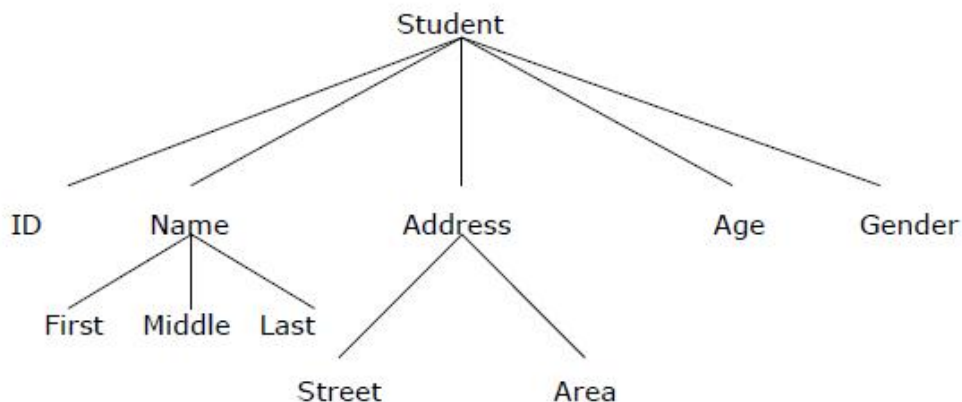


# UNIT-1

## **1.1 BASIC TERMINOLOGY: ELEMENTARY DATA ORGANIZATION**

### **1.1.1 Data and Data Item**

Data are simply collection of facts and figures. Data are values or set of values. A data item refers to a single unit of values. Data items that are divided into sub items are group items; those that are not are called elementary items. For example, a student's name may be divided into three sub items – [first name, middle name and last name] but the ID of a student would normally be treated as a single item.



In the above example ( ID, Age, Gender, First, Middle, Last, Street, Area ) are elementary data items, whereas (Name, Address ) are group data items.

### **1.1.2 Data Type**

Data type is a classification identifying one of various types of data, such as floating-point, integer, or Boolean, that determines the possible values for that type; the operations that can be done on values of that type; and the way values of that type can be stored. It is of two types: Primitive and non-primitive data type. Primitive data type is the basic data type that is provided by the programming language with built-in support. This data type is native to the language and is supported by machine directly while non-primitive data type is derived from primitive data type. For example- array, structure etc.

### **1.1.3 Variable**

It is a symbolic name given to some known or unknown quantity or information, for the purpose of allowing the name to be used independently of the information it represents. A variable name in computer source code is usually associated with a data storage location and thus also its contents and these may change during the course of program execution.

### **1.1.4 Record**

Collection of related data items is known as record. The elements of records are usually called fields or members. Records are distinguished from arrays by the fact that their number of fields is typically fixed, each field has a name, and that each field may have a different type.

### 1.1.5 Program

A sequence of instructions that a computer can interpret and execute is termed as program.

### 1.1.6 Entity

An entity is something that has certain attributes or properties which may be assigned some values. The values themselves may be either numeric or non-numeric.

Example:

Attributes:	<b>Name</b>	<b>Age</b>	<b>Gender</b>	<b>Social Society number</b>
Values:	Hamza	20	M	134-24-5533
	Ali Rizwan	23	M	234-9988775
	Fatima	20	F	345-7766443

### 1.1.7 Entity Set

An entity set is a group of or set of similar entities. For example, employees of an organization, students of a class etc. Each attribute of an entity set has a range of values, the set of all possible values that could be assigned to the particular attribute. The term “*information*” is sometimes used for data with given attributes, of, in other words meaningful or processed data.

### 1.1.8 Field

A field is a single elementary unit of information representing an attribute of an entity, a record is the collection of field values of a given entity and a file is the collection of records of the entities in a given entity set.

### 1.1.9 File

File is a collection of records of the entities in a given entity set. For example, file containing records of students of a particular class.

### 1.1.10 Key

A key is one or more field(s) in a record that take(s) unique values and can be used to distinguish one record from the others.

## 1.2 ALGORITHM

A well-defined computational procedure that takes some value, or a set of values, as input and produces some value, or a set of values, as output. It can also be defined as sequence of computational steps that transform the input into the output.

An algorithm can be expressed in three ways:-

- (i) in any natural language such as English, called pseudo code.
- (ii) in a programming language or
- (iii) in the form of a flowchart.

## 1.3 EFFICIENCY OF AN ALGORITHM

In computer science, algorithmic efficiency are the properties of an algorithm which relate to the amount of resources used by the algorithm. An algorithm must be analyzed to determine its

resource usage. Algorithmic efficiency can be thought of as analogous to engineering productivity for a repeating or continuous process.

For maximum efficiency we wish to minimize resource usage. However, the various resources (e.g. time, space) can not be compared directly, so which of two algorithms is considered to be more efficient often depends on which measure of efficiency is being considered as the most important, e.g. is the requirement for high speed, or for minimum memory usage, or for some other measure. It can be of various types:

- Worst case efficiency: It is the maximum number of steps that an algorithm can take for any collection of data values.
- Best case efficiency: It is the minimum number of steps that an algorithm can take any collection of data values.
- Average case efficiency: It can be defined as
  - the efficiency averaged on all possible inputs
  - must assume a distribution of the input
  - We normally assume uniform distribution (all keys are equally probable)

If the input has size  $n$ , efficiency will be a function of  $n$

## 1.4 TIME AND SPACE COMPLEXITY

Complexity of algorithm is a function of size of input of a given problem instance which determines how much running time/memory space is needed by the algorithm in order to run to completion.

Time Complexity: Time complexity of an algorithm is the amount of time it needs in order to run to completion.

Space Complexity: Space Complexity of an algorithm is the amount of space it needs in order to run to completion.

There are two points which we should consider about computer programming:-

- (i) An appropriate data structure and
- (ii) An appropriate algorithm.

## 1.5 ASYMPTOTIC NOTATIONS

### 1.5.1 Asymptotic

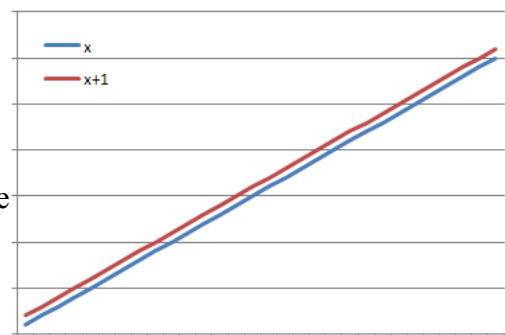
It means a line that continually approaches a given curve but does not meet it at any finite distance.

Example

$x$  is asymptotic with  $x + 1$  as shown in graph.

Asymptotic may also be defined as a way to describe the behavior of functions in the limit or without bounds.

Let  $f(x)$  and  $g(x)$  be functions from the set of real



numbers to the set of real numbers.

We say that  $f$  and  $g$  are asymptotic and write  $f(x) \approx g(x)$  if

$$\lim_{x \rightarrow \infty} f(x) / g(x) = c \text{ (constant)}$$

## 1.5.2 Asymptotic Notations

### 1.7.2.1 Big-Oh Notation (O)

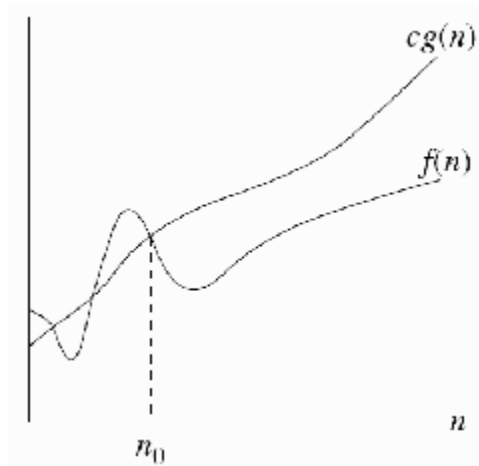
It provides possibly asymptotically tight **upper** bound for  $f(n)$  and it does not give best case complexity but can give worst case complexity.

Let  $f$  be a nonnegative function. Then we define the three most common asymptotic bounds as follows.

We say that  $f(n)$  is Big-O of  $g(n)$ , written as  $f(n) = O(g(n))$ , iff there are positive constants  $c$  and  $n_0$  such that

$$0 \leq f(n) \leq c g(n) \text{ for all } n \geq n_0$$

If  $f(n) = O(g(n))$ , we say that  $g(n)$  is an upper bound on  $f(n)$ .



**Example** -  $n^2 + 50n = O(n^2)$

$$0 \leq h(n) \leq cg(n)$$

$$0 \leq n^2 + 50n \leq cn^2$$

$$0/n^2 \leq n^2/n^2 + 50n/n^2 \leq cn^2/n^2 \quad \text{Divide by } n^2$$

$$0 \leq 1 + 50/n \leq c \quad \text{Note that } 50/n \rightarrow 0 \text{ as } n \rightarrow \infty$$

Pick  $n = 50$

$$0 \leq 1 + 50/50 = 2 \leq c = 2$$

With  $c=2$

$$0 \leq 1 + 50/n_0 \leq 2$$

Find  $n_0$

$$-1 \leq 50/n_0 \leq 1$$

$$-20n_0 \leq 50 \leq n_0 = 50$$

$$n_0=50$$

$$0 \leq n^2 + 50n \leq 2n^2$$

$$\forall n \geq n_0=50, c=2$$

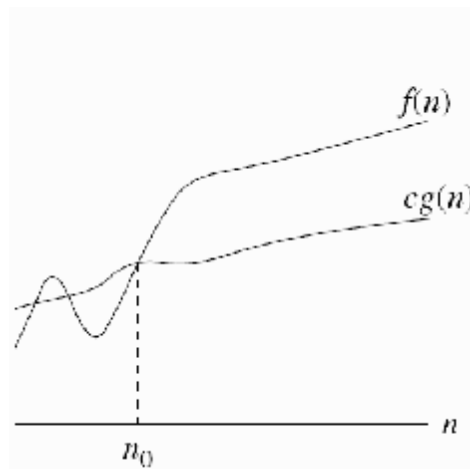
### 1.7.2.2 Big-Omega Notation ( $\Omega$ )

It provides possibly asymptotically tight **lower** bound for  $f(n)$  and it does not give worst case complexity but can give best case complexity

$f(n)$  is said to be Big-Omega of  $g(n)$ , written as  $f(n) = \Omega(g(n))$ , iff there are positive constants  $c$  and  $n_0$  such that

$$0 \leq c g(n) \leq f(n) \text{ for all } n \geq n_0$$

If  $f(n) = \Omega(g(n))$ , we say that  $g(n)$  is a lower bound on  $f(n)$ .



**Example** -  $n^3 = \Omega(n^2)$  with  $c=1$  and  $n_0=1$

$$0 \leq cg(n) \leq h(n)$$

$$0 \leq 1 \cdot 1^2 = 1 \leq 1 = 1^3$$

$$0 \leq cg(n) \leq h(n)$$

$$0 \leq cn^2 \leq n^3$$

$$0/n^2 \leq cn^2/n^2 \leq n^3/n^2$$

$$0 \leq c \leq n$$

$$0 \leq 1 \leq 1 \quad \text{with } c=1 \text{ and } n_0=1 \text{ since } n \text{ increases.}$$

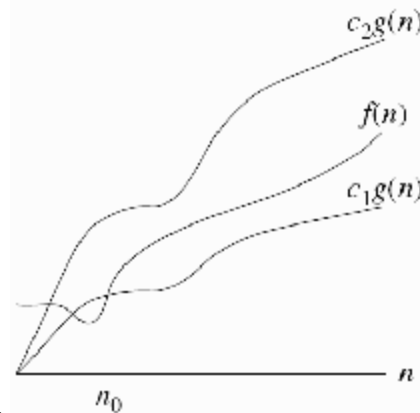
$$\lim_{n \rightarrow \infty} n = \infty$$

### 1.7.2.3 Big-Theta Notation ( $\Theta$ )

- We say that  $f(n)$  is Big-Theta of  $g(n)$ , written as  $f(n) = \Theta(g(n))$ , iff there are positive constants  $c_1$ ,  $c_2$  and  $n_0$  such that

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0$$

Equivalently,  $f(n) = \Theta(g(n))$  if and only if  $f(n) = O(g(n))$  and  $f(n) = \Omega(g(n))$ . If  $f(n) = \Theta(g(n))$ , we



say that  $g(n)$  is a tight bound on  $f(n)$ .

**Example -**  $n^2/2 - 2n = \Theta(n^2)$

$$c_1 g(n) \leq h(n) \leq c_2 g(n)$$

$$c_1 n^2 \leq n^2/2 - 2n \leq c_2 n^2$$

$$c_1 n^2/n^2 \leq n^2/2n^2 - 2n/n^2 \leq c_2 n^2/n^2 \quad \text{Divide by } n^2$$

$$c_1 \leq 1/2 - 2/n \leq c_2$$

O Determine  $c_2 = \frac{1}{2}$

$$\frac{1}{2} - \frac{2}{n} \leq c_2 = \frac{1}{2}$$

$$\lim_{n \rightarrow \infty} \frac{1}{2} - \frac{2}{n} = \frac{1}{2}$$

maximum of  $\frac{1}{2} - \frac{2}{n}$

$\Omega$  Determine  $c_1 = 1/10$

$$0 < c_1 \leq \frac{1}{2} - \frac{2}{n}$$

$$0 < c_1 \text{ minimum when } n=5$$

$$0 < c_1 \leq \frac{1}{2} - \frac{2}{5}$$

$$0 < c_1 \leq \frac{5}{10} - \frac{4}{10} = \frac{1}{10}$$

$n_0$  Determine  $n_0 = 5$

$$c_1 \leq \frac{1}{2} - \frac{2}{n_0} \leq c_2$$

$$\frac{1}{10} \leq \frac{1}{2} - \frac{2}{n_0} \leq \frac{1}{2}$$

$$\frac{1}{10} - \frac{1}{2} \leq -\frac{2}{n_0} \leq 0$$

Subtract  $\frac{1}{2}$

$$-\frac{4}{10} \leq -\frac{2}{n_0} \leq 0$$

$$-\frac{4}{10} n_0 \leq -2 \leq 0$$

Multiply by  $n_0$

$$-n_0 \leq -2 \cdot \frac{10}{4} \leq 0$$

Multiply by  $10/4$

$$n_0 \geq 2 \cdot \frac{10}{4} \geq 0$$

Multiply by -1

$$n_0 \geq 5 \geq 0$$

$$n_0 \geq 5$$

$n_0 = 5$  satisfies

$$\Theta \quad 0 < c_1 n^2 \leq \frac{n^2}{2} - 2n \leq c_2 n^2 \quad \forall n \geq n_0$$

with  $c_1 = 1/10$ ,  $c_2 = 1/2$  and  $n_0 = 5$

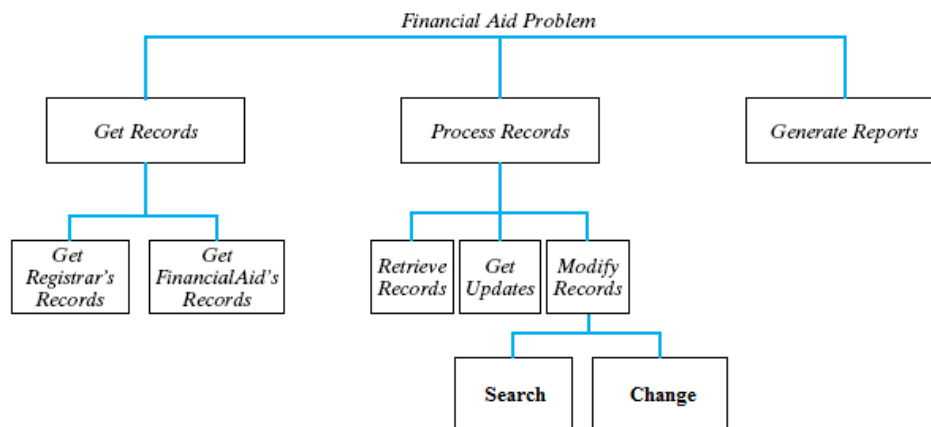
### 1.5.3 Time Space Trade-off

The best algorithm to solve a given problem is one that requires less memory space and less time to run to completion. But in practice, it is not always possible to obtain both of these objectives. One algorithm may require less memory space but may take more time to complete its execution. On the other hand, the other algorithm may require more memory space but may take less time to run to completion. Thus, we have to sacrifice one at the cost of other. In other words, there is Space-Time trade-off between algorithms.

If we need an algorithm that requires less memory space, then we choose the first algorithm at the cost of more execution time. On the other hand if we need an algorithm that requires less time for execution, then we choose the second algorithm at the cost of more memory space.

## 1.6 ABSTRACT DATA TYPE

It can be defined as a collection of data items together with the operations on the data. The word “abstract” refers to the fact that the data and the basic operations defined on it are being studied independently of how they are implemented. It involves **what** can be done with the data, not **how** has to be done. For ex, in the below figure the user would be involved in checking that what can be done with the data collected not how it has to be done.



An implementation of ADT consists of storage structures to store the data items and algorithms for basic operation. All the data structures i.e. array, linked list, stack, queue etc are examples of ADT.

## 1.7 DATA STRUCTURE

In computer science, a data structure is a particular way of storing and organizing data in a computer's memory so that it can be used efficiently. Data may be organized in many different ways; the logical or mathematical model of a particular organization of data is called a data structure. The choice of a particular data model depends on the two considerations first; it must be rich enough in structure to mirror the actual relationships of the data in the real world. On the other hand, the structure should be simple enough that one can effectively process the data whenever necessary.

### 1.7.1 Need of data structure

- It gives different level of organization data.
- It tells how data can be stored and accessed in its elementary level.
- Provide operation on group of data, such as adding an item, looking up highest priority item.
- Provide a means to manage huge amount of data efficiently.



- Provide fast searching and sorting of data.

### 1.7.2 Selecting a data structure

Selection of suitable data structure involve following steps –

- Analyze the problem to determine the resource constraints a solution must meet.
- Determine basic operation that must be supported. Quantify resource constraint for each operation
- Select the data structure that best meets these requirements.
- Each data structure has cost and benefits. Rarely is one data structure better than other in all situations. A data structure require :
  - Space for each item it stores
  - Time to perform each basic operation
  - Programming effort.

Each problem has constraints on available time and space. Best data structure for the task requires careful analysis of problem characteristics.

### 1.7.3 Type of data structure

#### 1.7.3.1 Static data structure

A data structure whose organizational characteristics are invariant throughout its lifetime. Such structures are well supported by high-level languages and familiar examples are arrays and records. The prime features of static structures are

- (a) None of the structural information need be stored explicitly within the elements – it is often held in a distinct logical/physical header;
- (b) The elements of an allocated structure are physically contiguous, held in a single segment of memory;
- (c) All descriptive information, other than the physical location of the allocated structure, is determined by the structure definition;
- (d) Relationships between elements do not change during the lifetime of the structure.

#### 1.7.3.2 Dynamic data structure

A data structure whose organizational characteristics may change during its lifetime. The adaptability afforded by such structures, e.g. linked lists, is often at the expense of decreased efficiency in accessing elements of the structure. Two main features distinguish dynamic structures from static data structures. Firstly, it is no longer possible to infer all structural information from a header; each data element will have to contain information relating it logically to other elements of the structure. Secondly, using a single block of contiguous storage is often not appropriate, and hence it is necessary to provide some storage management scheme at run-time.

#### 1.7.3.3 Linear Data Structure

A data structure is said to be linear if its elements form any sequence. There are basically two ways of representing such linear structure in memory.

- a) One way is to have the linear relationships between the elements represented by means of sequential memory location. These linear structures are called arrays.

b) 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.

The common examples of linear data structure are arrays, queues, stacks and linked lists.

#### 1.7.3.4 Non-linear Data Structure

This structure is mainly used to represent data containing a hierarchical relationship between elements. E.g. graphs, family trees and table of contents.

## 1.9 A BRIEF DESCRIPTION OF DATA STRUCTURES

### 1.8.1 Array

The simplest type of data structure is a linear (or one dimensional) array. A list of a finite number  $n$  of similar data referenced respectively by a set of  $n$  consecutive numbers, usually 1, 2, 3 . . . . .  $n$ . if we choose the name **A** for the array, then the elements of **A** are denoted by subscript notation

$A_1, A_2, A_3 \dots A_n$

or by the parenthesis notation

$A(1), A(2), A(3) \dots A(n)$

or by the bracket notation

$A[1], A[2], A[3] \dots A[n]$

#### Example:

A linear array **A[8]** consisting of numbers is pictured in following figure.



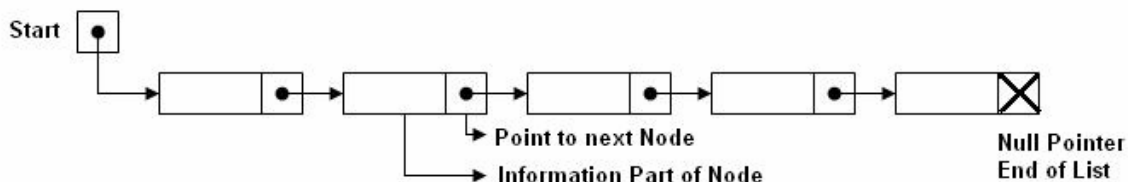
### 1.8.2 Linked List

A linked list or one way list is a linear collection of data elements, called nodes, where the linear order is given by means of pointers. Each node is divided into two parts:

- The first part contains the information of the element/node
- The second part contains the address of the next node (link /next pointer field) in the list.

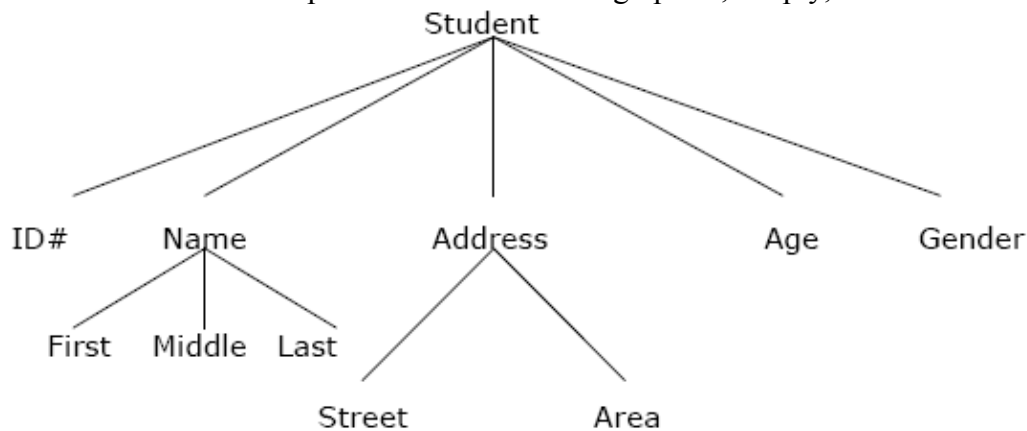
There is a special pointer Start/List contains the address of first node in the list. If this special pointer contains null, means that List is empty.

#### Example:



### 1.8.3 Tree

Data frequently contain a hierarchical relationship between various elements. The data structure which reflects this relationship is called a rooted tree graph or, simply, a tree.



### 1.8.4 Graph

Data sometimes contains a relationship between pairs of elements which is not necessarily hierarchical in nature, e.g. an airline flights only between the cities connected by lines. This data structure is called Graph.

### 1.8.5 Queue

A queue, also called FIFO system, is a linear list in which deletions can take place only at one end of the list, the Front of the list and insertion can take place only at the other end Rear.

### 1.8.6 Stack

It is an ordered group of homogeneous items or elements. Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack). The last element to be added is the first to be removed (LIFO: Last In, First Out).

## 1.9 DATA STRUCTURES OPERATIONS

The data appearing in our data structures are processed by means of certain operations. In fact, the particular data structure that one chooses for a given situation depends largely in the frequency with which specific operations are performed.

The following four operations play a major role in this text:

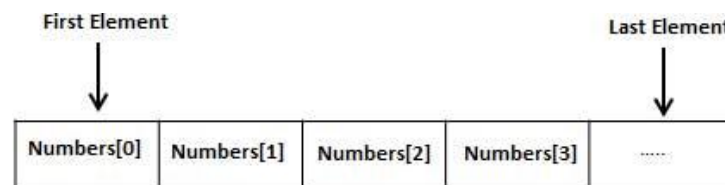
- **Traversing:** accessing each record/node exactly once so that certain items in the record may be processed. (This accessing and processing is sometimes called “visiting” the record.)
- **Searching:** Finding the location of the desired node with a given key value, or finding the locations of all such nodes which satisfy one or more conditions.
- **Inserting:** Adding a new node/record to the structure.
- **Deleting:** Removing a node/record from the structure.

## 1.10 ARRAYS: DEFINITION

C programming language provides a data structure called the array, which can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



The array may be categorized into –

- One dimensional array
- Two dimensional array
- Multidimensional array

### 1.10.1 Representation of One-Dimensional Array

In Pascal language we can define array as

VAR X: array [ 1 ... N] of integer {or any other type}

That's means the structure contains a set of data elements, numbered (N), for example called (X), its defined as type of element, the second type is the index type, is the type of values used to access individual element of the array, the value of index is

$$1 \leq I \leq N$$

By this definition the compiler limits the storage region to storing set of element, and the first location is individual element of array, and this called the Base Address, let's be as 500. Base Address (501) and like for the all elements and used the index I, by its value are range  $1 \leq I \leq N$  according to Base Index (500), by using this relation:

$$\text{Location ( X[I] )} = \text{Base Address} + (I-1)$$

When the requirement to bounding the forth element (I=4):

$$\text{Location ( X[4] )} = 500 + (4-1)$$

$$= 500 + 3$$

$$= 503$$

So the address of forth element is 503 because the first element in 500.

When the program indicate or dealing with element of array in any instruction like (write (X [I]), read (X [I] ) ), the compiler depend on going relation to bounding the requirement address.

### 1.10.2 Two-Dimensional Arrays

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y you would write something as follows:

```
type arrayName [ x ][ y ];
```

Where **type** can be any valid C data type and arrayName will be a valid C identifier. A two-dimensional array can be think as a table which will have x number of rows and y number of columns. A 2-dimensional array **a**, which contains three rows and four columns can be shown as below:

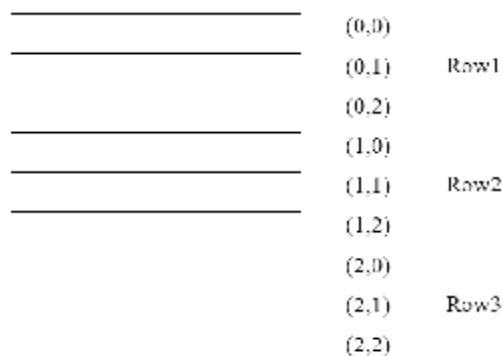
	Column 0	Column 1	Column 2	Column 3
Row 0	a[ 0 ][ 0 ]	a[ 0 ][ 1 ]	a[ 0 ][ 2 ]	a[ 0 ][ 3 ]
Row 1	a[ 1 ][ 0 ]	a[ 1 ][ 1 ]	a[ 1 ][ 2 ]	a[ 1 ][ 3 ]
Row 2	a[ 2 ][ 0 ]	a[ 2 ][ 1 ]	a[ 2 ][ 2 ]	a[ 2 ][ 3 ]

Thus, every element in array **a** is identified by an element name of the form **a[ i ][ j ]**, where **a** is the name of the array, and **i** and **j** are the subscripts that uniquely identify each element in **a**.

#### 1.10.2.1 Representation of two dimensional arrays in memory

A two dimensional 'm x n' Array A is the collection of m X n elements. Programming language stores the two dimensional array in one dimensional memory in either of two ways-

- **Row Major Order:** First row of the array occupies the first set of memory locations reserved for the array; Second row occupies the next set, and so forth.



To determine element address  $A[i,j]$ :

$$\text{Location} ( A[ i,j ] ) = \text{Base Address} + ( N \times ( I - 1 ) ) + ( j - 1 )$$

For example:

Given an array  $[1 \dots 5, 1 \dots 7]$  of integers. Calculate address of element  $T[4,6]$ , where  $BA=900$ .

Sol)  $I = 4$  ,  $J = 6$

$M = 5$  ,  $N = 7$

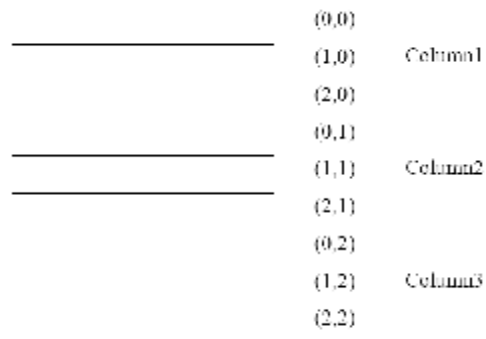
$$\text{Location} ( T [ 4,6 ] ) = BA + ( 7 \times ( 4-1 ) ) + ( 6-1 )$$

$$= 900 + ( 7 \times 3 ) + 5$$

$$= 900 + 21 + 5$$

$$= 926$$

- **Column Major Order:** Order elements of first column stored linearly and then comes elements of next column.



To determine element address  $A[i,j]$ :

$$\text{Location} ( A[ i,j ] ) = \text{Base Address} + ( M \times ( j - 1 ) ) + ( i - 1 )$$

For example:

Given an array  $[1 \dots 6, 1 \dots 8]$  of integers. Calculate address element  $T[5,7]$ , where  $BA=300$

Sol)  $I = 5$  ,  $J = 7$

$M = 6$  ,  $N = 8$

$$\text{Location} ( T [ 4,6 ] ) = BA + ( 6 \times ( 7-1 ) ) + ( 5-1 )$$

$$= 300 + ( 6 \times 6 ) + 4$$

$$= 300 + 36 + 4$$

$$= 340$$

### 1.10.3 Representation of Three & Four Dimensional Array

By the same way we can determine address of element for three and four dimensional array:

#### Three Dimensional Array

To calculate address of element  $X[i,j,k]$  using row-major order :

$$\text{Location} ( X[i,j,k] ) = BA + MN (k-1) + N (i-1) + (j-1)$$

using column-major order

$$\text{Location} ( X[i,j,k] ) = BA + MN (k-1) + M (j-1) + (i-1)$$

#### Four Dimensional Array

To calculate address of element  $X[i,j,k]$  using row-major order :

$$\text{Location} ( Y[i,j,k,l] ) = BA + MNR (l-1) + MN (k-1) + N (i-1) + (j-1)$$

using column-major order

$$\text{Location} ( Y[i,j,k,l] ) = BA + MNR (l-1) + MN (k-1) + M (j-1) + (i-1)$$

For example:

Given an array [ 1..8, 1..5, 1..7 ] of integers. Calculate address of element  $A[5,3,6]$ , by using rows & columns methods, if  $BA=900$ ?

Sol) The dimensions of A are :

$$M=8, N=5, R=7$$

$$i=5, j=3, k=6$$

Rows- wise

$$\text{Location} ( A[i,j,k] ) = BA + MN(k-1) + N(i-1) + (j-1)$$

$$\text{Location} ( A[5,3,6] ) = 900 + 8 \times 5(6-1) + 5(5-1) + (3-1)$$

$$= 900 + 40 \times 5 + 5 \times 4 + 2$$

$$= 900 + 200 + 20 + 2$$

$$= 1122$$

Columns- wise

$$\text{Location (A[i,j,k])} = BA + MN(k-1) + M(j-1) + (i-1)$$

$$\text{Location (A[5,3,6])} = 900 + 8 \times 5(6-1) + 8(3-1) + (5-1)$$

$$= 900 + 40 \times 5 + 8 \times 2 + 4$$

$$= 900 + 200 + 16 + 4$$

#### 1.10.4 Operations on array

**a) Traversing:** means to visit all the elements of the array in an operation is called traversing.

**b) Insertion:** means to put values into an array

**c) Deletion / Remove:** to delete a value from an array.

**d) Sorting:** Re-arrangement of values in an array in a specific order (Ascending or Descending) is called sorting.

**e) Searching:** The process of finding the location of a particular element in an array is called searching.

#### a) Traversing in Linear Array:

It means processing or visiting each element in the array exactly once; Let 'A' is an array stored in the computer's memory. If we want to display the contents of 'A', it has to be traversed i.e. by accessing and processing each element of 'A' exactly once.

**Algorithm:** (Traverse a Linear Array) Here **LA** is a Linear array with lower boundary **LB** and upper boundary **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  $K \leq UB$ .
3. [Visit element.] Apply PROCESS to  $LA[K]$ .
4. [Increase counter.] Set  $k = K + 1$ .  
[End of Step 2 loop.]
5. Exit.

The alternate algorithm for traversing (using for loop) is :

**Algorithm:** (Traverse a Linear Array) This algorithm traverse 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.

**This program will traverse each element of the array to calculate the sum and then calculate & print the average of the following array of integers.**

( 4, 3, 7, -1, 7, 2, 0, 4, 2, 13)



```

#include <iostream.h>
#define size 10 // another way int const size = 10
int main()
{ int x[10]={4,3,7,-1,7,2,0,4,2,13}, i, sum=0, LB=0, UB=size;
float av;
for(i=LB; i<UB; i++) sum = sum + x[i];
av = (float)sum/size;
cout<< "The average of the numbers=" << av << endl;
return 0;
}

```

### b) Sorting in Linear Array:

**Sorting** an array is the ordering the array elements in *ascending* (increasing from min to max) or *descending* (decreasing from max to min) order.

#### Bubble Sort:

The technique we use is called “*Bubble Sort*” because the bigger value gradually bubbles their way up to the top of array like air bubble rising in water, while the small values sink to the bottom of array. This technique is to make several passes through the array. On each pass, successive pairs of elements are compared. If a pair is in increasing order (or the values are identical), we leave the values as they are. If a pair is in decreasing order, their values are swapped in the array.

Pass = 1	Pass = 2	Pass = 3	Pass = 4
<u>2</u> 1 5 7 4 3	1 <u>2</u> 5 4 3 7	1 <u>2</u> 4 3 5 7	1 <u>2</u> 3 4 5 7
1 <u>2</u> 5 7 4 3	1 <u>2</u> 5 4 3 7	1 <u>2</u> 4 3 5 7	1 <u>2</u> 3 4 5 7
1 2 <u>5</u> 7 4 3	1 2 <u>5</u> 4 3 7	1 2 <u>4</u> 3 5 7	1 2 <u>3</u> 4 5 7
1 2 5 <u>7</u> 4 3	1 2 4 <u>5</u> 3 7	1 2 3 <u>4</u> 5 7	
1 2 5 4 <u>7</u> 3	1 2 4 3 <u>5</u> 7		
1 2 5 4 3 <u>7</u>			

➤ Underlined pairs show the comparisons. For each pass there are size-1 comparisons.  
 ➤ Total number of comparisons = (size-1)<sup>2</sup>

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

1. for pass=1 to N-1.
2.   for (i=0; i<= N-Pass; i++)
3.    If DATA[i]>DATA[i+1], then:  
             Interchange DATA[i] and DATA[i+1].  
             [End of If Structure.]
- [End of inner loop.]
- [End of Step 1 outer loop.]
4. Exit.

*/\* This program sorts the array elements in the ascending order using bubble sort method \*/*

```
#include <iostream.h>
int const SIZE = 6
void BubbleSort(int [ ], int);
int main()
{
  int a[SIZE]= {77,42,35,12,101,6};
  int i;
  cout<< "The elements of the array before sorting\n";
  for (i=0; i<= SIZE-1; i++) cout<< a[i]<<" ";
  BubbleSort(a, SIZE);
  cout<< "\n\nThe elements of the array after sorting\n";
  for (i=0; i<= SIZE-1; i++) cout<< a[i]<<" ";
  return 0;
}
void BubbleSort(int A[ ], int N)
{
  int i, pass, hold;
  for (pass=1; pass<= N-1; pass++)
  {
    for (i=0; i<= SIZE-pass; i++)
    {
      if(A[i] >A[i+1])
      {
        hold =A[i];
        A[i]=A[i+1];
        A[i+1]=hold;
      }
    }
  }
}
```

### 1.10.5 Applications of array

Arrays are used to implement mathematical vectors and matrices, as well as other kinds of rectangular tables. Many databases, small and large, consist of (or include) one-dimensional arrays whose elements are records.

Arrays are used to implement other data structures, such as heaps, hash tables, deques, queues, stacks, strings, and VLists.

One or more large arrays are sometimes used to emulate in-program dynamic memory allocation, particularly memory pool allocation. Historically, this has sometimes been the only way to allocate "dynamic memory" portably.

Arrays can be used to determine partial or complete control flow in programs, as a compact alternative to (otherwise repetitive) multiple IF statements. They are known in this context as control tables and are used in conjunction with a purpose built interpreter whose control flow is altered according to values contained in the array.

#### **1.10.6 Sparse matrix**

Matrix with maximum zero entries is termed as sparse matrix. It can be represented as:

- Lower triangular matrix: It has non-zero entries on or below diagonal.
- Upper Triangular matrix: It has non-zero entries on or above diagonal.
- Tri-diagonal matrix: It has non-zero entries on diagonal and at the places immediately above or below diagonal.

### **1.11 STATIC AND DYNAMIC MEMORY ALLOCATION**

In many programming environments memory allocation to variables can be of two types static memory allocation and dynamic memory allocation. Both differ on the basis of time when memory is allocated. In static memory allocation memory is allocated to variable at compile time whereas in dynamic memory allocation memory is allocated at the time of execution. Other

differences between both memory allocation techniques are summarized below-

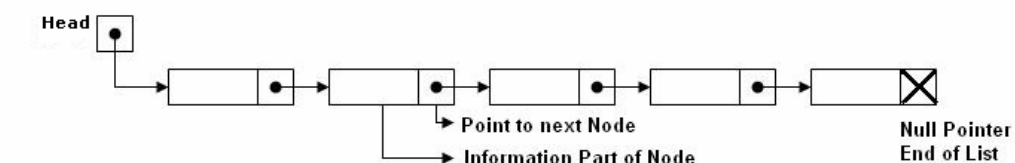
S. No.	Static memory allocation	Dynamic memory allocation
1.	Memory allotted before run time (mostly at compile time)	Memory allotted at run time.
2.	Programmer must know in advance amount of memory needed.	Programmer is not expected to have knowledge of amount of memory requirement in advance.
2.	Since we may only know maximum need therefore finite amount of memory allotted	Only required amount of memory allotted.
3.	Since maximum amount of memory is allocated, but actual usage may be far less therefore wastage is there.	Only required amount of memory is allocated therefore no wastage.
4.	No functions for memory request is needed. Memory will be allocated automatically.	Predefined functions for memory allocation request will be used such as malloc, calloc in C programming language
5.	Since memory is already allotted to variables before runtime therefore it save run time.	Time required in memory allotment will increase run time of program.
6.	Static allocation is fast	Dynamic allocation is slower.
7.	Allot memory of data segment of memory.	Allot memory on heap (Pool of free memory).
8.	The space is allotted once and is never freed.	Programmer can free block of memory once it has longer needed.
9.	Static memory allotted to elementary data item, array, structure	Memory allotted to pointer is dynamic

## 1.12 LINKED LIST

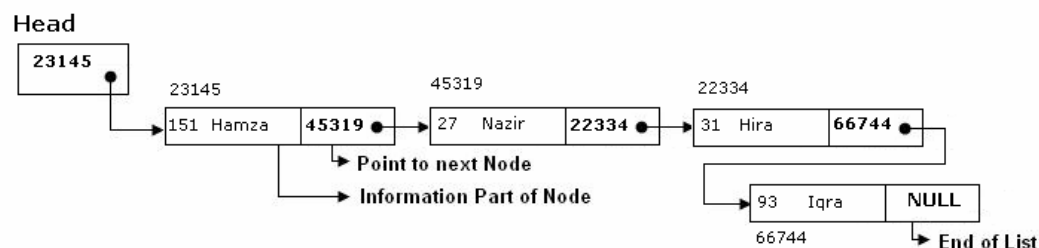
A linked list or one way list is a linear collection of data elements, called nodes, where the linear order is given by means of “pointers”. Each node is divided into two parts.

- The first part contains the information of the element.
- The second part called the link field contains the address of the next node in the list.

To see this more clearly lets look at an example:



For example:

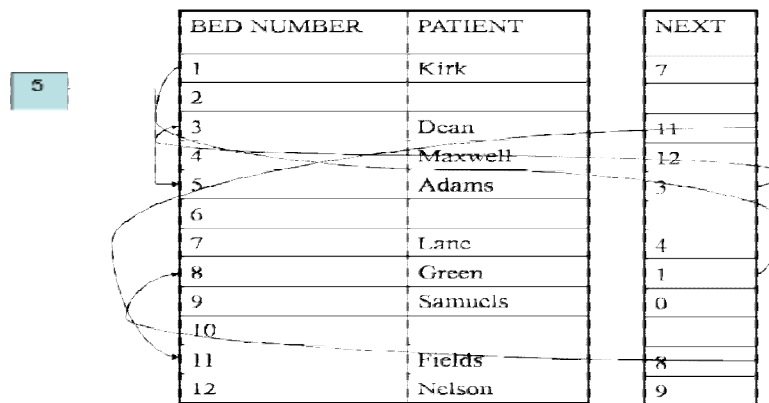


The **Head** is a special pointer variable which contains the address of the first node of the list. If there is no node available in the list then **Head** contains **NULL** value that means, List is empty. The left part of the each node represents the information part of the node, which may contain an

entire record of data (e.g. ID, name, marks, age etc). the right part represents pointer/link to the next node. The next pointer of the last node is **null** pointer signal the end of the list.

### 1.12.1 Representation of Linked list in memory

Linked list is maintained in memory by two linear arrays: **INFO** and **LINK** such that **INFO [K]** and **LINK [K]** contain respectively the information part and the next pointer field of node of **LIST**. **LIST** also requires a variable name such as **START** which contains the location of the beginning of the list and the next pointer denoted by **NULL** which indicates the end of the **LIST**.



### 1.12.4 Types of linked lists

- Singly linked list
  - Begins with a pointer to the first node
  - Terminates with a null pointer
  - Only traversed in one direction
- Circular, singly linked
  - Pointer in the last node points back to the first node
- Doubly linked list
  - Two “start pointers” – first element and last element
  - Each node has a forward pointer and a backward pointer
  - Allows traversals both forwards and backwards
- Circular, doubly linked list
  - Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node
- Header Linked List
  - Linked list contains a header node that contains information regarding complete linked list.

### 1.12.3 The Operations on the Linear Lists

Various operations are:

- 1- Search: This operation involves the searching of an element in the linked list.
- 2- Additional (Inserting) : To add new node to data structure.
- 3- Deletion : To delete a node from data structure.
- 4- Merge : To merge two structures or more to constituting one structure.
- 5- Split : To divide data structure to two structures or more.
- 6- Counting : counting some of items or nodes in data structure.

- 7- Copying : copy data of data structure to another data structure.
- 8- Sort : sort items or nodes in data structure according to the value of the field or set of fields.
- 9- Access : To access from node or item to another one may be need some of purposes to test or change...etc.

#### 1.12.4 Comparison of Linked List and Array

Comparison between array and linked list are summarized in following table –

S.No	Array	Linked List
1.	List of elements stored in contiguous memory location i.e. all elements linked physically.	List of elements need not stored in contiguous memory location i.e. all elements will be linked logically.
2.	Contiguous memory required for complete list that will be large requirements.	Contiguous memory required for single node of list that will be small requirements.
3.	List is static in nature i.e. created at compile time mostly.	List is dynamic i.e. created and manipulated at time of execution .
4.	List can't grow and shrink	List can grow and shrink dynamically
5.	Memory allotted to single item of list can't free	Memory allotted to single node of list can also be free.
6.	Random access of 1 <sup>st</sup> element is possible through a[1]	Sequential access to 1 <sup>st</sup> element i.e. all previous 1 -1 node have to traverse before.
7.	Traversal easy since any elements can access dynamically and randomly.	Traversal is done node by node hence not as good as in array.
8.	Searching can be linear and if sorted than in array we can also apply binary search of logarithm time complexity.	Searching operation must be linear in case of sorted list also. Time complexity is proportional to list length..
9.	Insertion Deletion is costly since require shifting of many items.	Insertion Deletion is performed by simply pointer exchange. Only set of pointer assignment statements can perform insertion deletion operation.

#### 1.12.5 Advantages

List of data can be stored in arrays but linked structures (pointers) provide several advantages:

A linked list is appropriate when the number of data elements to be represented in data structure is unpredictable. It also appropriate when there are frequently insertions & deletions occurred in the list. Linked lists are dynamic, so the length of a list can increase or decrease as necessary.

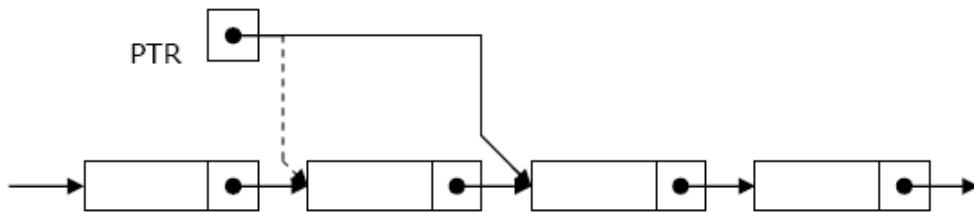
#### 1.12.6 Operations on Linked List

There are several operations associated with linked list i.e.

##### a) Traversing a Linked List

Suppose we want to traverse LIST in order to process each node exactly once. The traversing algorithm uses a pointer variable PTR which points to the node that is currently being processed. Accordingly, PTR->NEXT points to the next node to be processed so,

PTR=HEAD [ Moves the pointer to the first node of the list]  
 PTR=PTR->NEXT [ Moves the pointer to the next node in the list.]



**Algorithm:** (Traversing a Linked List) Let LIST be a linked list in memory. This algorithm traverses LIST, applying an operation PROCESS to each element of list. The variable PTR point to the node currently being processed.

1. Set PTR=HEAD. [Initializes pointer PTR.]
2. Repeat Steps 3 and 4 while PTR!=NULL.
3.     Apply PROCESS to PTR-> INFO.
4.     Set PTR= PTR-> NEXT             [PTR now points to the next node.]  
[End of Step 2 loop.]
5. Exit.

#### b) Searching a Linked List:

Let list be a linked list in the memory and a specific ITEM of information is given to search. If ITEM is actually a key value and we are searching through a LIST for the record containing ITEM, then ITEM can appear only once in the LIST. Search for wanted ITEM in List can be performed by traversing the list using a pointer variable PTR and comparing ITEM with the contents PTR->INFO of each node, one by one of list.

**Algorithm:** SEARCH(INFO, NEXT, HEAD, ITEM, PREV, CURR, SCAN)  
LIST is a linked list in the memory. This algorithm finds the location LOC of the node where ITEM first appear in LIST, otherwise sets LOC=NULL.

1. Set PTR=HEAD.
2. Repeat Step 3 and 4 while PTR!=NULL:
3.     if ITEM = PTR->INFO then:  
       Set LOC=PTR, and return. [Search is successful.]  
       [End of If structure.]
4.     Set PTR=PTR->NEXT  
       [End of Step 2 loop.]
5. Set LOC=NULL, and return. [Search is unsuccessful.]
6. Exit.

#### Searching in sorted list

Algorithm: SRCHSL (INFO, LINK, START, ITEM, LOC)

LIST is sorted list (Sorted in ascending order) in memory. This algorithm finds the location LOC of the node where ITEM first appears in LIST or sets LOC=NULL.

1. Set PTR:= START

2. Repeat while PTR  $\neq$  NULL

    If ITEM > INFO[PTR], then:

        Set PTR := LINK[PTR]

    Else If ITEM = INFO[PTR], then:

        Set LOC := PTR

    Return

    Else Set LOC:= NULL

    Return

    [End of If structure]

    [End of step 2 Loop]

3. Set LOC:= NULL

4. Return

### Search Linked List for insertion and deletion of Nodes:

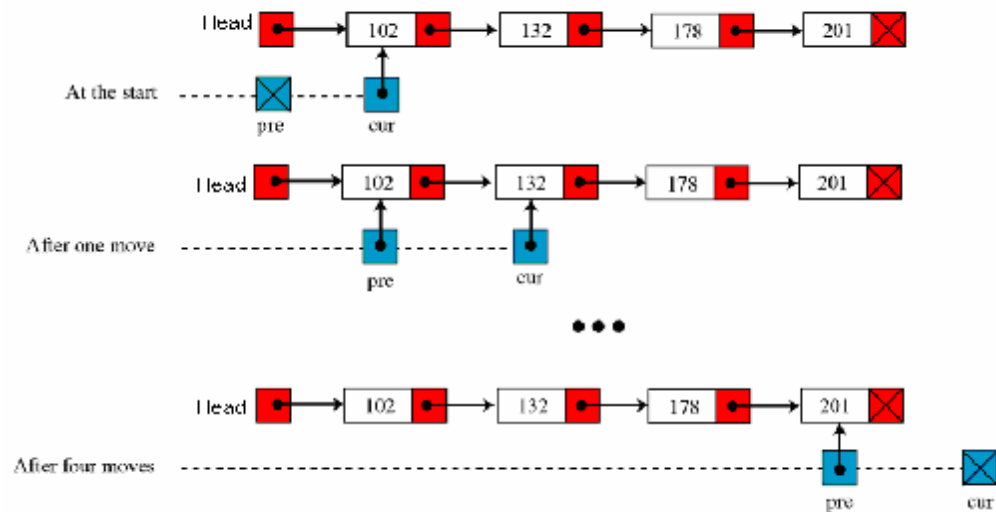
Both insertion and deletion operations need searching the linked list.

- To add a new node, we must identify the logical predecessor (address of previous node) where the new node is to be inserting.
- To delete a node, we must identify the location (addresses) of the node to be deleted and its logical predecessor (previous node).

### Basic Search Concept

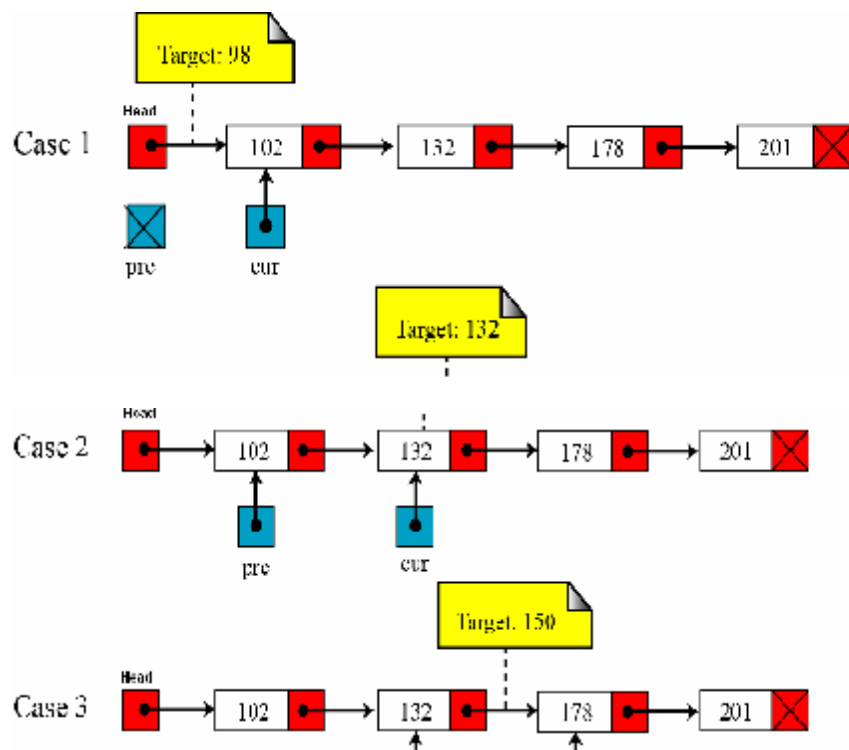
Assume there is a sorted linked list and we wish that after each insertion/deletion this list should always be sorted. Given a target value, the search attempts to locate the requested node in the linked list. Since nodes in a linked list have no names, we use two pointers, **pre** (for previous) and **cur** (for current) nodes. At the beginning of the search, the **pre** pointer is **null** and the **cur** pointer points to the first node (**Head**). The search algorithm moves the two pointers together towards the end of the list. Following Figure shows the movement of these two pointers through the list in an extreme case scenario: when the target value is larger than any value in the list.

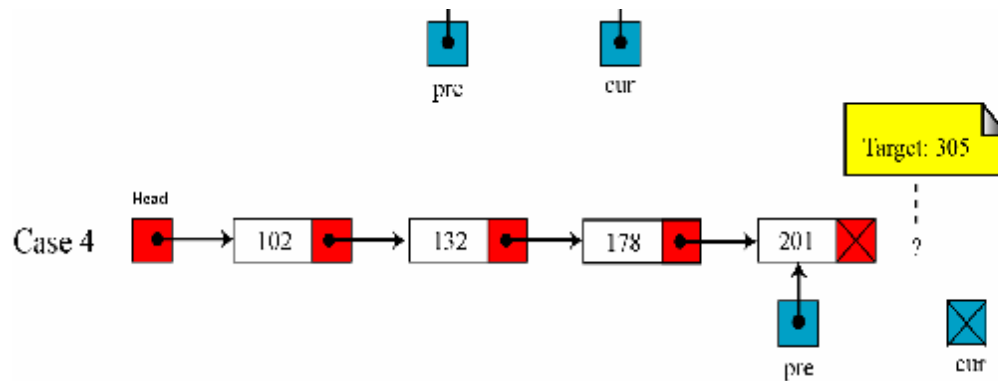




Moving of *pre* and *cur* pointers in searching a linked list

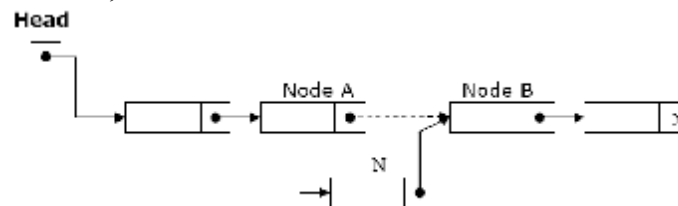
Values of *pre* and *cur* pointers in different cases





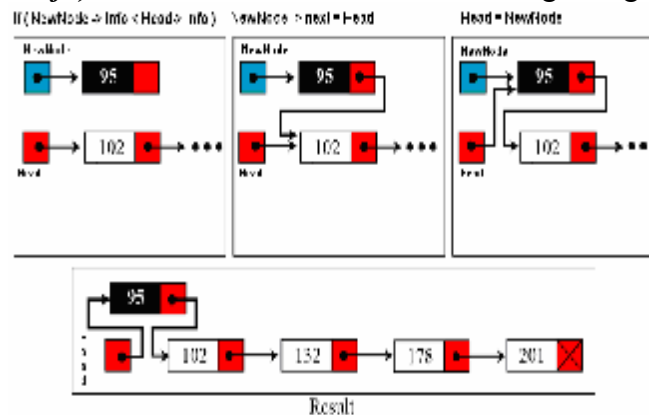
### c) Insertion into a Linked List:

If a node **N** is to be inserted into the list between nodes **A** and **B** in a linked list named **LIST**. Its schematic diagram would be;



### Inserting at the Beginning of a List:

If the linked list is sorted list and new node has the least low value already stored in the list i.e. *(if New->info < Head->info)* then new node is inserted at the beginning / Top of the list.



Algorithm: INSFIRST (INFO, LINK, START, AVAIL, ITEM)

This algorithm inserts ITEM as the first node in the list

Step 1: [OVERFLOW ?] If AVAIL=NULL, then

Write: OVERFLOW

Return

Step 2: [Remove first node from AVAIL list ]

Set NEW:=AVAIL and AVAIL:=LINK[AVAIL].

Step 3: Set INFO[NEW]:=ITEM [Copies new data into new node]

Step 4: Set LINK[NEW]:= START

[New node now points to original first node]

Step 5: Set START:=NEW [Changes START so it points to new  
node]

Step 6: Return

### **Inserting after a given node**

Algorithm: INSLOC(INFO, LINK, START, AVAIL, LOC, ITEM)

This algorithm inserts ITEM so that ITEM follows the  
node with location LOC or inserts ITEM as the first node  
when LOC =NULL

Step 1: [OVERFLOW] If AVAIL=NULL, then:  
Write: OVERFLOW

Return

Step 2: [Remove first node from AVAIL list]

Set NEW:=AVAIL and AVAIL:=LINK[AVAIL]

Step 3: Set INFO[NEW]:= ITEM [Copies new data into new node]

Step 4: If LOC=NULL, then:

Set LINK[NEW]:=START and START:=NEW

Else:

Set LINK[NEW]:=LINK[LOC] and LINK[LOC]:= NEW

[End of If structure]

Step 5: Return

### **Inserting a new node in list:**

The following algorithm inserts an ITEM into LIST.

**Algorithm: INSERT( ITEM)**

[This algorithm add newnodes at any position (Top, in Middle and at End) in the List ]

1. Create a **NewNode** node in memory
2. Set **NewNode -> INFO = ITEM**. [Copies new data into INFO of new node.]
3. Set **NewNode -> NEXT = NULL**. [Copies NULL in NEXT of new node.]
4. If **HEAD=NULL**, then **HEAD=NewNode** and return. [Add first node in list]
5. if **NewNode-> INFO < HEAD->INFO**  
then Set **NewNode->NEXT=HEAD** and **HEAD=NewNode** and return  
[Add node on top of existing list]
6. PrevNode = NULL, CurrNode=NULL;
7. for(CurrNode =HEAD; CurrNode != NULL; CurrNode = CurrNode ->NEXT)  
{ if(NewNode->INFO <= CurrNode ->INFO)  
{  
break the loop  
}  
PrevNode = CurrNode;  
} [ end of loop ]  
[Insert after PREV node (in middle or at end) of the list]
8. Set **NewNode->NEXT = PrevNode->NEXT** and
9. Set PrevNode->NEXT= **NewNode**.
- 10.Exit.

**d) Delete a node from list:**

The following algorithm deletes a node from any position in the LIST.

**Algorithm: DELETE(ITEM)**

LIST is a linked list in the memory. This algorithm deletes the node where ITEM first appear in LIST, otherwise it writes "NOT FOUND"

1. if **Head =NULL** then write: "Empty List" and return [Check for Empty List]
2. if **ITEM = Head -> info** then: [ Top node is to delete ]  
Set **Head = Head -> next** and return
3. Set PrevNode = NULL, CurrNode=NULL.
4. for(CurrNode =HEAD; CurrNode != NULL; CurrNode = CurrNode ->NEXT)  
{ if (ITEM = CurrNode ->INFO ) then:  
{  
break the loop  
}  
Set PrevNode = CurrNode;  
} [ end of loop ]
5. if(CurrNode = NULL) then write : Item not found in the list and return
6. [delete the current node from the list]  
Set PrevNode ->NEXT = CurrNode->NEXT
7. Exit.

**e) Concatenating two linear linked lists**

Algorithm: Concatenate(INFO,LINK,START1,START2)

This algorithm concatenates two linked lists with start  
pointers START1 and START2

Step 1: Set PTR:=START1

Step 2: Repeat while LINK[PTR]≠NULL:

Set PTR:=LINK[PTR]

[End of Step 2 Loop]

Step 3: Set LINK[PTR]:=START2

Step 4: Return

**// A Program that exercise the operations on Liked List**

```
#include<iostream.h>
#include <malloc.h>
#include <process.h>
struct node
{
int info;
struct node *next;
};
struct node *Head=NULL;
struct node *Prev,*Curr;
void AddNode(int ITEM)
{
struct node *NewNode;
NewNode = new node;
// NewNode=(struct node*)malloc(sizeof(struct node));
NewNode->info=ITEM; NewNode->next=NULL;
if(Head==NULL) { Head=NewNode; return; }
if(NewNode->info < Head->info)
{ NewNode->next = Head; Head=NewNode; return;}
Prev=Curr=NULL;
for(Curr = Head ; Curr != NULL ; Curr = Curr ->next)
{
if( NewNode->info < Curr ->info) break;
else Prev = Curr;
}
NewNode->next = Prev->next;
Prev->next = NewNode;
} // end of AddNode function
void DeleteNode()
{ int inf;
```

```

if(Head==NULL){ cout<< "\n\n empty linked list\n"; return;}
cout<< "\n Put the info to delete: ";
cin>>inf;
if(inf == Head->info) // First / top node to delete
{ Head = Head->next; return;}
Prev = Curr = NULL;
for(Curr = Head ; Curr != NULL ; Curr = Curr ->next )
{
if(Curr ->info == inf) break;
Prev = Curr;
}
if( Curr == NULL)
cout<<inf<< " not found in list \n";
else
{ Prev->next = Curr->next; }
} // end of DeleteNode function

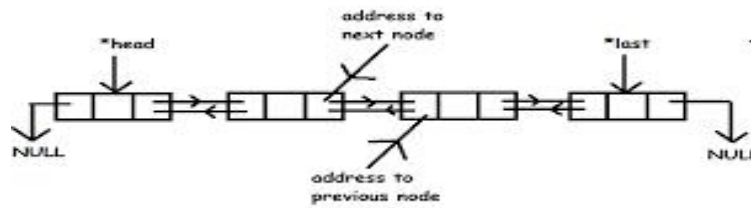
void Traverse()
{
for(Curr = Head; Curr != NULL ; Curr = Curr ->next )
cout<< Curr ->info<<"\t";
} // end of Traverse function
int main()
{ int inf, ch;
while(1)
{ cout<< " \n\n\n Linked List Operations\n\n";
cout<< " 1- Add Node \n 2- Delete Node \n";
cout<< " 3- Traverse List \n 4- exit\n";
cout<< "\n\n Your Choice: "; cin>>ch;
switch(ch)
{ case 1: cout<< "\n Put info/value to Add: ";
cin>>inf;
AddNode(inf);
break;
case 2: DeleteNode(); break;
case 3: cout<< "\n Linked List Values:\n";
Traverse(); break;
case 4: exit(0);
} // end of switch
} // end of while loop
return 0;
} // end of main ( ) function

```

### 1.12.7 Doubly Linked Lists

A doubly linked list is a list that contains links to next and previous nodes. Unlike singly

linked lists where traversal is only one way, doubly linked lists allow traversals in both ways.



### Dynamic Implementation of doubly linked list

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
    struct node *previous;
    int data;
    struct node *next;
} *head, *last;

void insert_begning(int value)
{
    struct node *var,*temp;
    var=(struct node *)malloc(sizeof(struct node));
    var->data=value;
    if(head==NULL)
    {
        head=var;
        head->previous=NULL;
        head->next=NULL;
        last=head;
    }
    else
    {
        temp=var;
```

```

        temp->previous=NULL;
        temp->next=head;
        head->previous=temp;
        head=temp;
    }
}

```

**void** insert\_end(**int** value)

```

{
    struct node *var,*temp;
    var=(struct node *)malloc(sizeof(struct node));
    var->data=value;
    if(head==NULL)
    {
        head=var;
        head->previous=NULL;
        head->next=NULL;
        last=head;
    }
    else
    {
        last=head;
        while(last!=NULL)
        {
            temp=last;
            last=last->next;
        }
        last=var;
        temp->next=last;
        last->previous=temp;
        last->next=NULL;
    }
}

```

**int** insert\_after(**int** value, **int** loc)

```

{

```



```

struct node *temp,*var,*temp1;
var=(struct node *)malloc(sizeof(struct node));
var->data=value;
    if(head==NULL)
    {
        head=var;
        head->previous=NULL;
        head->next=NULL;
    }
else
    {
        temp=head;
        while(temp!=NULL && temp->data!=loc)
        {
            temp=temp->next;
        }
        if(temp==NULL)
        {
            printf("\n%d is not present in list ",loc);
        }
        else
        {
            temp1=temp->next;
            temp->next=var;
            var->previous=temp;
            var->next=temp1;
            temp1->previous=var;
        }
    }
    last=head;
    while(last->next!=NULL)
    {
        last=last->next;
    }
}

int delete_from_end()

```

```

{
    struct node *temp;
    temp=last;
    if(temp->previous==NULL)
    {
        free(temp);
        head=NULL;
        last=NULL;
        return 0;
    }
    printf("\nData deleted from list is %d \n",last->data);
    last=temp->previous;
    last->next=NULL;
    free(temp);
    return 0;
}

```

```

int delete_from_middle(int value)
{
    struct node *temp,*var,*t, *temp1;
    temp=head;
    while(temp!=NULL)
    {
        if(temp->data == value)
        {
            if(temp->previous==NULL)
            {
                free(temp);
                head=NULL;
                last=NULL;
                return 0;
            }
            else
            {
                var->next=temp1;
                temp1->previous=var;
            }
        }
        temp=temp->next;
    }
}

```

```

        free(temp);
        return 0;
    }
}
else
{
    var=temp;
    temp=temp->next;
    temp1=temp->next;
}
}
printf("data deleted from list is %d",value);
}

```

```

void display()
{
    struct node *temp;
    temp=head;
    if(temp==NULL)
    {
        printf("List is Empty");
    }
    while(temp!=NULL)
    {
        printf("-> %d ",temp->data);
        temp=temp->next;
    }
}

```

```

int main()
{
    int value, i, loc;
    head=NULL;
    printf("Select the choice of operation on link list");
    printf("\n1.) insert at begning\n2.) insert at at\n3.) insert at middle");
    printf("\n4.) delete from end\n5.) reverse the link list\n6.) display list\n7.)exit");
}

```

```

while(1)
{
    printf("\n\nenter the choice of operation you want to do ");
    scanf("%d",&i);
    switch(i)
    {
        case 1:
        {
            printf("enter the value you want to insert in node ");
            scanf("%d",&value);
            insert_begning(value);
            display();
            break;
        }
        case 2:
        {
            printf("enter the value you want to insert in node at last ");
            scanf("%d",&value);
            insert_end(value);
            display();
            break;
        }
        case 3:
        {
            printf("after which data you want to insert data ");
            scanf("%d",&loc);
            printf("enter the data you want to insert in list ");
            scanf("%d",&value);
            insert_after(value,loc);
            display();
            break;
        }
        case 4:
        {
            delete_from_end();
            display();

```

```

        break;
    }
    case 5:
    {
        printf("enter the value you want to delete");
        scanf("%d",value);
        delete_from_middle(value);
        display();
        break;
    }
    case 6 :
    {
        display();
        break;
    }
    case 7 :
    {
        exit(0);
        break;
    }
}
}
printf("\n\n%d",last->data);
display();
getch();
}

```

### 1.13.8 Circular Linked List

A circular linked list is a linked list in which last element or node of the list points to first node. For non-empty circular linked list, there are no NULL pointers. The memory declarations for representing the circular linked lists are the same as for linear linked lists. All operations performed on linear linked lists can be easily extended to circular linked lists with following exceptions:

- While inserting new node at the end of the list, its next pointer field is made to point to the first node.
- While testing for end of list, we compare the next pointer field with address of the first node

Circular linked list is usually implemented using **header linked list**. Header linked list is a linked list which always contains a special node called the **header node**, at the beginning of the list. This header node usually contains vital information about the linked list such as number of nodes in lists, whether list is sorted or not etc. Circular header lists are frequently used instead of ordinary linked lists as many

- operations are much easier to state and implement using header list This comes from the following two properties of circular header linked lists:
- The null pointer is not used, and hence all pointers contain valid addresses
- Every (ordinary) node has a predecessor, so the first node may not require a special case.

**Algorithm: (Traversing a circular header linked list)**

This algorithm traverses a **circular header linked list** with  
START pointer storing the address of the header node.

Step 1: Set PTR:=LINK[START]

Step 2: Repeat while PTR≠START:

    Apply PROCESS to INFO[PTR]

    Set PTR:=LINK[PTR]

    [End of Loop]

Step 3: Return

**Searching a circular header linked list**

Algorithm: SRCHHL(INFO, LINK, START, ITEM, LOC)

This algorithm searches a circular header linked list

Step 1: Set PTR:=LINK[START]

Step 2: Repeat while INFO[PTR]≠ITEM and PTR≠START:

    Set PTR:=LINK[PTR]

    [End of Loop]

Step 3: If INFO[PTR]=ITEM, then:

    Set LOC:=PTR

    Else:

    Set LOC:=NULL

    [End of If structure]

Step 4: Return

**Deletion from a circular header linked list**

Algorithm: DELLOCHL(INFO, LINK, START, AVAIL, ITEM)

This algorithm deletes an item from a circular header

linked list.

Step 1: CALL FINDBHL(INFO, LINK, START, ITEM, LOC, LOCP)

Step 2: If LOC=NULL, then:

Write: 'item not in the list'

Exit

Step 3: Set LINK[LOCP]:=LINK[LOC] [Node deleted]

Step 4: Set LINK[LOC]:=AVAIL and AVAIL:=LOC

[Memory returned to Avail list]

Step 5: Return

### **Searching in circular list**

Algorithm: FINDBHL(INFO, LINK, START, ITEM, LOC, LOCP)

This algorithm finds the location of the node to be deleted  
and the location of the node preceding the node to be  
deleted

Step 1: Set SAVE:=START and PTR:=LINK[START]

Step 2: Repeat while INFO[PTR]≠ITEM and PTR≠START

Set SAVE:=PTR and PTR:=LINK[PTR]

[End of Loop]

Step 3: If INFO[PTR]=ITEM, then:

Set LOC:=PTR and LOCP:=SAVE

Else:

Set LOC:=NULL and LOCP:=SAVE

[End of If Structure]

Step 4: Return

### **Insertion in a circular header linked list**

Algorithm: INSRT(INFO, LINK, START, AVAIL, ITEM, LOC)

This algorithm inserts item in a circular header linked list  
after the location LOC

Step 1: If AVAIL=NULL, then

Write: 'OVERFLOW'

Exit

Step 2: Set NEW:=AVAIL and AVAIL:=LINK[AVAIL]

Step 3: Set INFO[NEW]:=ITEM

Step 4: Set LINK[NEW]:=LINK[LOC]

Set LINK[LOC]:=NEW

Step 5: Return

### **Insertion in a sorted circular header linked list**

Algorithm: INSERT(INFO, LINK, START, AVAIL, ITEM)

This algorithm inserts an element in a sorted circular header linked list

Step 1: CALL FINDA(INFO, LINK, START, ITEM, LOC)

Step 2: If AVAIL=NULL, then

Write: 'OVERFLOW'

Return

Step 3: Set NEW:=AVAIL and AVAIL:=LINK[AVAIL]

Step 4: Set INFO[NEW]:=ITEM

Step 5: Set LINK[NEW]:=LINK[LOC]

Set LINK[LOC]:=NEW

Step 6: Return

Algorithm: FINDA(INFO, LINK, ITEM, LOC, START)

This algorithm finds the location LOC after which to insert

Step 1: Set PTR:=START

Step 2: Set SAVE:=PTR and PTR:=LINK[PTR]

Step 3: Repeat while PTR≠START

If INFO[PTR]>ITEM, then

Set LOC:=SAVE

Return

Set SAVE:=PTR and PTR:=LINK[PTR]

[End of Loop]

Step 4: Set LOC:=SAVE

Step 5: Return

### **1.12.9 Polynomial representation and addition**

One of the most important application of Linked List is representation of a polynomial in memory. Although, polynomial can be represented using a linear linked list but common and preferred way of representing polynomial is using circular linked list with a header node.

Polynomial Representation: Header linked list are frequently used for maintaining polynomials in memory. The header node plays an important part in this representation since it is needed to represent the zero polynomial.

Specifically, the information part of node is divided into two fields representing respectively, the coefficient and the exponent of corresponding polynomial term and nodes are linked according to decreasing degree. List pointer variable POLY points to header node whose exponent field is



assigned a negative number, in this case -1. The array representation of List will require three linear arrays as COEFF, EXP and LINK.

### **Addition of polynomials using linear linked list representation for a polynomial**

Algorithm: **ADDPOLY( COEFF, POWER, LINK, POLY1, POLY2, SUMPOLY, AVAIL)**

This algorithm adds the two polynomials implemented using linear linked list and stores the sum in another linear linked list. POLY1 and POLY2 are the two variables that point to the starting nodes of the two polynomials.

Step 1: Set SUMPOLY:=AVAIL and AVAIL:=LINK[AVAIL]

Step 2: Repeat while POLY1  $\neq$  NULL and POLY2 $\neq$ NULL:

    If POWER[POLY1]>POWER[POLY2],then:

        Set COEFF[SUMPOLY]:=COEFF[POLY1]

        Set POWER[SUMPOLY]:=POWER[POLY1]

    Set POLY1:=LINK[POLY1]

        Set LINK[SUMPOLY]:=AVAIL and AVAIL:=LINK[AVAIL]

    Set SUMPOLY:=LINK[SUMPOLY]

    Else If POWER[POLY2]>POWER[POLY1], then:

        Set COEFF[SUMPOLY]:=COEFF[POLY2]

        Set POWER[SUMPOLY]:=POWER[POLY2]

    Set POLY2:=LINK[POLY2]

        Set LINK[SUMPOLY]:=AVAIL and AVAIL:=LINK[AVAIL]

    Set SUMPOLY:=LINK[SUMPOLY]

Else:

    Set COEFF[SUMPOLY]:=COEFF[POLY1]+COEFF[POLY2]

    Set POWER[SUMPOLY]:=POWER[POLY1]

    Set POLY1:=LINK[POLY1]

    Set POLY2=LINK[POLY2]

    Set LINK[SUMPOLY]:=AVAIL and AVAIL:=LINK[AVAIL]

    Set SUMPOLY:=LINK[SUMPOLY]

[End of If structure]

[End of Loop]

Step3: If POLY1=NULL , then:

    Repeat while POLY2 $\neq$ NULL

        Set COEFF[SUMPOLY]:=COEFF[POLY2]

        Set POWER[SUMPOLY]:=POWER[POLY2]

        Set POLY2:=LINK[POLY2]

```

        Set LINK[SUMPOLY]:=AVAIL and AVAIL:=LINK[AVAIL]
        Set SUMPOLY:=LINK[SUMPOLY]
    [End of Loop]
[End of If Structure]
Step 4: If POLY2=NULL, then:
    Repeat while POLY1≠NULL
    Set COEFF[SUMPOLY]:=COEFF[POLY1]
    Set POWER[SUMPOLY]:=POWER[POLY1]
    Set POLY1:=LINK[POLY1]
    Set LINK[SUMPOLY]:=AVAIL and AVAIL:=LINK[AVAIL]
    Set SUMPOLY:=LINK[SUMPOLY]
    [End of Loop]
[End of If Structure]
Step 5: Set LINK[SUMPOLY]:=NULL
        Set SUMPOLY:=LINK[SUMPOLY]
Step 6: Return

```

### **Multiplication of Polynomials using linear linked list representation for polynomials**

Algorithm: **MULPOLY( COEFF, POWER, LINK, POLY1, POLY2, PRODPOLY, AVAIL)**

This algorithm multiplies two polynomials implemented using linear linked list. POLY1 and POLY2 contain the addresses of starting nodes of two polynomials. The result of multiplication is stored in another linked list whose starting node is PRODPOLY.

```

Step 1: Set PRODPOLY:=AVAIL and AVAIL:=LINK[AVAIL]
        Set START:=PRODPOLY
Step 2: Repeat while POLY1 ≠ NULL
Step 3: Repeat while POLY2≠NULL:
    Set COEFF[PRODPOLY]:=COEFF[POLY1]*COEFF[POLY2]
        Set POWER[PRODPOLY]:=POWER[POLY1]+POWER[POLY2]
        Set POLY2:=LINK[POLY2]
        Set LINK[PRODPOLY]:=AVAIL and AVAIL:=LINK[AVAIL]
        Set PRODPOLY:=LINK[PRODPOLY]
    [End of step 4 loop]
    Set POLY1:=LINK[POLY1]
[End of step 3 loop]
Step 4 Set LINK[PRODPOLY]:=NULL and PRODPOLY:=LINK[PRODPOLY]
Step 5: Return

```

```

/* Program of polynomial addition and multiplication using linked list */
#include<stdio.h>
#include<stdlib.h>
struct node
{
    float coef;
    int expo;
    struct node *link;
};
struct node *create(struct node *);
struct node *insert_s(struct node *,float,int);
struct node *insert(struct node *,float,int);
void display(struct node *ptr);
void poly_add(struct node *,struct node *);
void poly_mult(struct node *,struct node *);
main( )
{
    struct node *start1=NULL,*start2=NULL;
    printf("Enter polynomial 1 :\n");
    start1=create(start1);
    printf("Enter polynomial 2 :\n");
    start2=create(start2);
    printf("Polynomial 1 is : ");
    display(start1);
    printf("Polynomial 2 is : ");
    display(start2);
    poly_add(start1, start2);
    poly_mult(start1, start2);
}/*End of main()*/
struct node *create(struct node *start)
{
    int i,n,ex;
    float co;
    printf("Enter the number of terms : ");
    scanf("%d",&n);
    for(i=1;i<=n;i++)

```

```

{
    printf("Enter coeficient for term %d : ",i);
    scanf("%f",&co);
    printf("Enter exponent for term %d : ",i);
    scanf("%d",&ex);
    start=insert_s(start,co,ex);
}
return start;
}/*End of create()*/
struct node *insert_s(struct node *start,float co,int ex)
{
    struct node *ptr,*tmp;
    tmp=(struct node *)malloc(sizeof(struct node));
    tmp->coef=co;
    tmp->expo=ex;
    /*list empty or exp greater than first one */
    if(start==NULL || ex > start->expo)
    {
        tmp->link=start;
        start=tmp;
    }
    else
    {
        ptr=start;
        while(ptr->link!=NULL && ptr->link->expo >= ex)
            ptr=ptr->link;
        tmp->link=ptr->link;
        ptr->link=tmp;
    }
    return start;
}/*End of insert()*/

struct node *insert(struct node *start,float co,int ex)
{
    struct node *ptr,*tmp;
    tmp=(struct node *)malloc(sizeof(struct node));

```

```

tmp->coef=co;
tmp->expo=ex;
/*If list is empty*/
if(start==NULL)
{
    tmp->link=start;
    start=tmp;
}
else /*Insert at the end of the list*/
{
    ptr=start;
    while(ptr->link!=NULL)
        ptr=ptr->link;
    tmp->link=ptr->link;
    ptr->link=tmp;
}
return start;
}/*End of insert()*/

void display(struct node *ptr)
{
    if(ptr==NULL)
    {
        printf("Zero polynomial\n");
        return;
    }
    while(ptr!=NULL)
    {
        printf("%.1fx^%d", ptr->coef,ptr->expo);
        ptr=ptr->link;
        if(ptr!=NULL)
            printf(" + ");
        else
            printf("\n");
    }
}/*End of display()*/

```

```

void poly_add(struct node *p1,struct node *p2)
{
    struct node *start3;
    start3=NULL;

    while(p1!=NULL && p2!=NULL)
    {
        if(p1->expo > p2->expo)
        {
            start3=insert(start3,p1->coef,p1->expo);
            p1=p1->link;
        }
        else if(p2->expo > p1->expo)
        {
            start3=insert(start3,p2->coef,p2->expo);
            p2=p2->link;
        }
        else if(p1->expo==p2->expo)
        {
            start3=insert(start3,p1->coef+p2->coef,p1->expo);
            p1=p1->link;
            p2=p2->link;
        }
    }
    /*if poly2 has finished and elements left in poly1*/
    while(p1!=NULL)
    {
        start3=insert(start3,p1->coef,p1->expo);
        p1=p1->link;
    }
    /*if poly1 has finished and elements left in poly2*/
    while(p2!=NULL)
    {
        start3=insert(start3,p2->coef,p2->expo);
        p2=p2->link;
    }
}

```

```

    printf("Added polynomial is : ");
    display(start3);
}/*End of poly_add() */

void poly_mult(struct node *p1, struct node *p2)
{
    struct node *start3;
    struct node *p2_beg = p2;
    start3=NULL;
    if(p1==NULL || p2==NULL)
    {
        printf("Multiplied polynomial is zero polynomial\n");
        return;
    }
    while(p1!=NULL)
    {
        p2=p2_beg;
        while(p2!=NULL)
        {
            start3=insert_s(start3,p1->coef*p2->coef,p1->expo+p2->expo);
            p2=p2->link;
        }
        p1=p1->link;
    }
    printf("Multiplied polynomial is : ");
    display(start3);
}/*End of poly_mult()*/

```

## 1.14 GENERALIZED LINKED LIST

A generalized linked list contains structures or elements with every one containing its own pointer. It's generalized if the list can have any deletions, insertions, and similar inserted effectively into it. A list which has other list in it as its sublists. Eg (5,(2,3,4),6,(1,2,3))

This list consists of two sublists i.e. (2,3,4) and (1,2,3) and main list consists of four nodes having 5, (2,3,4),6 and (1,2,3)