) Finding the number of automobiles selling for under \$10 000.
(d) Listing all automobiles selling for under \$10 000.

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

**4.9** A class of student records is organized as follows:

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

(a) How many elementary items are there?
(b) Describe a record structure—for example, a PL/1 structure or a Pascal record, to store the data.
(c) Describe the output of each of the following Write statements: (i) Write: Final[15], (ii) Write: Name[15] and (iii) Write: Test[4].

**4.10** Consider the data structure in Solved Problem 4.18. Write a procedure which

(a) Finds the average of the sophomore GPA scores
(b) Finds the number of biology majors
(c) Finds the number of CUM scores exceeding K

## PROGRAMMING PROBLEMS

### Arrays

Assume that the data in Table 4.1 are stored in linear arrays SSN, LAST, GIVEN, CUM and YEAR (with space for 25 students) and that a variable NUM is defined which contains the actual number of students.

**4.1** Write a program for each of the following:

(a) Listing all students whose CUM is K or higher. (Test the program using K = 3.00.)
(b) Listing all students in year L. (Test the program using L = 2, or sophomore.)

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

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

**Table 4.1**

| *Social Security Number* | *Last Name* | *Given Name* | *CUM* | *Year* |
|---|---|---|---|---|
| 211-58-1329 | Adams | Bruce | 2.55 | 2 |
| 169-38-4248 | Bailey | Irene L. | 3.25 | 4 |
| 166-48-5842 | Cheng | Kim | 3.40 | 1 |
| 187-52-4076 | Davis | John C. | 2.85 | 2 |
| 126-63-6382 | Edwards | Steven | 1.75 | 3 |
| 135-58-9565 | Fox | Kenneth | 2.80 | 2 |

(*Contd.*)

*(Contd.)*

| | | | | |
|---|---|---|---|---|
| | | Gerald S. | 2.35 | 2 |
| 172-48-1849 | Green | Gary | 2.70 | 2 |
| 192-60-3157 | Hopkins | Deborah M. | 3.05 | 1 |
| 160-60-1826 | Klein | John | 2.60 | 3 |
| 166-52-4147 | Lee | William | 2.30 | 2 |
| 186-58-0430 | Murphy | Ronald P. | 3.90 | 4 |
| 187-58-1123 | Newman | Paul | 2.05 | 3 |
| 174-58-0732 | Osborn | David | 1.55 | 2 |
| 183-52-3865 | Parker | Mary J. | 1.85 | 1 |
| 135-48-1397 | Rogers | Joanna | 2.95 | 2 |
| 182-52-6712 | Schwab | David E. | 3.15 | 3 |
| 184-48-8539 | Thompson | Adam | 2.50 | 2 |
| 187-48-2377 | White | | | |

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

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

**4.6** Write a program which reads the record of a student

SSNST, LASTST, GVNST, CUMST, YEARST

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

**(a)** 168-48-2255, Quinn, Michael, 2.15, 3
**(b)** 177-58-0772, Jones, Amy, 2.75, 2

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

**4.8** Write a program for each of the following:

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

**(b)** Reading the social security number SOC of a student and using BINAR and the array NUMBER to find and print the student's record. Test the program using (i) 174-58-0732, (ii) 172-55-5554 and (iii) 126-63-6382. (Compare with Programming Problem 4.4.)

## Pointer Arrays

Assume the data in Table 4.2 are stored in a single linear array CLASS (with space for 50 names). Also assume that there are 2 empty cells between the sections, and that there are linear arrays NUMB, PTR and FREE defined so that NUMB[K] contains the number of elements in Section K, PTR[K] gives the location in CLASS of the first name in Section K, and FREE[K] gives the number of empty cells in CLASS following Section K.

**Table 4.2**

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

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

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

**4.11** Write a program which prints the names in columns as they appear in Table 4.2.

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

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

## Miscellaneous

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

**4.15** Translate the matrix multiplication algorithm, Algorithm 4.7, into a subprogram

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

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

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

**4.16** Consider the polynomial

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

Evaluating the polynomial in the obvious way would require

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

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

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

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

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

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

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