CHAPTER 1
ARRAYS AND RECORDS

1.1 INTRODUCTION

Data Structure are divided in two parts as

a) Linear structure or linear list.

b) Non linear list.

Example of linear structure are arrays and link list and of non linear structure are trees and graphs.

Linear Structure - If element of data structure form a sequence. In memory linear structure can be represented by two ways.

1) By having linear relationship between the elements represented by means of sequential memory location e.g. arrays.
2) By having linear relationship between the elements represented by means of pointers or links .eg. linked lists.

1.2 OPERATION THAT CAN BE PERFORM ON LINEAR STRUCTURE

(A) **Traversal** - Processing each element
(B) **Searching** - To find the location of a particulars element.
(C) **Insertion** - Adding a new element.
(D) **Deletion** - Removing an element.
(E) **Sorting** - Arranging the elements in order.
(F) **Merging** - Combining two different lists into one single list.

1.3 LINEAR ARRAYS

It is a group of contiguous or related data item that share a common name. It is basically a collection of same data type in other words.

Array may be defined as a collection of homogeneous elements of a specific data type so that each element of array are stored respectively in successive memory locations. Length or size of array is the total number n of elements.

Length = UB - LB + 1

Here UB the [upper bound] is the largest index and LB the lower bound is the smallest index. if LB = 1 then Length = UB

The elements of an array A may be denoted by A [1], A [2], A [3], ...A [N], the number K in A [K] is called subscript or an index and A [K] is called subscripted variable.

1.4 MEMORY REPRESENTATION OF LINEAR ARRAYS

Let we have a linear array A and A [n] be the n^{th} element of array then

Loc [A[n]] = address of the element A [n] of array A

Loc [A [n]] = address of n^{th} element of an array

= Base address + size of data type * (n-lower bound)

for example we have an array A having element A_1, A_2, A_3,.....A_n i.e. n element with starting address 5555

A_1	A_2	A_3	A_4	A_5	A_6	……	A_n
5555	5556	5557	5558	5559	5560	……	…….

Let we have to find out address of A_6 the 6^{th} element of array.

here lower bound = 1 as A_1 is the first element of the array

n = 6^{th} element

Address of A_6 = Base address + 1*(6-1)

= 5555 + 1*(6-1) = 5555 + 1*5

= 5560 *as can be seen from figure

Here we assume data type is character so for character size is equal to one.

Q 1.1 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].

Solution:

(a) The number of elements is equal to the length; hence use the formula
Length = UB-LB+1
Accordingly, Length (AAA) = 50-5+1 = 46
Length (BBB) = 10-(-5) + 1 = 16
Length (CCC) = 18-1+1 = 18
Note that Length (CCC) = UB, since LB = 1.

(b) Use the formula
Hence; LOC (AAA [K]) = Base (AAA) + w (n-LB)
LOC (AAA [15]) = 300 +4(15-5) =340
LOC (AAA [35]) = 300 + 4(35-5) = 420
AAA [55] is not an element of AAA. Since 55 exceeds UB = 50.

1.5 TRAVERSING LINEAR ARRAYS

Let A be an array containing data elements (A_1, A_2,.....A_n) stored in memory of computer now traversing A means accessing and processing (visiting) each element of A exactly once

Algorithm To Traverse the given Array

```
Algorithm_Traverse ( )
                {
                        i = LB    /* set i equal to lower bound */
                   while(i<=UB)
                   {
                    write (A[i]);
                   }
                    i = i +1          /* increment i */
                }
```

Explanation:

Let we have an array given below

3	6	9	12	18
1	2	3	4	5

1) here LB = lower bound = 1
So i = LB = 1 and UB = upper bound = 5
as i < UB i.e. 1 < 5
We enter while loop
Write (A[i])
that is A [1] = >3
so process **3**
Output 3
2) i = i + 1 ,i will be now
i = 1 + 1 = 2
as i < UB 2 < 5
Write A [2] = >**6**
Process 6
Output 6

Like wise this we process each element of the array and at the last processing I becomes 5
A[i] => A[5]=>**18**
i = i +1 = 5+1= 6
as i UB as 6 5 so we stop processing
so output will be
3, 6, 9, 12, 18

1.6 INSERTION AND DELETION

Let A be an array insertion means adding a new element at any position to an existing array A and deletion is removing an element from array A.

Algorithm To Insert element into the given Array

```
Algorithm_Insert ( )
    {
        i = n
            while (i ≥ K)
        {
            A [i + 1] = A [i];
            i = i -1;
        }
            A [K] = item
            n = n+1
    }
```

Explanation :

Let we have an array A [ ] =

3	6	9	12	18	21
1	2	3	4	5	6

and we have to add item with value 15 at position fifth (K=5)
as above array contain Six elements i.e. its size is equal to six n = 6
So by statement
i = n
i will become six i = 6 ;
for i = 6 as (i ≥ K) i.e. (6 ≥ 5) as the condition is true
we enter the while loop
now by the statement
A [i+1] = A [i]
That is A [7] = A [6]
So sixth positioned element is assigned to seventh position
and i = i-1 means i will be decremented by one that is iI becomes i = 5
for i =5 and as (i ≥ K) i.e. (5 ≥ 5) again the condition is true
we again enter the while loop
and A [i + 1] = A [i]
A [6] = A [5]
fifth positioned element is assigned to sixth position

i = i-1= > i becomes , i = 4
as i not equal and greater than K i.e. 4 ≥ 5 condition is false So we do not enter while loop
by above two statements
A [7] = A [6]
and A [6] = A [5]
the sixth position element has been shifted to seventh position element and fifth position element has been shifted to sixth position element so array becomes

3	6	9	12		18	21
1	2	3	4	5	6	7

then we have
A [K] = Item here k is 5 so it become
A [5] = Item
as Item is the new element which is to be inserted and in this example its value is 15 so

after this statement value of item is assigned at fifth position and array now becomes

3	6	9	12	15	18	21

and n = n+1 = 6+1 = seven as after insertion size is increased
Now after adding one element the size become seven

Algorithm To Delete element from the given Array

```
Algorithm_delete ( )
{
    Item = A [K]
        for (i = K, i < = n-1, i + +)
    {
        A [i] = A [i + 1]
    }
        n = n-1
}
```

Explanation :

Let we have an array shown below.

3	6	9	12	15	18
1	2	3	4	5	6

here size n =6 Let we have to delete the item 9
at position K =3
Item = A [K] = A [3] Item = 9
for i =3 as i $\leq$ n-1 i.e. 3 $\leq$ 5
A [i] = A [i + 1]=>A [3] = A [4]
so i = i+1 So now i will be one more i = 4
for i = 4 or i $\leq$ n-1 i.e. 4 $\leq$ 5
A [i] = A [i + 1]=>A [4] = A [5]
i = i +1 So i =5
for i =5 as i $\leq$ n-1 i.e. 5 $\leq$ 5
A [5] = A [6]
i = i+1 i =6
by above statement
A [3] = A [4]
A [4] = A [5]
A [5]= A [6]

The fourth position element has been shifted to third position, fifth position element has been shifted to fourth position and sixth position element has been shifted to fifth position respectively.

3	6	12	15	18
1	2	3	4	5

for i =6 as i $\leq$ n - 1 is not posible
6 is not less than or equal to 5
we have n = n -1
n = 6-1 = 5
So size will be one less than present value as one element has been deleted.

1.7 TWO DIMENSIONAL ARRAYS

Two dimensional m x n array A is a collection fo m. n data elements such that each element is specified by a pair of integers (such as i, j) called subscripts, with the property that

$1 \leq i \leq m$ and $1 \leq j \leq n$

The element of A with first subscript i and second subscript j will be denoted by

A [I,J]

Two dimensional arrays are called matrix arrays. In two dimensional m x n array A where the elements of A form a rectangular array with m rows and n columns and where the element. A [I,J] appears in row i and column j (row is horizontal lists of elements, and column is a vertical lists of element here given below figure that show 2 x3 array A having 2 rows and three 3 column.

	1	2	3
1	A [1, 1]	A [1,2]	A [1,3]
2	A [2,1]	A [2,2]	A [2, 3]

Two dimensional 2 x 3 array A

1.8 MEMORY ALLOCATION OF TWO DIMENSIONAL ARRAYS

Two dimensional (2D) arrays can be stored in an array in any of the two ways.

1. ***Row Major.***
2. ***Column Major.***

1. Row Major.

All the rows are stored first i.e. row number 1 then row number 2 and So on

Let we have an array of Size 3 x 4

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

It can be stored in memory by

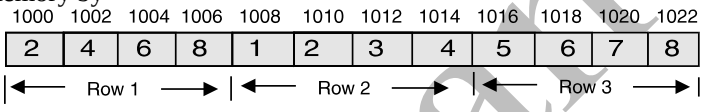

Figure 1.1

Calculation of Memory address for m x n array for i the row and j th column

= base address + [(i-1)*n+(j-1)]*(Size of data type)

the size of above array is 3 x 4

so m = 3 and n =4

now let us find out the address of 2nd row and 2nd column

= 1000 + [(2-1)*4+(2-1)]*2

= 1000 + (4+1)*2

= 1000 + 10 = 1010 which can be seen from the fig. 1.1

2. Column Major:

The elements are stored column wise, that is first column 1 then column 2 and so on the same matrix which took earlier, will be stored in following way in case of column major.

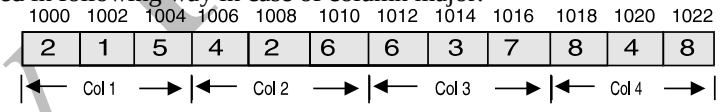

Figure 1.2

In Case of column major

Calculation of Memory address for m x n array for ith row and j th column

= base address + [(i-1) + (j-1)*m]*(Size of data type)

Here also m =3 and n = 4

Let's find out the address of 2nd row and 2nd col.

= 1000 + [(2-1) + (2-1)*3]*(2)

= 1000 + [1+3] * 2

= 1000 + 8= 1008

Which can be seen from the fig.1.2

1.9 MATRICES

Matrices: It is a collection of number which are analogous to two dimensional arrays so An mxn matrix A is an array of m.n numbers arrangned in m rows and n columns as given below:

$$A = \begin{matrix} A_{11}, A_{12}, \ldots A_{1n} \\ A_{21}, A_{22}, \ldots A_{2n} \end{matrix}$$

.....

A_{m1}, A_{m2},....A_{mn}

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)

Square matrix - If a matrix have equal number n of rows and columns it is known as Square Matrix.

Algorithm for matrix multiplication

```
matrix_mul( )
{
    for (i = 1; i < = m ; i + +)
        {
            for (j = 1 ; j < = n; j ++)
                {
                    S [ i,j] = 0 ;
                    for (k = 1 ; k <= P; K ++)
                    {
                    S [i,j] = S[i,j] + A [i,k]*B [k,j]
                    }
                }
        }
}
```

here in above algorithm let A be an M x P matrix array and B be an PxN matrix array. This algorithm computes the product of A and B in an MxN matrix array S.

1.10 SPARSE MATRIX

An array or matrix with relatively higher number of zeros then the elements is known as **sparse matrix.** Let us take the example of two matrices.

0	0	0
1	0	0
0	0	1

Figure 1.3

0	0	0	0	0
0	0	0	0	0
1	0	0	0	0
0	0	3	0	0
1	0	0	6	0

Figures 1.4

In case of fig.1.3 we are utilizing only 2 memory units, rest of the 7 are just wastage of memory spaces. Similarly, the fig.1.4 has 4 data items and remaining 21 are just zeroes and hence wastage of memory as well as time.

To save this space and time, sparse matrices require some sparse representation.

Representation of Sparse Matrix

Sparse matrices are represented using **3 tuple forms**, where we have 3 columns and n + 1 rows (where n are the non-zero entries).The first row in the 3 tuple form is the size of sparse matrix and number of non zero entries. Rest all the entries are row, column, and value of the non-zero entries.

Figure 1.3 will be represented in the following form

3	3	2
2	1	1
3	3	1

in fig.1.3 size is 3 x 3 and non zero entries are 2.

The second row i.e. 2,1,1 corresponds to second row, first column, value 1. Similarly, the third row i.e. 3,3,1 corresponds to third row, third column , Value 1.

It may seen from the above example that no space has been saved in comparison to fig.1.3 But this is not the case for higher number of rows and columns, as we will see from the following representation of fig.1.4.

5	5	4

3	1	1
4	3	3
5	1	1
5	4	6

The above representation use only 5 x 3 = 15 units of space as compared to 25 units in fig. 1.4 hence, the space of 10 units has been saved in this form of representation.

1.11 RECORDS, RECORD STRUCTURES

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 sub items; 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 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 sub item, the data items in a record form a hierarchical structure which can be described by means of "level" numbers.

1.12 SEARCHING

1. Linear Search:

It implies comparing each and every element of a linear list until a particular element is found. If the element is present in the list, the algorithm should return the position of that element, else the algorithm should return NULL.

Algorithm Linear Search

```
Algorithm Linear Search (array a[], n, data)
{
        found=1;
        for (i =1 ;i<= n;i++)
        {
                if (a[i] = data)
                {
                found=1;
                break;
                }

        }
        if(found==1)
          {
            write("Element is present at",i);
          }
      else
          {
            write("Element is not present ");

          }

}
```

2. Binary Search:

The binary search can only be applied to the sorted list of the elements. We will discuss the binary search in case where the list is implemented by arrays.

Algorithm (Binary Search)

Algorithm (Binary Search)

```
    Algorithm Binary_Search()
{
    mid = (beg + end) / 2
    while (beg ≤ end and data [mid] ! = item)
            {
            if (item < data [mid])
                    end = mid -1
              else
                    beg = mid + 1
                    mid = (beg  + end) /2
            }
            if (data [mid] = = item)
                    Loc = mid
               else
                    Loc = Null
}
```

Here data [ ] is sorted array and item is given item of information the variable beg, end, mid, denote, respectively, the beginning, end, and middle location's of a segment of elements of data, this above algorithm finds the locations Loc of item in data array or set Loc = Null if element is not present

Example: 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. 1.6 where the values of DATA [BEG] and DATA [END] in each stage of the algorithm are indicated by circles and the value of DATA [MID] by a square. Specifically, BEG, END and MID will have the following successive values:

(1) Initially, BEG = 1 and END = 13, Hence
MID = INT [(1+13)/2] = 7 and so DATA [MID] = 55

(2) Since 40 < 55, END has its value changed by END = MID-1=6, Hence.
MID=INT [(1+6)/2] = 3 and so DATA [MID] = 30

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

MID = INT[(4+6)/2] = 5 and so DATA [MID] = 40
We have found ITEM in location LOC = MID =5.
(1) 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99
(2) 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99
(3) 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99 [Successful]

Figure 1.6

Time Complexity of Binary Search

In this case the list is divided into two parts and every time our searching is reduced to one part of the list. Only one comparison is required when list is having n items, then next comparison is when elements become equal to n/2, then next when elements become equal to n/4 and so on till the elements reduce to 1.

Hence the **Time Complexity** is **O (log n)** for **worst case** and **O(log n) / 2)** for average case log n < n for all n > = 1. Hence, binary search is faster as compared to linear search. The only disadvantage for binary search is that the list must be sorted to search an element.

Binary Search	Time Complexity
Average Case	O (n/2)
Worst case	O (n)