3.2.3 Applications of Linear Arrays

Linear arrays are one of the primitive data structures. These are used represent all sorts of lists. In addition, these are used implement other data structures such as *stacks*, *queues*, *heaps*, etc..

3.2.4 Limitations of Linear Arrays

The main limitations of linear arrays are

1. The prior knowledge of number of elements in the linear array is necessary.
2. These are static structures. Static in the sense that whether memory is allocated at compilation time or runtime, their memory used by them cannot be reduced or extended.
3. Since the elements of these arrays are stored in consecutive locations, the insertions and deletions in these arrays are time consuming. This is because of moving down or up to create a space of new element or to occupy the space vacated by the deleted element.

3.3 TWO-DIMENSIONAL ARRAYS

A two-dimensional array is a list of a finite number $m*n$ of homogeneous data elements such that

- The elements of the array are referenced by two index sets consisting of m and n consecutive integer numbers.
- The elements of the array are stored in consecutive memory locations.

The *size* of the two-dimensional array is denoted by $m \times n$ and pronounced as m by n.

Suppose b is the name of the two-dimensional array, then its element in the ith row and jth column is denoted as

b_{ij}	in mathematical notation
$b(i, j)$	in BASIC and FORTRAN languages
$b[i, j]$	in Pascal language
$b[i][j]$	in C/C++ and Java languages

wo-dimensional arrays are called *matrices* in mathematics and *tables* in business pplications. A two-dimensional $m \times n$ arrayais pictured as a rectangular pattern of m row and columns, where the element a_{ij} appears in row i and column j. Figure 3.6 pictures two-imensional array a of size 3×3.

	Col #0	Col #1	Col #2
Row #0	a_{00}	a_{01}	a_{02}
Row #1	a_{10}	a_{11}	a_{12}
Row #2	a_{20}	a_{21}	a_{22}

Figure 3.6 Picturing a two-dimensional array a

3.3.1 Representing Two-dimensional Array in Memory

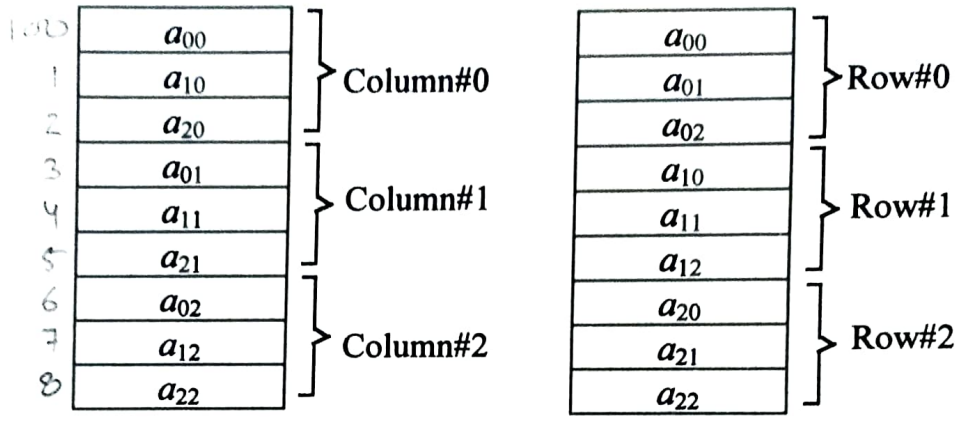
Let a be a two-dimensional $m \times n$ array. Though a is pictured as a rectangular pattern with m row and n columns, it is represented in memory by a block of $m*n$ sequential memory locations. However, the elements can be stored in two different ways —

- ***Column major order*** – the elements are stored column by column i.e. m elements of the first column ~~and~~ are stored in first m locations, elements of the second column are stored in next m locations, and so on.

- ***Row major order*** – the elements are stored row by row i.e. n elements of the first row and stored in first n locations, elements of the second row are stored in next n locations, and so on.

Figure 3.7 shows these storage schemes for two-dimensional array A of size 3×3.

In Pascal, C/C++ the storage order is column major order, where as in FORTRAN the storage order is row major order.



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

Figure 3.7 Representation of two-dimensional array *a* of size 3×3 in memory

Let us consider a two-dimensional array *a* of size *m*×*n*. Further consider that the lower bound for the row index is *lbr* and for column index is *lbc*.

Like linear array, system keeps track of the address of first element only i.e. the *base address* of the array.

Using this base address, the computer computes the address of the element in the *i*th row and *j*th column i.e. *loc*(*a*[*i*][*j*]), using the following formulae:

Column Major Order

$$loc(a[i][j]) = base(a) + w\,[m\,(j - lbc) + (i - lbr)] \qquad \text{in general}$$

$$loc(a[i][j]) = base(a) + w\,(m \times j + i) \qquad \text{in C/C++}$$

Row Major Order

$$loc(a[i][j]) = base(a) + w\,[n\,(i - lbr) + (j - lbc)] \qquad \text{in general}$$

$$loc(a[i][j]) = base(a) + w\,(n \times i + j) \qquad \text{in C/C++}$$

where *w* is the number of bytes per storage location for one element of the array.