CHAPTER 1

Introduction to Data Structure

What is Data Structure?

The term "Data Structure" consists of two words i.e. "Data" and "Structure". By "data" we mean collection of facts and figures which is not in organized form, whereas by "structure" we mean arrangement or organization. So data structure can be defined as under:

"Data structure is a technique or way to arrange or organize the data in such a manner that it could be used/proceed effectively".

OR

"Data Structure is a systematic structure that can store the data in computer memory in an organized form".

The basic and well known data structures are array, stack, queue, tree and graph.

Why we study Data Structure? or importance of data structure:

Computer is a machine that works on data. To make computer more efficient in terms of speed and space, data should be stored in an organized way in the form of a data structure. Thus importance of data structure is:

I. To save computer memory.
II. To store the data in such a way so that it could be used efficiently and effectively.
III. Data structure causes better utilization of data and easy searching.
IV. Data Structure represents the relation between real word data in a logical way. So, Data Structure models the real word data very well.

Types of Data Structure:

Data structure can be categorized into the following two basic types.

a) *Linear Data Structure or List:*

The data structure where data items are stored/lying in a linear form is called linear data structure. It is also called list. Stack is a well known example of linear data structure.

b) *Non-Linear Data structure:*

The data structure where data items are not stored in a linear form is called non-linear data structure. Tree and Graph are the two examples of non-linear data structure.

Static Data Structure and Dynamic Data structure:

I. Static Data Structure: It is that data structure whose size/length is fixed after it is defined i.e. its size can not be changed later on. By length/size we mean the no. of items/elements a data structure can have. Example of static data structure is array. (we will discuss array later on)

II. Dynamic Data Structure: It is that data structure whose length may change after it is defined. So, dynamic data structure is more flexible than static data structure. Example of dynamic data structure is Linked List. (we will discuss Liked List later on)

Data: The collection of facts and figures which is not in organized form is called data. For example, the random list of students seeking admission in first year is data.

Data is in raw form, and hence on the basis of data we can not make any decision. Data may be about a touchable thing (e.g. chair) or it may be about an untouchable thing (e.g. bank account). Similarly, data may either be qualitative or it may be quantitative.

Qualitative data is that one which describes the quality of some thing. e.g. the qualitative data about a rose may be:

Operations on Data Structure:

The different actions or functions that can be performed over a data structure are called data structure operations. Following are the basic operations that can be performed on any data structure.

1: Insertion

The operation of adding a new item to a data structure is called Insertion.
For example if we have 3 elements in an array of size 5 and we want to add a new item to the array then this operation is called Insertion.

2: Deletion

The operation of removing an existing item from a data structure is called Deletion. In the above example if we want to remove item no. 4 from the array then the operation is called deletion.

3: Traversing

The operation of accessing all the items of a data structure is called traversing. Traversing means to meet with every item of data structure. Suppose if we want to list / display all the items of an array, then the action to access every item of the array is called traversing.

4: Searching

The operation of finding out an item in a data structure is called Searching. Suppose we have Roll numbers of fifty students in an array and we want to find out RNo. 36 in it, then this operation is called searching.

5: Sorting

The operation of arranging the elements/items of a data structure either in ascending or in descending order is called Sorting. Suppose we want to prepare a merit list for

120 students then we would have to arrange the marks of 120 students in descending order. This action/operation is called sorting.

6: Merging

The operation to combine two or more data structures into a single one is called Merging. Merging is also called concatenation. Suppose we have two arrays and we want to combine them to make one big array, the action/operation is called merging.

Algorithm

The step by step approach to solve a problem is called Algorithm. OR Algorithm is a systematic mechanism to solve a problem.

Algorithm is named after a Muslim scientist and Mathematician Ibni Musa Alkhowarzmi.

For example, suppose we want to reach to University of Peshawar from GDC Thana. We can solve this problem in steps as under;

- Walk on foot from GDC Thana to G.T road.
- Set in a vehicle (e.g. Flying coach or Coaster) to Peshawar.
- On reaching to Haji Camp Addah , get into wagon or bus.
- Drop near Khyber Teaching Hospital and enter into university of Peshawar.

Parts of Algorithm:

There are mainly two parts of an Algorithm which are given as follow.

I. Name and Introduction:

This is the first part of Algorithm Here, the name and task of algorithm is stated along with the variables used in the algorithm. Name of the algorithm and variables are usually

written in Capital. The name of Algorithm should be meaningful such as SUM or ADD for an algorithm of adding values.

II. Body of Algorithm or Steps:

This part of algorithm includes all actual steps of the algorithm. Each step should have a brief comment in pair of brackets over its purpose or function. Suppose a step in which we increase the value of variable should be commented as:

[Increment] or [increasing variable value].

It is good practice to label or number each and every step.

Language of Algorithm:

The language of algorithm should be in between computer language and human natural language. Remember that there are no hard and fast rules for the language of algorithm but it should not be totally English like statements. In other words the language and grammar of algorithm may be the combination of English, Math and artificial language.

For example, use '->' or '=' for assignment, INPUT or READ for inputting, PRINT for displaying/outputting, REPEAT for looping and EXIT for the termination of Algorithm.

Characteristics of Algorithm:

An algorithm must have the following characteristics.

I. Input: algorithm must take inputs. These inputs may be explicitly defined, assigned values or implicitly supposed to be read.

II. Output: It is the return value or values of the actual steps of an algorithm or it is the result of an algorithm.

III. Finiteness: This characteristics means that an algorithm must have finite number of steps. In other words there must be an end to the steps of algorithm.

IV. Effectiveness: It means that algorithm must be effective in terms of time and function. An algorithm must perform a best suitable function out of a number of possible choices, in comparatively short time.

V. Conciseness: An algorithm should be as brief and compact as possible. There should be no statement which is not needed and thus memory space should not be unnecessarily wasted.

Types of Algorithm: There are two types of algorithm.

1. ***Formal Algorithm***
2. ***Informal Algorithm***

1) Formal Algorithm:

That type of algorithm which is written under the basic rules, syntax and grammar of algorithm. Formal algorithm is always near to computer language syntax. Here frank and English like statements are avoided and Mathematical and artificial language style is followed. For example to do Input and output using formal algorithm, the steps will be,

1) [INPUTTING]
   READ a
2) [OUTPUTTING]
   PRINT a

2) Informal Algorithm:

Algorithm which is not formal is called informal algorithm. This type of algorithm is free from the hard and fast type syntax and rules. Statements are written in the form

of descriptive form of English sentences. For example the above input and output of formal style be written in informal as,

1) *Input a character*
2) *Display the value of a*

Analysis and complexity of Algorithm

From analysis of algorithm we mean to check and analyze each and every step of an algorithm. We do analysis to accomplish complexity. Complexity of algorithm means that an algorithm should do its job/task as quick as possible and should use less memory space. Steps must be concise, concrete and discrete to save time and space. You will study analysis of algorithm in higher classes in detail.

Some Examples of Algorithms:

Example No.1: Algorithm for adding two Numbers.

SUM (This algorithm will add two variables. X, Y and S are variables)

```
Step1: [ INPUT ]
       READ X
       READ Y
Step2: [ADDITION]
       Set S = X+Y
Step 3: [ OUTPUT]
       PRINT S
Step 4: [FINISH]
       Exit
```

Example No.2: Algorithm for Table of a number.

TABLE (This algorithm will display table of given number up to 16. n and i are variables.)

```
Step 1: [ INPUT ]
        READ n
Step 2: [INITIALIZATION]
        Set i = 1
Step 3: [LOOP]
        Repeat step4 and step5 While  i <=16
Step 4: [OUTPUT]
        PRINT n "x" i " = " , n*i
Step 5: [INCREMENT]
        i =  i+1
step 6: [FINISH]
        Exit
```

Example No.3: Algorithm for Searching in Array.

SEARCH (This algorithm will search an element x in array ARR. i is a variable and N is the last location of array.)

```
Step 1: [ INITIALIZATION]
        Set i = 0
Step 2: [ LOOP]
        While i<=n Repeat step 3 and step 4
Step3 [ CHECKING]
        IF( ARR[ i ] = X ) THEN
                PRINT "Item found at location " , i
        GOTO Step 5
Step 4: [ INCREMENT ]
        Set i = i+1
Step 5: [FINISH]
        Exit
```

Example No. 4: Algorithm for Sorting.

SORT(This algorithm will arrange the elements of an array ARR in Ascending order. i and Temp are variables while n is the last location of ARR.)

1) One-dimensional array:

That type of array whose elements are exceeding in only one dimension, is called one-dimensional array. It is also called linear array.

One-dimensional array is in vector form. The elements of a one-dimensional array can be accessed using a single subscript.

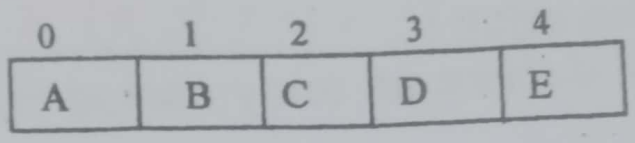


Fig (one-dimensional array of size 5)

2) Multi-dimensional array:

The type of array whose elements are exceeding in more than one dimensions, is called multi-dimensional array. The elements of multi-dimensional array require more than one subscript for accessing them.

An example of multi-dimensional array is a two-dimensional array. Two-dimensional array is in tabular form or matrix form. To access a particular element of a two-dimensional array we need two subscripts i.e. one subscript for row number and the other for column number.

0	1	2
A	B	C
D	E	F
G	H	I
J	K	L

Fig (Two-dimensional array of size 4x3)

Defining/declaring an Array:

Like other variables, array must be declared/defined before using it. However, the declaration of one-dimensional and two-dimensional array is slightly different.

a. Declaration of One-dimensional array:

Type array_name[size] ;

Where type may be int(for integer data), char(for character data), float(for foat data) etc.

Example:

int marks[6];

This statement will declare an array named 'marks' of type integer and size 6.

b. Declaration of Two-dimensional array:

Type array_name[m][n];

Where m is the number of rows and n is the number of columns.

Example:

float d[3][4] ;

This will declare an array d of type float and size 3x4.
(3X4 means having 3 rows and 4 columns.)

Representation of Two-dimensional array:

To store/represent a two-dimensional array in memory it should be converted into vector form. The following two methods are used to do so.

I. Row-Major order
II. Column-Major order

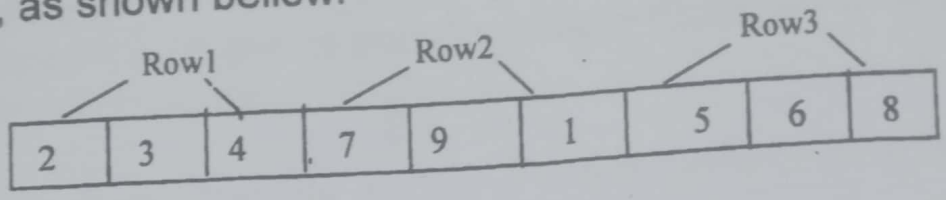
Suppose we have a Two-dimensional array with 3 rows and columns as shown bellow.

2	3	4
7	9	1
5	6	8

Now, to convert it into Vector form the two methods are as follows.

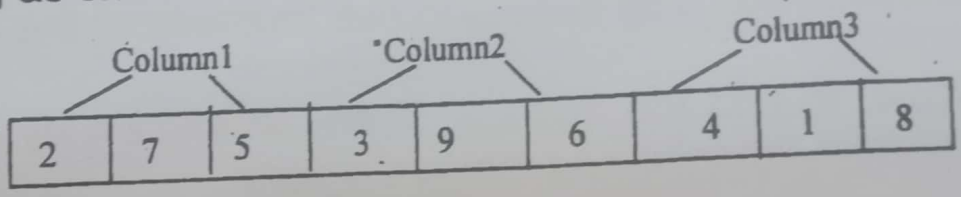
1) Row-Major Order:

Here a vector form is obtained by placing one row after another, as shown bellow.



2) Column-Major Order:

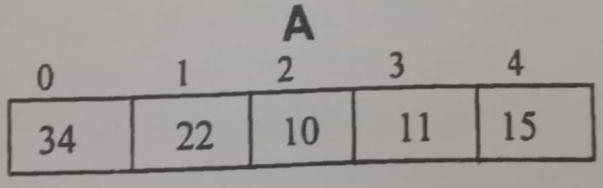
Here a vector form is obtained by placing one column after another, as shown bellow.



Accessing the elements/items of one-dimensional array:

In computer languages, array values/items are accessed by special variables called 'indexes' or 'subscripted variables'.

Consider the following one-dimensional array A of size 5.



Any of the five elements can be accessed by typing the name of the array followed by the subscript/position numbe of the particular element in the square brackets ([ ]). Not that in C language, the position number of the first elemer in every array is 0. The position number of the secon element is 1 and so on. Thus the first element of the array

is referred to as A[0], the second element of array A is referred to as A[1] and the nth element of array A is referred to as A[n-1].

Accessing one-dimensional array by Dope Vector method: Consider an array B of size as shown in the following figure.

B

1	2	3	4	5	6	7	8	9
22	34	44	63	59	66	84	11	48
200	201	202	203	204	205	206	207	208

Let the position numbers starts from 1 and ends at location number 10. The base address (address of the first location) is 200.

Now, to find the address of any location n by Dope Vector method the following formula is used;

$$Address(B[n]) = Base\ Address + w(n-1)$$

Where w is the no. of memory locations per word.

Suppose we want to find the address of memory location number 3 then;

$$Address(B[3]) = 200 + 1(3-1)$$
$$= 202$$

Accessing the elements of a 2-Dimensional array:

As discussed earlier there are two methods to represent a two dimensional array in memory. Therefore, there are two methods to access its elements.

a) In case of Row-Major Order Representation.

in Row-Major order the following formula is used to access or to find the address of elements of two dimensional array.

ML[i,j] = Base address + C(i-1) + (j-1)

Where
ML = Memory Location
i = the ith row
j = the jth column
Base Address = address of the 1st memory location
C = total no. of columns

For example if we have the following two-dimensional array.

2	3	4
7	9	1
4	6	8

The Row-Major order representation of this array is as under:

2	3	4	3	9	6	4	1	8
200	201	202	203	204	205	206	207	208

Let the base address of the array is 200. if we want to access value 9 i.e. element at 2nd row and 2nd column then the address of value 9 can be found as follow.

ML[i,j]= Base(A)+C(i-1)+(j-1)
ML[2,2]= 200 + 3(2-1) +(2-1)
ML[2,2]=200+3+1
ML[2,2]=204

Which is the address of value 9.

b) In case of Column-Major Order Representation:

The following formula is used to access elements of two dimensional array if it is stored by Column-Major Order.

ML[i,j] = Base address + (i-1) + r(j-1)

here ,

L = Memory Location

= the ith row

= the jth column

= total no. of Rows

or example if we have the following 2-Dimensional array.

2	3	4
7	9	1
5	6	8

he Column-Major Order representation of this array is nder:

2	3	4	3	9	6	4	1	8
300	301	302	303	304	305	306	307	308

et the base address of the array is 300. Suppose we want o access 1 i.e the element at 2nd Row and 3rd Column then he address of value 9 can be found as:

ML[i,j]= Base(A)+(i-1)+r(j-1)

ML[2,3]= 300 +(2-1) + 3(3-1)

ML[2,3]= 300+1+6

ML[2,3]=307

Which is the address of value 1.

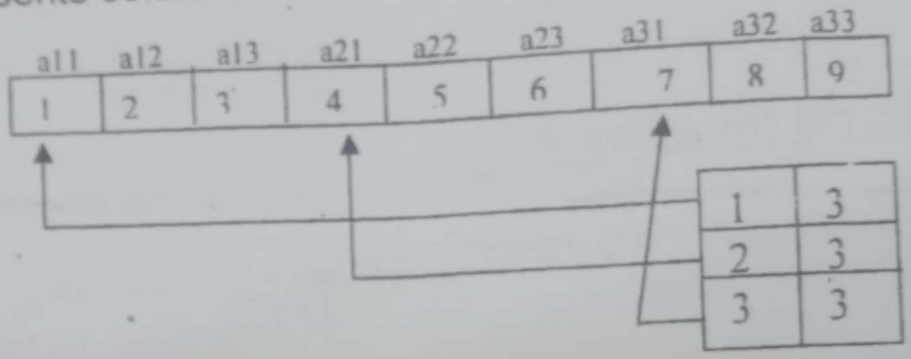
iffee Vector Method:

This is another method to access elements of ultidimensional array. Here a pointer table is created where he first pointer points to Row/Column number while the econd entry of the table represents the total umber of Rows in case of Column major order representation) or total umber of Columns (in case of Row major order epresentation).

Consider the following 3X3 array a.

1	2	3
4	5	6
7	8	9

Suppose the above array has been represented by Row-Major order. Liffee Vector Representation is given below. Note that in the subscripts (aij) , 'i' represents row while 'j' represents column.



Now any element of the given 3X3 can be easily accessed. Suppose we want to access A[2,3]. The first pointer will point to second Row and then after moving to third location element at 2nd Row and 3rd Column will be accessed which is '6'.

Access Table Method:

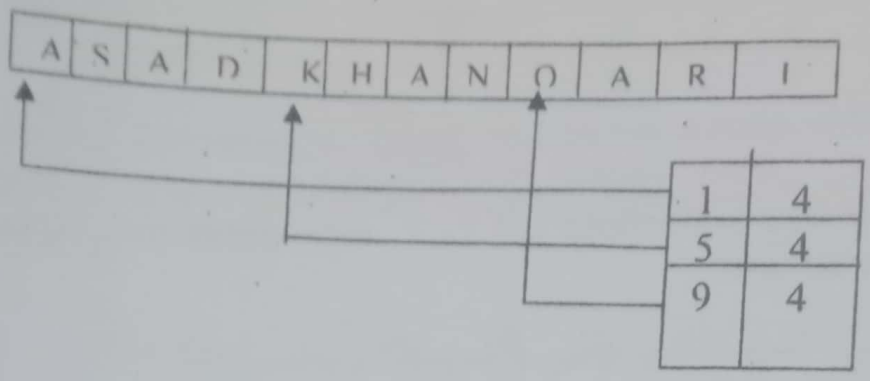
Liffee vector is suitable if array is of numeric data type. In case of array of characters or string, Liffee method is not efficient. In such cases Access Table method is used.

In this method elements are accessed with the help of two pointers, the first pointing to the starting location of string while the second represents the total number of characters in that string.

Suppose we have a character array of three Rows and four Columns as shown bellow.

A	S	A	D
K	H	A	N
Q	A	R	I

Now the Access Table representation is as bellow.



Algorithms for different operations in Array:

Following are some algorithms to perform some basic operations over array.

1) Insertion Algorithm:

Insertion is to add a new item into array. The algorithm for insertion is given below.

INSERT (This algorithm is to insert an item X at a location I array ARR. N is the last location of the array.)

```
Step1: [ OVERFLOW CHECKING ]
       If ( I > = N ) THEN
          PRINT "Overflow occurred" and goto step3
Step2 : [ INSERTION]
       Set  ARR[ I ] = X
Step3: [FINISH]
       Exit
```

Explanation:

In the first step we check overflow condition; if the array is already filled then overflow message will be printed and will jump to step3. In step2 we perform insertion at our desired

location I. Note that it is supposed that I is the location where we want to perform insertion .

2) Deletion Algorithm :

Deletion is the operation of removing an existing item from array. Here is the algorithm.

```
DEL ( This algorithm will delete an item X from location I.
ARR is the array , and N is the last location .)

Step1: [ CHECKING RANGE]
       If (I < 0  OR  I > N ) then
          PRINT " Out of Range" and goto step5
Step2 :Repeat step 3 and step 4 while I < = N
Step 3: [ OVERRIDING]
       Set  ARR[ I ] =  ARR [ I +1 ]
Step 4: [ INCREMENT]
       Set  I = I+1
Step 5: [FINISH]
       Exit
```

Explanation:

Step1 is to check that whether the location from which we want to delete an item exists or not. So if it is less than the first or greater than the last location then it would be out of range. In such case a message will be displayed and a jump is made to step 5. the step2 is to perform a loop. In step3 overriding occurs which replaces item at location no. by item at location no.I+1. Step4 is to increase I so that in the next iteration of the loop a new overriding is performed with new entries.

3) Searching Algorithm:

The process of finding out an element in a data structure is called searching. There are two ways/types of searching, each have a separate technique and separa

algorithm. The two common way/types of searching are as bellow.

a) Linear Search:

it is the normal and simple method of searching where each location of the array is checked sequentially. Algorithm for this type of searching is given below.

SEARCH (This algorithm will search an element X in array ARR. i is a variable and N is the last location of Array)

```
Step 1: [ INITIALIZATION]
        Set i = 0
Step 2: [ LOOP]
        Repeat step 3 and step 4 while i<=N
Step3: [ CHECKING]
        IF( ARR [ i] = X ) THEN
          PRINT " Item found at location no" , i
        GOTO Step 5
Step 4: [ INCREMENT ]
        Set i=i+1
Step 5: [FINISH]
        Exit
```

Explanation:

Step1 is to initialize loop variable i. In step2 we perform looping. In step3 we do checking that whether the item we want to search is present at location i or not. If it exists then 'item found' message will be printed and the algorithm will end. Step4 is to increase the loop variable I by 1.

b) Binary Search:

In this method of searching, array is divided into two sub-parts again and again till the required item is found. It should be noted that the array must be already in order form (Ascending or Descending). Following is the algorithm.

BISEARCH (This algorithm will find out an item X in the array ARR. E,S ,LB, UB and M are variables for End ,Start ,Lower Bound , Upper Bound and Middle respectively.

```
Step1: [ INITIALIZATION]
       Set S = LB, E = UB , and M = INT( ( S+E)/2)
Step 2: [ LOOP ]
       Repeat step 3 to 4 while S<=E and ARR[M] ≠ X
Step 3: if X < ARR[M] then
          Set E=M-1
       Else
          Set S=M+1
Step 4: [ FORMULA]
       Set M= INT ( S+E)/2)
Step 5: [ CHECKING]
       If ARR[M]= X then
          PRINT " ITEM FOUND"
Step 6: [ FINISH]
       Exit
```

Explanation:

Step1 is to initialize our variables. In step2 we have started a loop that will go on till S<=E and the item we wan to search i.e. X does not exist at location M. step3 is t check whether X is less than the item present at the middl position of the list or not. If it is less then it would means tha it is somewhere above the middle value as it has bee supposed that the list is already in sorted form (ascendin order) , so E or End will become one position up than th middle value. If it is not less then it would be mean that it somewhere below the middle value i.e. greater than t middle value so our E will the position one step after t middle value. In step5 we calculate the value of the midd by adding S and E dividing the result by 2 and th converting that into integer value. In step6 we check whet

our element is present at the middle position or not. if it is present then a found message will be printed.

4) Sorting algorithm:

The operation of arranging items of a data structure is called sorting. There are different types of sorting but the two well-known are given as follow.

a) Selection Sorting:

In this method of sorting a location selected and is compared with all the remaining locations. If the comparison obeys the desired order then no change will occur otherwise the elements of the location will swap. Following is the algorithm of selection sorting.

SELECTION_SORT (This algorithm will arrange the elements of an array ARR in Ascending order. i and Temp are variables while N is the last location of ARR.)

```
Step 1: [ INITIALIZATION]
        Set  i = 0
Step 2: [LOOP]
        Repeat Step 3 and Step 4 While i<N
Step 3: [ SORTING]
        IF ( ARR [i] > ARR[i+1] THEN
            [Interchange the elements at i and i+1]
            Temp = ARR[i]
            ARR[i]= ARR[i+1]
            ARR[i+1]=Temp
Step 4: [ INCREMENT]
        Set i = i+1
Step 5: [ FINISH]
        Exit
```

Explanation:

The algorithm is self explanatory except the third step in which we perform swapping i.e. placing the value of one location in other. If our desired order is obeyed then an item

at location i is saved in a temporary variable say Temp. The value at location i+1 is saved in location I and the value of variable Temp is saved location i+1. By this way we interchange the values of i and i+1 locations.

b) Bubble Sorting:

In this method of sorting an item is either bubbled up or down in each iteration. Here an element is selected and is compared with the next item. If the items are already in sorted form then no change occurs otherwise will swap. This process goes on till we reach the N-1 location i.e. the second last location.

BUBBLE_SORT (This algorithm will sort an array ARR in ascending order. i and J are variables and N is the last location.)

```
Step 1: [ outer Loop ]
        Repeat step 2 and 3 for J = 0 to N-1
Step 2: [ INITIALIZATION]
        Set i =1
Step 3: [ Inner Loop]
        Repeat while i < = N – J
        If ( ARR[ i ] > ARR[ i+1] Then
           Temp = ARR[i]
           ARR[i] = ARR[i+1]
           ARR[i+1] = Temp
           i = i +1
Step 4: [ FINISH]
           Exit
```

Explanation:

The algorithm starts from a loop that will go on till a variable j reaches to the second last location of the array. In step2 we initialize I. in third step we perform an inner loop and in the body of that inner loop we do checking for our desired order (ascending here). If the desired order obey

then no change otherwise the elements at i and i+1 locations will interchange.

Traversing Algorithm:

The operation of accessing each element of a data structure is called Traversing. The algorithm to traverse a linear array is given bellow.

TRAVERSE (This algorithm will traverse an array ARR. I is a variable and N is the last location of the array.)

```
Step 1: [ INITIALIZATION ]
        Set I =0
Step 2: [ LOOP ]
        Repeat step3 and step 4 while I <=N
Step 3: [ ACCESSING]
        Apply some operation on ARR[I]
Step 4: [INCREMENT]
        Set I = I+1
Step 5: [FINISH]
        Exit
```

Explanation:

The algorithm starts from initializing a loop variable I. in step2 we start our loop that will go on till the last element of the array. Step3 is to access the element of array at location I. Step4 is to go one step up i.e. to increase I so that we could access element of location no.I+1.

CHAPTER 3

Lists

What is List ?

It is a data structure where the data items are lying in a linear form. It is also called list.

Lists are further divided into two types;

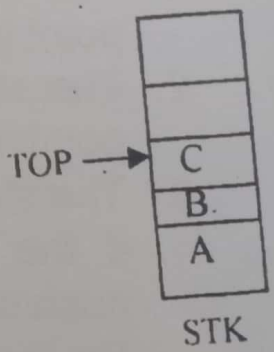
I. Sequential Lists
II. Linked Lists.

Sequential lists are the Lists whose data items are lying adjacent to each other. Common examples of Sequential Lists are Stack and Queue.

Linked Lists are those Lists whose elements are not lying adjacent to each other; rather they are connected/liked by pointers. Example of Linked List is a one-way Linked List.

Stack

It is a linear data structure in which the insertion and deletion takes place at only one end called the 'Top of the stack'. The item which is inserted first will be deleted in the last and vice versa. Therefore, a stack is also called FILO(First In-Last Out) list or LIFO(Last In-First Out) list.



The insertion of an item into a stack is called pushing where as the deletion of an item from a stack is called popping. Here a pointer called 'TOP' pointer is used that points to last value of the stack.

Suppose STK is a stack with five locations that contain three values A, B, C as shown in the fig.

Common Life example of Stack:

Common life examples of stack are:

1: A pile of plates in a hotel.
2: CD's placed over one another on a computer table forms a stack.

Why we use Stack?

Stack has many uses in computer sciences, for example,
1: Computer uses the logic of stack while evaluating mathematical expressions.
2: The philosophy of stack is used in the processing of nested loop. The outer loop which comes first is executed last while the innermost loop which comes last is fully is executed first.

Algorithms for Pushing and Popping

1) PUSHING:

Inserting a new item into stack is called pushing. It should be noted that before pushing a value into a stack we must first increase the 'TOP' pointer by 1 so that it could point to a new empty location.

If:

1. The stack is empty then the 'Top' pointer will point to null.
2. The stack contains only one value then the 'Top' pointer will point to first location.
3. The stack is full then Top will point to last location.
4. if the stack is full and we want to perform a push operation then the stack overflow will occur. So for push operation there must be an empty space in stack.

Following is the algorithm for pushing.

PUSH (This algorithm will insert a new item X in Stack STK. TOP is a pointer while N is the last location of STK.)

```
Step1:  [ CHECKING OVERFLOW]
        If ( TOP == N) Then
              PRINT " Stack Overflow" and goto step 4
Step2:  [ INCREMENT THE  TOP POINTER]
        Set TOP = TOP+1
Step3: [ INSERTION]
        Set STK[TOP] =  X
Step4: [FINISH]
        Exit.
```

2) POPPING:

Deleting an item from a Stack is called popping.
For popping there must be at least one value in the stack. We must check first whether the stack is empty or not. If the stack is empty and we want to perform a push operation then stack underflow will occur.

Algorithm for Popping is given below.

POPPING(This algorithm is to delete an item from a stack STK)

```
Step1: [ STACK UNDERFLOW]
       If ( TOP == NULL)
          PRINT  "Stack Underflow " and goto step 3
Step2: [DECREMENT THE TOP PONTER]
       Set TOP = TOP-1
Step3: [FINISH]
       Exit.
```

3) TRAVERSING:

Traversing is to access every element of a data structure. Here is an algorithm to traverse a Stack.

TRAVERSE (This algorithm will traverse a Stack STK. I is a variable, TOP is a pointer and N is the last location.)

Step1: [INITIALIZATION]
Set I = 0
Step1: [LOOP]
Repeat step2 and step3 while I <= TOP
Step2: [ACCESSING]
Access element at STK [I] and apply the operation
Step3: [INCREMENT]
Set I = I+1
Step4: [FINISH]
Exit.

NOTE: All other operations such as searching and sorting are almost the same for stack and array.

Evaluation of Postfix Expression by Stack:

A postfix expression is that expression where the operands are lying after their operands. For example, A + B is an infix expression whereas AB+ is a postfix expression. One application of stack is that we can evaluate a postfix expression by a 'Simulation Algorithm" given as follow.

Step1: [Adding sentinel to the expression]
Place a ")" symbol at the end of the expression
Step2 [SCANNING]
Read the expression from left to right and
Repeat Step3 and Step4.
Step3 [INSERTION]
If an operand is encountered, push that into Stack.
Step4: [OPERATION and INSERTION]
If an Operator say "X" is encountered then take the top two operands and apply "X" over them. Push the result into STACK.

Step5: [ASSIGNMENT]
Set RESULT = STACK[TOP]
Step6: [FINISH]
Exit.

Example:

Let's take a postfix expression which is to be evaluated by the above algorithm. Suppose the expression is :

9 4 7 - + 16 8 / +

Evaluation by stack.

Symbol Scanned	STACK
1) 9	9
2) 4	9 4
3) 7	9 4 7
4) -	9 -3
5) +	6
6) 16	6 16
7) 8	6 16 8
8) /	6 2
9) +	8

Conversion of Infix notation to Postfix by Stack.

Another application of Stack is that it is used to convert an Infix notation/expression into postfix form. Following is the algorithm to do so.

POST_FIX(This algorithm will convert an infix expression 'IN' into a postfix expression 'P'.)

Step1: [Insertion]
Push "(" into STACK. Place ")" at the end of IN.
Step2: [Scanning and Loop]
Scan the expression 'IN' from left to right and repeat step3 to step6.
Step3: [Forwarding]
IF an operand is encountered, forward it to expression 'P'.
Step4: [PUSHING]
if left parenthesis encountered, push it into STACK.
Step5: [TESTING AND PUSHING]
IF an operator say "X" is encountered then if the priority of "X" is equal or low than the operator already present in the STACK then repeatedly pop and add to P. Push "X" into STACK.
Step6: [TESTING OF RIGHT PARANTHESIS]
IF a right parenthesis is encountered then repeatedly Pop from STACK and add to P until left parenthesis is encountered. Remove the left parenthesis but not add it to P.
Step7: [EXIT]
Finish.

Example:

Convert the following infix expression into Postfix using Stack.

9 + (2 * 3 – (6 / 4 ^ 2) * 8) * 7

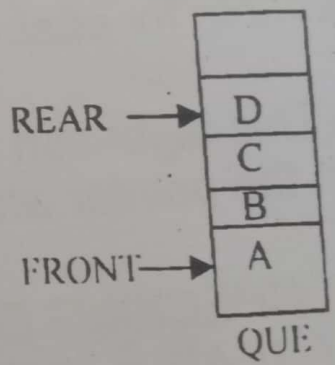
Solution:

Symbol Scanned	STACK	P (Postfix Expression)
9	(	9
+	(+	9
(	(+(	9
2	(+(	9 2
*	(+(*	9 2

3	(+(*	9 2 3
-	(+(-	9 2 3 *
(	(+(-(	9 2 3 *
6	(+(-(	9 2 3 * 6
/	(+(-(/	9 2 3 * 6
4	(+(-(/	9 2 3 * 6 4
^	(+(-(/^	9 2 3 * 6 4
2	(+(- (/^	9 2 3 * 6 4 2
)	(+(-	9 2 3 * 6 4 2 ^ /
*	(+(-*	9 2 3 * 6 4 2 ^ /
8	(+(-*	9 2 3 * 6 4 2 ^ / 8
)	(+	9 2 3 * 6 4 2 ^ / 8 * -
*	(+ *	9 2 3 * 6 4 2 ^ / 8 * -
7	(+ *	9 2 3 * 6 4 2 ^ / 8 * - 7
)		9 2 3 * 6 4 2 ^ / 8 * - 7 * +

Queue:

Queue is a linear data structure in which insertion takes place at one end where as deletion takes place at the other end. The end at which insertion takes place is called 'Rear end' while the end at which deletion takes place is called 'Front end'. Here is a simple Queue of size 5.



The item which is inserted first is deleted first and vice versa. Therefore, Queue is also called FIFO(First In-First Out) List or LILO(Last In-Last Out)List.

In Queue there are two pointers. The RAER pointer points to the REAR end and is used during insertion operation. The FRONT pointer points to the FRONT end and is used during Deletion operation.

Common Life Examples of Queue:

The best examples of Queue in our daily life are:

1. People awaiting in front of banks in a row to deposit there bills. Here, a man who comes first deposits his bill first.
2. People waiting for a train on a railway station. Here, a man who comes first gets into the train first.

Why we study Queue?

Computer follows the logic of Queue in so many situations. For example, printer makes a queue of the printing jobs and then prints those jobs one by one on the basis of that Queue.

Similarly, Queue is also used in the processing of programs by the processor/CPU.

Algorithms for operations on Queue:

Followings are the algorithms for some basic operations on Queue.

PUSHING:

Insertion in Queue is called pushing. Pushing is to add a new item in a Queue. It should be noted that before pushing a value into a Queue we must first increase the REAR pointer to point to a new empty location. Following is the algorithm for pushing into a stack.

PUSH (This algorithm will add a new item X in Queue QUE. REAR is the pointer while N is the last location.)

```
Step1:  [ CHECKING OVERFLOW]
     If ( REAR>= N) Then
            PRINT " Queue overflow" and goto step4
Step2:  [ INCREMENT  REAR pointer by 1]
     Set REAR = REAR+1
Step3: [ INSERTION]
     Set  QUE[REAR] = X
Step4: [FINISH]
     Exit.
```

2) POPPING:

Deletion from a Queue is also called popping. It is to remove an item from a Queue. Algorithm for Popping is given below.

POPPING (This algorithm is to remove an item from a Queue QUE. FRONT is the pointer.)

Step1: [CHECKING UNDERFLOW]
If (FRONT = NULL)
PRINT " Queue Underflow " and goto step 3
Step2: [INCREMENT FRONT pointer by 1]
Set FRONT = FRONT+1
Step3: [FINISH]
Exit.

3) TRAVERSING:

Traversing is to access every element of a da structure. Here is algorithm to traverse a Queue.

TRAVERSE (This algorithm will traverse a queue QU. I is variable, FRONT and REAR are the two pointers and N the last location.)

Step1: [INITIALIZATION]
Set I = FRONT
Step1: [LOOP]
Repeat step2 and step3 While I <=REAR
Step2: [ACCESSING]
Access element at QU [I] and apply operation
Step3: [INCREMENT]
Set I = I+1
Step4: [FINISH]
Exit.

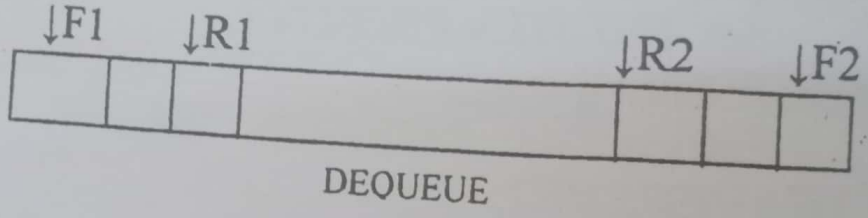
The logic of algorithms for other basic operations like searching and sorting etc in queue are the almost the same as that of array.

Types of Queue:

Following are the main types of queue.

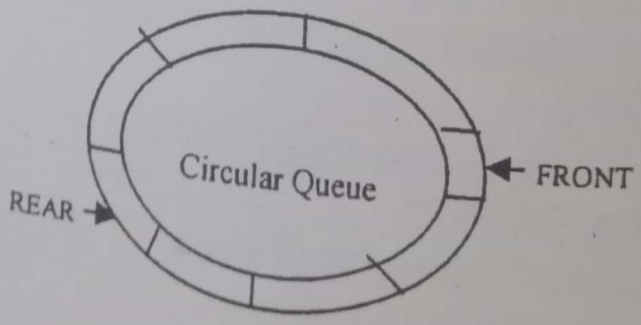
1) DEQUEUE:

DEQUEUE stands for Double Ended Queue. It is the Queue where insertion and deletion both can be performed at both ends of the Queue but not in the middle. There are four pointers in DEQUEUE. One REAR and FRONT pointers at the left end while one REAR and FRONT pointers at the right end of the DEQUEUE. DQUEUE can be traversed from both the sides.



2) Circular Queue:

The type of queue in which the last location is followed by the first location is called Circular Queue. In circular queue the two pointers i.e. REAR and FRONT are used but both the pointers move in a circular way and one may reach or even cross the position of the other.



Linked Lists

It is that type of linear data structure where nodes/items are connected with one another through pointers. Diagrammatically the connection between two nodes is represented by an arrow or arrows. These items/nodes are stored at random memory locations but the linear shape is given to it through pointers. The first node of the liked list is called the 'head' while the last node is called the 'tail'.

Types of Linked list:

Following are the three types of linked list.

1. One –way Linked Lists
2. Circular Liked Lists
3. Multiple Linked Lists

1) One-way Linked List:

It is a liked list where two adjacent nodes are connected by one pointer (represented by a single headed arrow). Here the preceding node contains the address of its succeeding node; however the succeeding node does not contain the address of its preceding node.

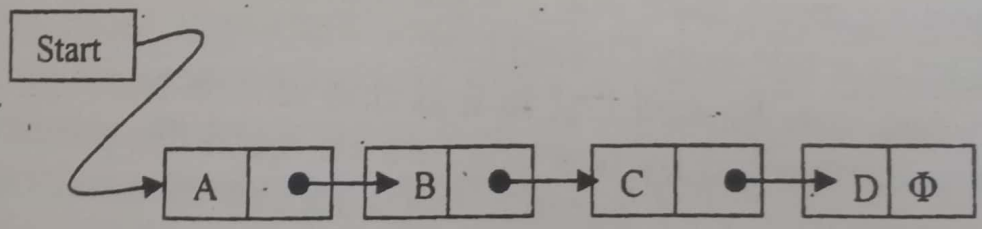


Fig (One-Way Linked List)

In a one-way linked list each node is divided into two parts:

I. INFO Part

II. ADDRESS Part or Pointer Part or Link part

The INFO part contains the actual data/information. The ADDRESS part contains the address of the next or succeeding node. If the ADDRESS part of a node A contains the address of another node B, then we say that the two

nodes are connected. We represent this connection by an arrow drawn from the ADDRESS part of node A toward node B.

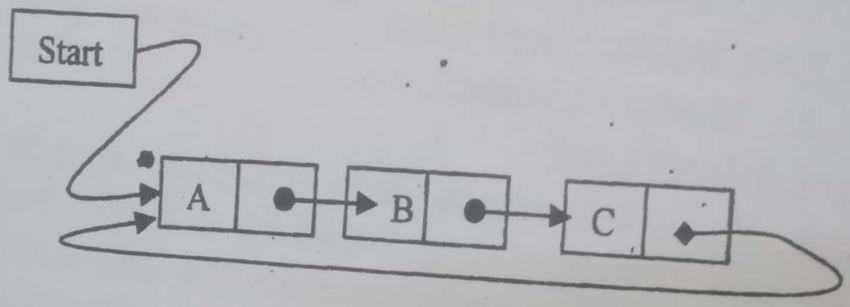
It may be noted that the ADDRESS part of the last node does not contain any address i.e. it contains null address represented by symbol Φ or X.

Here, a pointer called "START" pointer is also used that contains the address of the first/head node i.e. "START" is a pointer to the first node.

A one-way liked list can be traversed form one side/end only.

2) Circular Linked List:

It is linked list where the last node is connected back to the first node i.e. the ADDRESS part of the last node contains the address of the first node.



3) Multiple Linked List:

It is that type of Linked List where nodes are connected by more than one pointer. An example of Multiple Linked List is a two-way Linked List.

A two-way Linked List is that one where two adjacent nodes are connected by two pointers i.e. both the preceding and succeeding nodes contain the address of each other. The connection is represented by a double headed arrow. Here each node has three parts.

1) INFO Part: This Part contains the actual data
2) LEFT Part: This part contains the address of preceding node.
3) RIGHT Part: This part contains the address of succeeding node.

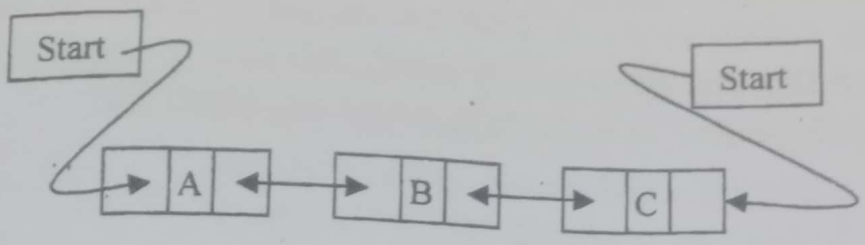


Fig (Two-way Liked List)

Note that the node at the left end contains Φ or x in its LEFT part; whereas the node at the right end contains Φ or x in its RIGHT part to represent null address.

A two-way liked list can be traversed from either sides/ends.

Why we use Linked List ? or importance of Linked List:

Linked list has many important applications in computer sciences. For example, one limitation of Array is that while doing insertion or deletion in a particular position the entire array has to be shuffled up or down which is a difficult job. To avoid this difficulty Linked list is used.

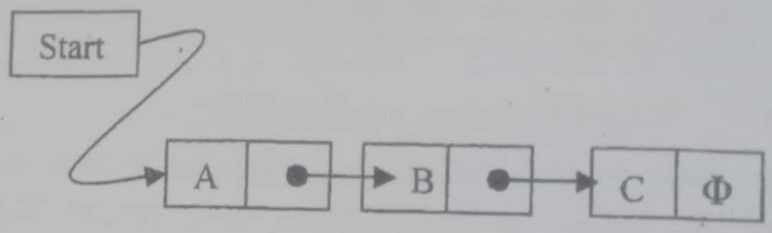
Another limitation of array is that it is static in nature i.e. when we define an array of any length, then we can not store more items than that size. However, liked list is dynamic in nature i.e. we can store as many information/items as we wish in a liked list.

The most important application of Linked list is the creation of files on secondary storage devices. Operating system creates and manages files on hard disk in terms of linked list where each node represents a small manageable block called Cluster i.e. the Clusters of a file are linked with each other like the nodes of linked list. Last cluster/node has null address from which Operating System knows that it is the last cluster/node of the file.

Memory representation of Linked List

A Linked List (one-way linked list) is represented in memory with the help of two arrays. One array named 'INFO'

for storing actual data while the other array named 'LINK' for storing addresses. A pointer named 'Start' is used to point to the first location of the array INFO.
Suppose we have the following one-way liked list:



The above liked list can be represented in memory as under.

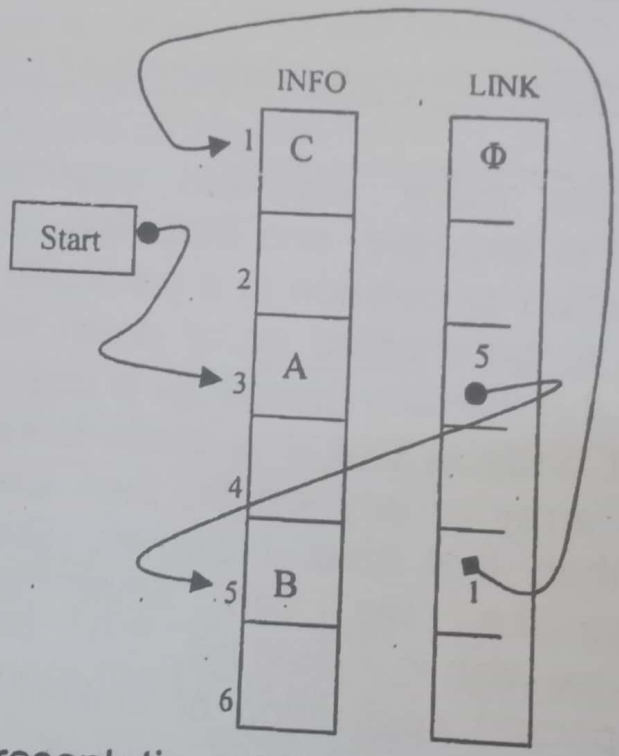


Fig (Representation of One-way Linked List in memory)

Algorithms of Linked List:

Before writing algorithms of Linked list, note the following abbreviation used in the coming algorithms.

Ptr = Pointer
LOC = Location
PLOC = Previous Location
NULL = Null means void or empty.
LINK = It means Link of a pointer i.e. Address part.

START = A pointer pointing to the first node of a linked list.

AVAIL = It is a pointer pointing to the first node of AVAIL list. AVAIL list is an imaginary list of empty nodes. Note that we take a node from AVAIL list during insertion or put a node to AVAIL list during deletion.

1) Traversing a Linked List:

Traversing is to access each and every element of a data structure. The following algorithm will traverse a linked list.

TRAVERSE (This algorithm will traverse a one-way linked list.)

Step 1: [INITIALIZATION]
Set Ptr = START
Step 2: [LOOP]
Repeat step 3 and Step 4 Until Ptr = NULL
Step 3: [TRAVERSE]
Perform some processing on INFO[Ptr]
Step 4: [MOVE]
Set Ptr = LINK[Ptr]
Step 5: [FINISH]
Exit

Explanation:

In Stpe1 we have initialized our pointer Ptr to the START pointer. Step2 is Loop which will repeat Step3 and Step4 till the last Node i.e. when Ptr becomes NULL. In Step3 we do some action on the INFO part of the node to which Ptr will point. Step4 is to move the pointer Ptr to next node. Last step is to finish and exit.

2) Insertion in Linked List:

Insertion is to add a new element into a Data structure.

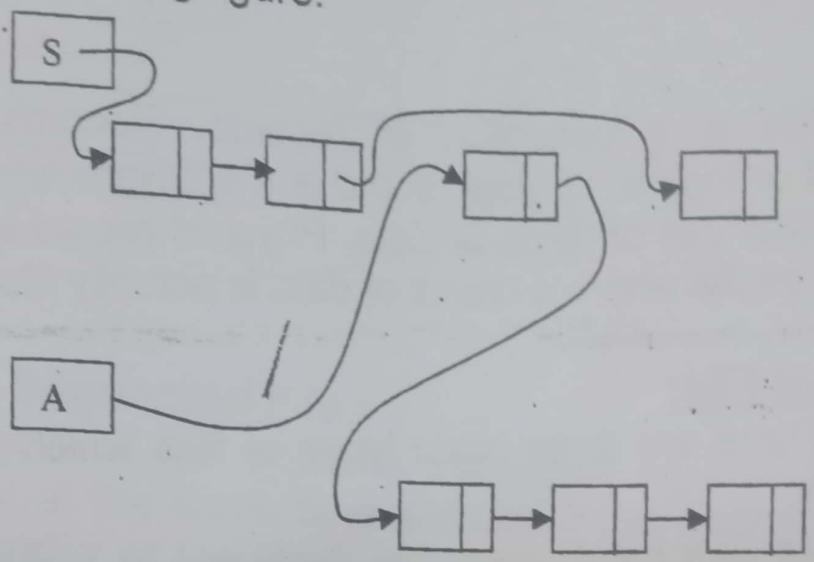
INSERT (This algorithm will add a new item X into a one-way linked list.)

```
Step 1: [OVERFLOW CHECKING]
        If AVAIL = NULL then
            PRINT "Overflow occurs" and goto step5
Step 2: [ GET A NEW NODE]
        Set NEW = AVAIL and AVAIL = LINK[AVAIL]
Step 3: [ ASSINGING VALUE]
        Set INFO[NEW] = X
Step 4: [ INSERTION]
        Set LINK[NEW]=LINK [LOC] and LINK [LOC]=NEW
Step 5: [FINISH]
        Exit
```

Explanation:

The first step is to check whether there is any node available in the AVAILABLE list. AVAIL is an abstract list from where we will get a new node and will add that into our list. Step2 is to get that NEW node form AVAILABLE. It should be noted that will get the first node from AVAILABLE list and now the AVAIL pointer is made such that to point to the following (2nd node i.e. the node to which NEW was pointing.) by the statement " AVAIL=LINK[AVAIL]". In step3 we make the value X , which we want to insert as the INFO part of the NEW node. With step4 we insert the NEW node into the list at a location say LOC. To connect the NEW node first the LINK of LOC will point to NEW node and the LINK of NEW will point to the node to which LOC was previously pointing i.e. LINK[NEW] = LINK[LOC]. All this is clear from the diagram shown on next page.

PLOC is the previous node of a node which is to be deleted. PLOC should be made such that to point to the node to which LOC is pointing. So node LOC will be deleted as shown in the fig. Now we have to add this node to the AVAILABLE list of empty nodes. We will assign LOC as first node to the AVAILABLE list by the statement "LINK[LOC] = AVAIL" and "AVAIL = LOC". All this is clear from the following figure.



Where:

S = Start pointer

A = AVAIL pointer

4) Searching in Linked List:

The process of finding out a data item in a data structure is called searching.

FIND (This algorithm will find out a node in liked list whose INFO part has value X.)

Step 1: [CHECKING UNDERFLOW]

If START = NULL Then PRINT "empty " and goto Step6

Step 2: [INITIALIZATION]

Set Ptr = START

```
Step3: [LOOP]
    Repeat step2 and step3 while LINK[N] ≠ NULL
Step4 : [ SEARCHING]
    If  INFO[Ptr] =  X Then Write Item found and goto step6
Step5: [ LINK TO OTHER]
    Set Ptr = LINK[Ptr]
Step 6: [FINISH]
      Exit
```

File:

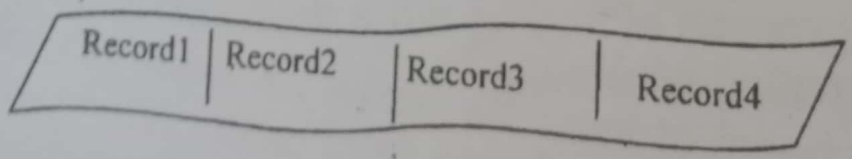
File is a collection of records. Records may be physical or logical. Logical record is a software term referred to the collection of fields or data. Physical record is a unit of storage media where a chunk of data is actually stored.

Types of File:

There are three basic types of files which are given as follow.

1) Sequential File:

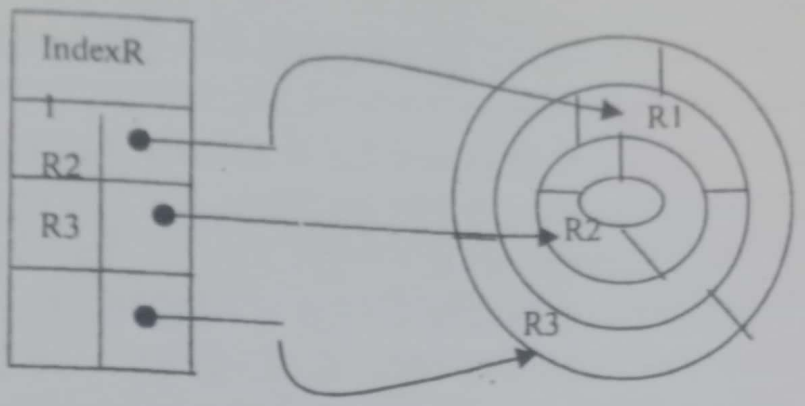
The type of file where records are stored one after another in a sequence is called Sequential file. Records of sequential files can be traversed successively. Random access of records of such files is impossible. A good example of sequential file is the VCR or Tape cassette. If you want to play song number 8 then you must traverse from song no.1 to 8 and you can not directly play song number 8.



2) Index File:

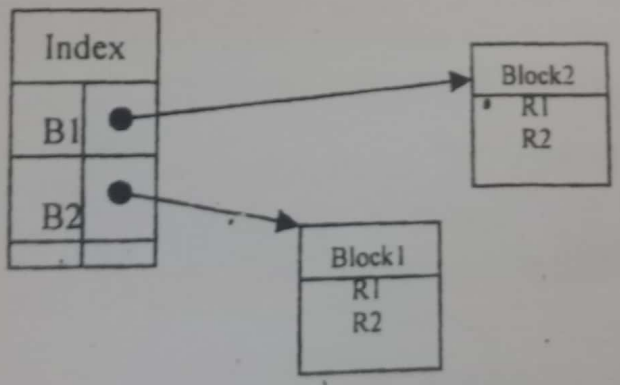
The type of file where records are stored and accessed based on an index is called Index file. The records of such files can be accessed randomly. The records of Index files are scatteredly stored on mass storage and are retrieved with the help of their entries from the Index

of the file. There exist the linkage of pointer of the index entry and the address of the record as shown in the following figure.



3) Index Sequential File :

The type of file which is sequential as well as non-sequential at the same time is called Index-Sequential file. Here an index of the block of records is created but inside the block records are stored sequentially. Such files have the characteristics of both index and sequential files. A typical view is given below.



Hashing and Hash Function:

The technique used in Filing where a function is used to find out address for a record to be saved (or retrieved from) is called Hashing. The function used is called Hash function. Hash function gets a key value and find out the

corresponding address for the key. If K is the key and A is the address then mathematically,

$$H: K \rightarrow A$$

Types of Hashing:

The three basic types of Hashing are given as follow.

1) Division Method:

In this method Key 'K' is divided by a prime number say 'n' and the remainder is used as the address for the record. For example if we have assigned a 4-digit key to the employees of education department and we use (prime)number 47 for our hashing then the then the record of employee 2345 will be stored at 42 as 2345 divide 47 and the remainder will be 42.

2) Mid-square Method:

In this method the square of Key 'K' is taken and then the mid one or more digits are taken as address. For example if we take the above example of 2345, with mid-square method we will proceed as,

2345X2345 = 5499025

Here deleting 3 digits from left and right we will get '9' as the result.

3) Folding Method:

In folding method the key 'K' is divided into two parts. The parts are added and the result is taken as the address. For example if again we take the same Key 2345 the result will be,

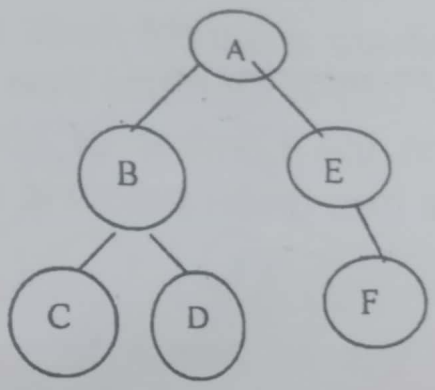
2345 => 23, 45 => 23+45 = 68

So '68' is taken as the address.

TREES

What is Tree?

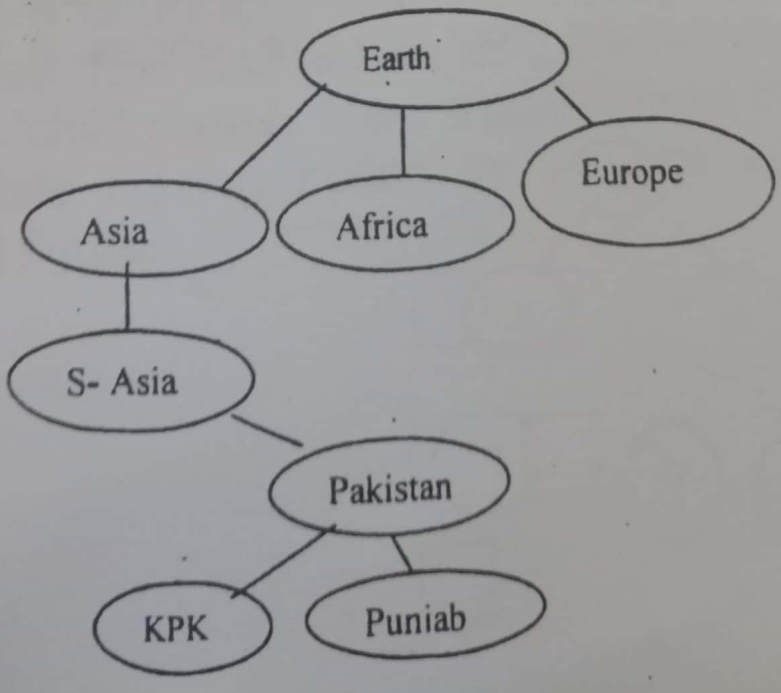
Tree is a non-linear data structure where nodes are liked with each other in a hierarchical manner. A simple Tree is given below.



Why Tree is used?

In general life we some time come across with a hierarchical relationships of data (for example relationship of family members). In such cases Tree is used to represents such relationships.

Lets take the example of earth, where different continents and countries can be represented by Tree as shown bellow.



Importance of Tree in Computer Science:

Tree is a very important concept in computer sciences. Some applications of tree are given below.

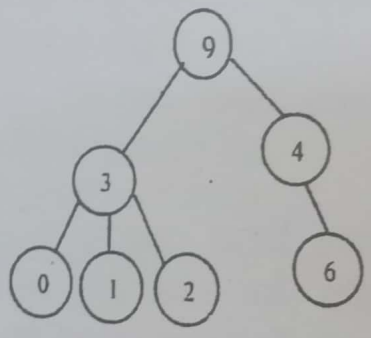
- Evaluation of a mathematical expression.
- Conversion of one expression into another.
- Sorting (using heap-sort).
- Searching and finding path in AI.

Types of Tree:

There are two basic types of tree.

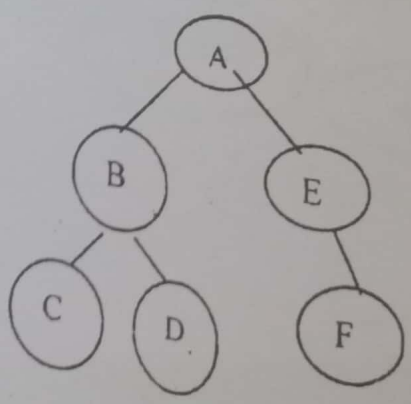
1) General Tree:

The type of tree where a parent node might have any number of child nodes is called General tree. In such a tree there is no restriction on the number of child nodes of a parent. For example, the tree given below is a General Tree.



2) Binary Tree:

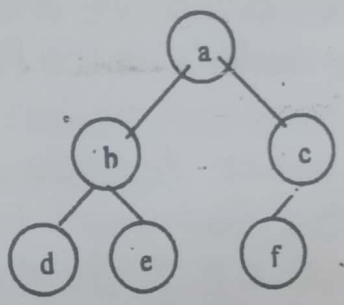
The type of tree where a node might have zero, one or two child nodes is called Binary tree. In Binary tree a parent node can have at the most two child nodes. A simple Binary Tree is given as under.



Notice that in the previous figure node A has two children(B,E), node B has two children(C,D), node E has one child(F) , and nodes C,D,F has no child. Therefore, it is a Binary Tree.
There are further two types of Binary Tree.

a) Complete Binary Tree:

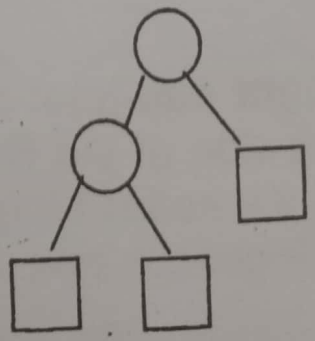
It is that Binary tree where all the levels except the last level have the maximum number of nodes, and all the nodes at the last level appear as left as possible.
For example the following Tree is a complete Binary Tree.



The maximum no. of nodes at any given level 'r' can be obtained by formula 2^r. The upper most level is called level 0, the next upper level is called level 1 and so on. So, for r=0 the no. of nodes will be $2^0 = 1$, for r = 1 the no. of nodes will be $2^1 = 2$.

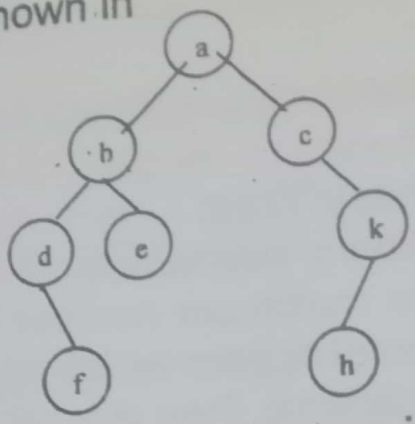
b) Extended Binary Tree or 2 Tree:

It is that Binary tree in which each parent node have either zero or exactly two child nodes is called Extended Binary tree. It is also known as 2-tree as each parent node have two child nodes. Here a parent node is represented by circle while leaving nodes by squares as shown in the following figure.



Tree Terminologies:

Consider the Tree as shown in figure.



I. Root node: The node at the top the tree is called the Root node. In the given Binary Tree node 'a' is the Root node.

II. Parent node or Predecessor: A node is said to be Parent node if it is connected downward to some other node(s) i.e. it has childe node(s). e.g. In the given Tree node b is the parent of nodes d and e.

III. Child node or Successor: A node is said to be child node if it is connected upward to a node i.e. it has a parent node. e.g. In the given Tree nodes d and e are the child nodes of node b. however node b is the child of node a.

IV. Leaf nodes: A node is said to be Leaf node if it has no child node. e.g. in the given Tree nodes f, e, h are Leaf nodes.

V. Brothers: The nodes which have the same Parent node are called Brothers of each other. In the given Tree nodes d and e are Brothers.

VI. Left sub-tree and Right sub-tree: In a Binary Tree all the nodes lying at the left side of the Root node forms the Left sub-tree, whereas all the nodes lying at the right side of the Root node forms the Right sub-tree. In the given Binary

Tree nodes b, d, e, f forms the Left sub-tree while the nodes c, k, h forms the Right sub-tree.

VII. Edge: The line connecting a parent node to its child node(s) is called Edge.

VIII. Path: The sequence of consecutive edges is called Path.

IX. Branch: A path ending in a Leaf node is called Branch.

X. Height or Depth of Tree: The no. of nodes in the longest Branch of a Tree is called Depth or Height of the Tree. e.g. the Depth of the given Tree is 4.

XI. Similar Trees: Two trees are said to be similar if they have the same shape.

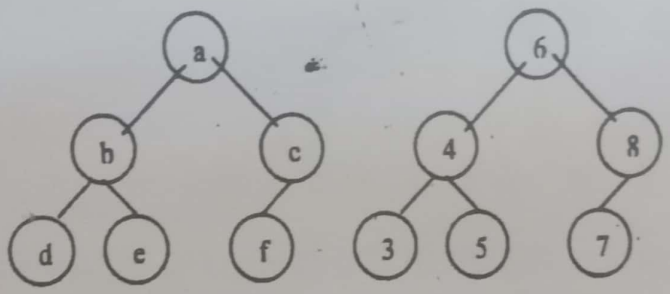


Fig (Similar Trees)

XII. Copy of a Tree: Two Trees are said to be copies of each other if they have the same shape and have the same nodes at the corresponding positions.

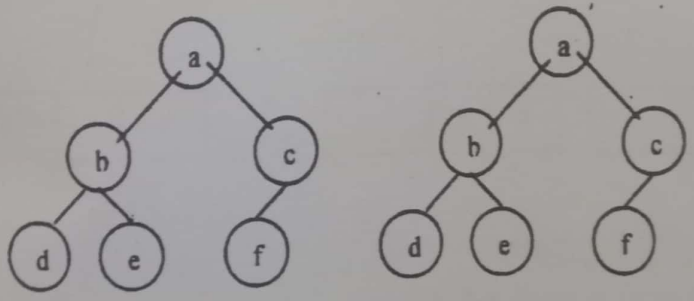


Fig (Copy of Tree)

Computer Representation of a Tree (Binary tree):

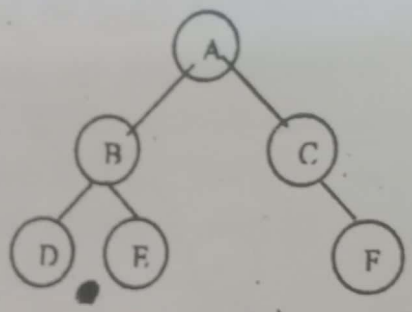
There are two ways by which a Binary Tree is represented in computer memory.

a. Linked List representation
b. Sequential representation.

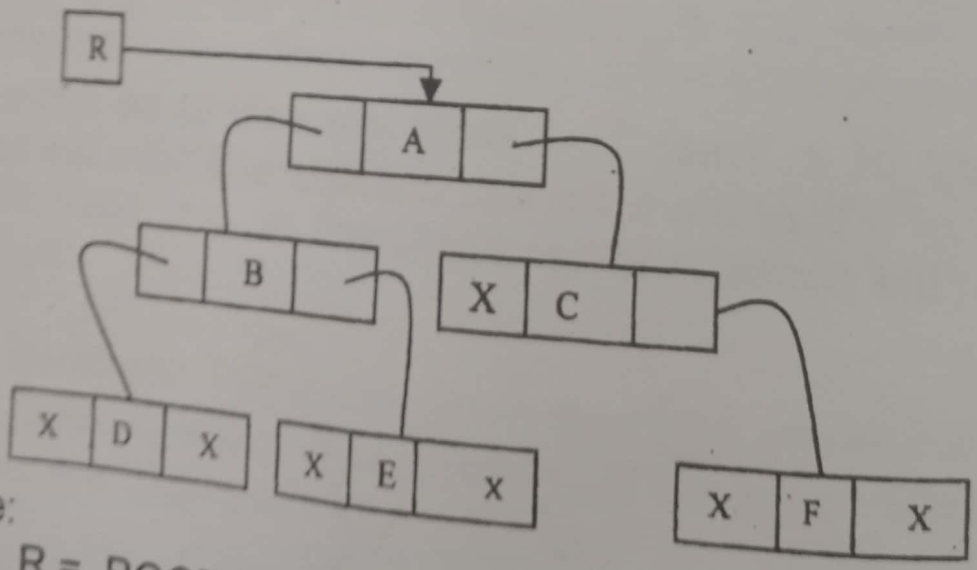
a) Linked List Representation of Tree:

In this method three arrays namely INFO, LEFT and RIGHT are used that represent Information parts, left children and right children respectively. There are two Pointers named ROOT and AVAIL pointing to the beginning of the Tree (Root node) and AVAIL list respectively.

Consider the following Binary Tree.



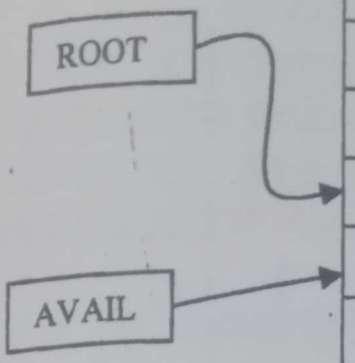
The above Tree can be drawn in the following figure also.



Where:

R = ROOT pointer
And X shows null address

Now the Liked List representation of this tree will be as shown on next page.



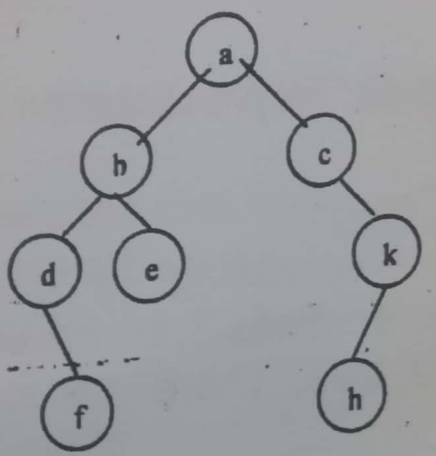
	INFO	LEFT	RIGHT
15	C	Φ	24
16			
17	A	19	15
18			
19	B	20	22
20	D	Φ	Φ
21			
22	E	Φ	Φ
23			
24	F	Φ	Φ

b) Sequential Representation of Tree:

In this method a Binary tree is represented by a single linear array with name TREE. The array will be formed as

I. The root node of tree R is stored in TREE[1].
II. If a parent node N is at kth location of the array then its left child will be saved at TREE[2*k] and its right child at TREE[2*k+1]

Consider the following Binary Tree.



This Tree can be represented in memory by sequential representation as shown in figure.

Now you can see that 'a' is the root Node of the tree, so it is stored at 1st location. Its left child is 'b' and is therefore stored at 2*1 i.e. at 2nd location. Similarly 'c' is the right child of the root node, therefore stored at 3rd location. The same process is repeated for all the parent nodes and their children.

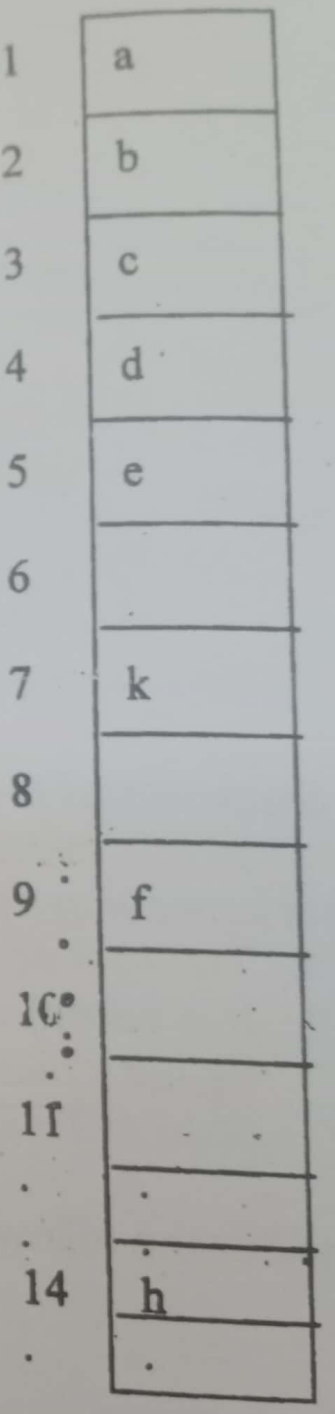


Fig (Sequential Representation of Binary Tree)

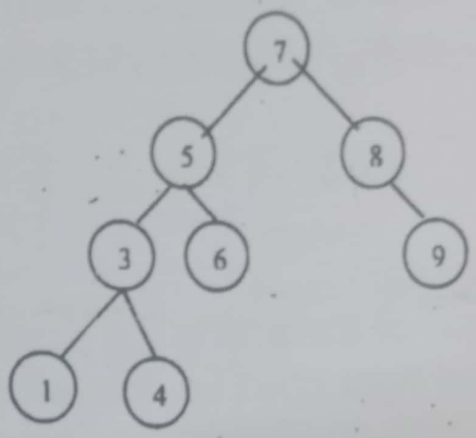
Traversing a Binary Tree:

Traversing is the accessing of each and every element of a data structure. Thus Tree traversing means to access every node of a tree. There are four methods to traverse a Binary tree.

1) Level-By-Level Traversal:

In this method the nodes are traversed level after level. Here, traversing will be started from level 0 (zeroth level) and will be proceed sequentially, accessing each node from left-to-right at every level. It should be noted that the first level (in which Root node lies) is called Level Zero.

Consider the following Binary Tree.



The Level-By-Level Traversal of the above Binary Tree is:

7 5 8 3 6 9 1 4

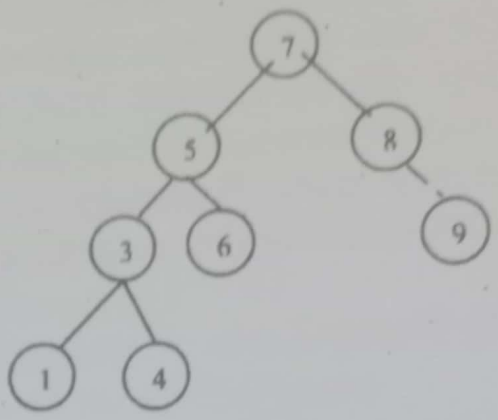
2) Pre-Order Traversal:

In this method the parent node is accessed before its children, and the left child is accessed before the right child. However the children of a node N are accessed before the brother of node N.

For pre-order traversal follow the following steps:

I. Access the root node
II. Traverse the left sub-tree by pre-order
III. Traverse the right sub-tree by pre-order

Consider the following binary Tree.



The Pre-Order Traversing of the above Binary Tree is:

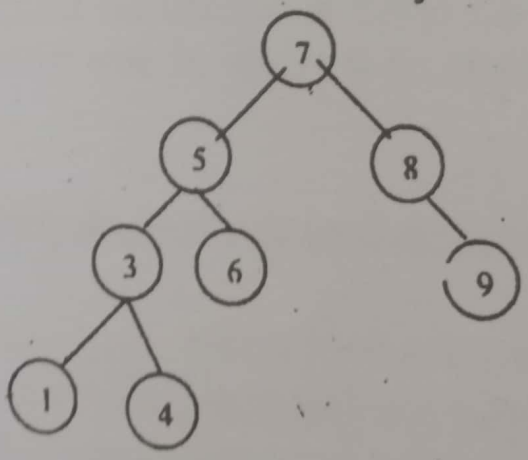
7 5 3 1 4 6 8 9

3) In-Order Traversal:

In this method of traversing the parent is accessed in between the left and right child i.e. the left child is accessed first, then the right child and finally the right child is accessed. For in-Order traversal follow the following steps.

I. Traverse the left sub-tree by in-order
II. Access the root node
III. Traverse the right sub-tree by in-order

Lets again take the same Binary tree.



The In-order Traversal is:

1 3 4 5 6 7 8 9

Constructing a Binary Tree from arithmetic Expression:

Computer evaluates expression using the logic of binary tree. For this purpose Binary tree is constructed from mathematical expressions. Before listing the steps to form a Binary tree, it is wise to note the priorities of operators on the basis of which computer evaluates mathematical expression.

The following table shows the priority of operators starting from highest priority and moving down to lowest.

Operator	Symbol	Priority
Parenthesis	( )	Highest
Power	^ or ↑	Next highest
Multiplication and Division	* and /	
Addition and Subtraction	+ and -	Lowest

Note: It should be noted that if two or more operators of the same priority occurs then one at the leftmost side has the higher priority and is processed first. For example * and / have the same priority but one that occurs to the left has high priority. Same is the case with + and -.

The following steps should be followed to construct a Binary Tree from mathematical/arithmetic expression.

- Take the operator having lowest priority in the given expression
- Make the scanned operator as the Parent node.
- Step2 will divide the expression in two parts i.e. left and right. The left will become the left child and right as the right child.

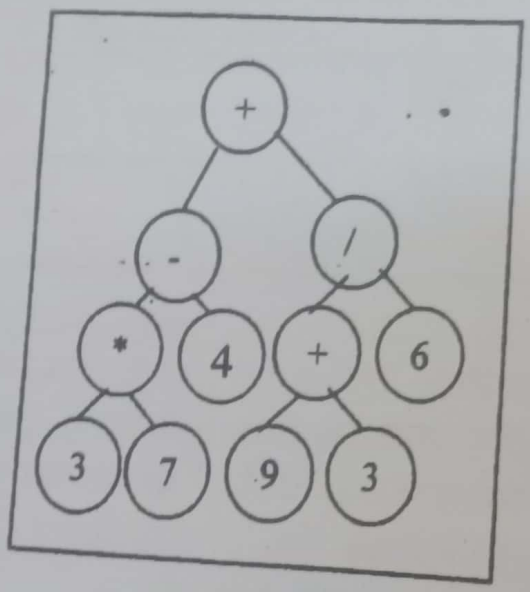
- Take one side of the expression at a time and repeat step1 to step2.
- When one side of expression is completed take the other side.
- Repeat the above steps till the entire expression is converted into Binary tree.

Example:

Consider the following expression.

3 * 7 – 4 + (9+3) / 6

The Binary Tree of the given expression will be as under.



Constructing a Binary Tree from given values:

We can also construct a binary tree from values. Use the following steps to do so.

- Take the first value and make it the root node of the tree.
- Repeat step3 to step 5 until the last value is taken.
- Take the next value. Compare it with the parent node.
- If the value is less than the parent node then make it as the left child of the parent node.

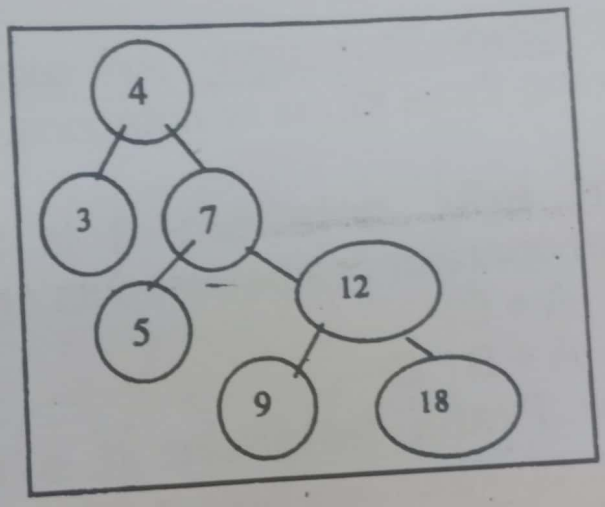
- If the value is greater than or equal to the parent node then make it as the right child.

Example:

Construct Binary Tree for the following values.

4 7 12 9 5 3 18

The Binary Tree for these values will be as under.



CHAPTER 5

SIMPLE SYNTAX ANALYSIS

Forms of Expression/Notation: An expression may appear in the following three forms or notations.

1. Infix form or infix notation: It is the form of expression where an operator lies between its operands.

Examples: I. A + B
II. A + B * C

2. Prefix form or Prefix notation: It is the form of expression where an operator lies before its operands. It is also called Polish notation.

Example: +AB

3. Postfix form or Postfix notation: it is the form of expression where an operator lies after its operands. It is also called Reverse Polish notation.

Example: AB+

Conversion of Expression from one form to another:

There are two methods to convert a given expression from one form to another.

I. Hand inspection method
II. Binary Tree method
III. Stack method (for infix to postfix)

Conversion by Hand Inspection method:

In this method we use the following instructions to convert a mathematical expression from one form to another e. without using Binary tree.

Now traverse the Binary Tree by Preorder;

+ - * 3 7 4 / + 9 3 6

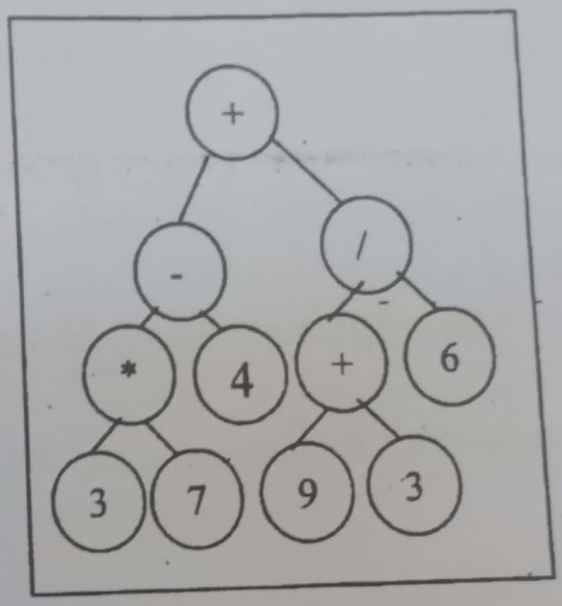
which is the required Prefix expression.

Example2: Convert the following infix expression Postfix.

3 * 7 – 4 + (9+3) / 6

Solution:

The Binary Tree of the given expression will be as under.



Now traverse the Binary Tree by Post-order;

3 7 * 4 – 9 3+6 / +

which is the required Postfix expression.

III. Conversion using stack (Stack method):

This method is specially used to convert an infix expression to postfix. (This method is explained at page no. 261).

CHAPTER 6

GRAPHS

What is Graph?

A graph is a non-linear data structure containing nodes connected by lines/arrows, but not in a hierarchical form.

The nodes in a graph are also called vertices and the lines connecting two nodes are also called edges or arcs. If V is the set of vertices and E is the set of edges then a graph G is mathematically defined as;

$$G = (V, E)$$

Note: The set of vertices and edges must be finite.

Examples: The figures given bellow are graphs.

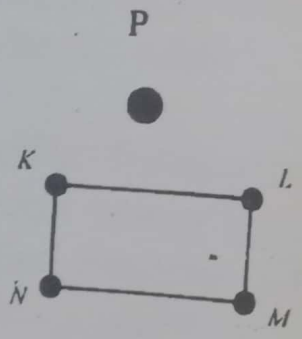


Fig 1

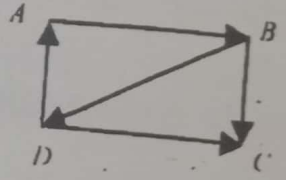


Fig2

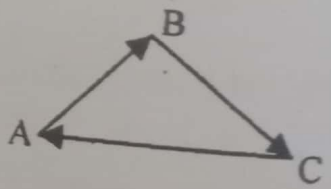


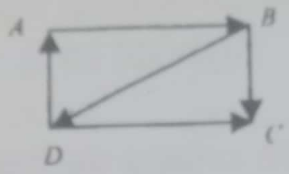
Fig3

Adjacent Nodes:

Two nodes connected by an edge are called adjacent nodes. They are also known as Neighbors of each other.
OR Node B is said to be adjacent to node A if there is an edge form node A to node B.
For example, In Fig1 nodes K and L are adjacent nodes.

Isolated Node:
A node which is not connected to any other node is called isolated node. OR A node having no edge at all is called isolated node. In Fig1, node P is an isolated node.

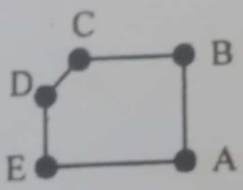
Path: The sequence of edges lying between two nodes is called a path. The no of edges in a path is called length of that path.



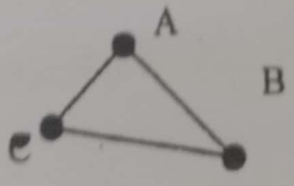
In the above graph, the path between node A and C is having two edges i.e. AB, BC. Thus its length is 2.

Cycle: A Path from a node X to itself is called cycle.

Connected Graph:
A graph G is said to be connected if there is a simple path between two nodes of the graph. Following graph is a connected graph.



Complete Graph:
A graph G is said to be complete if all of its nodes are adjacent to each other. All complete graphs are connected graphs but for a connected graph it is not necessary to be a complete graph.
OR A graph in which each and every node is adjacent to all the other nodes is called complete graph.
For example, the following graph is a complete graph.



CHAPTER 6

GRAPHS

What is Graph?

A graph is a non-linear data structure containing nodes connected by lines/arrows, but not in a hierarchical form.

The nodes in a graph are also called vertices and the lines connecting two nodes are also called edges or arcs. If V is the set of vertices and E is the set of edges then a graph G is mathematically defined as;

$$G = (V, E)$$

Note: The set of vertices and edges must be finite.

Examples: The figures given bellow are graphs.

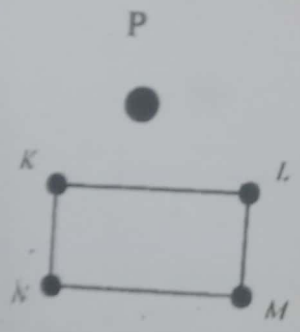


Fig 1

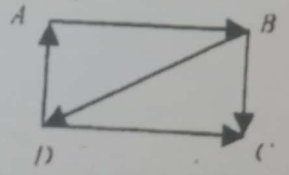


Fig2

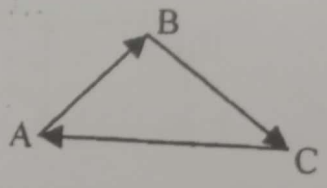


Fig3

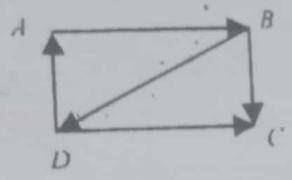
djacent Nodes:

Two nodes connected by an edge are called adjacent des. They are also known as Neighbors of each other.
R Node B is said to be adjacent to node A if there is an ge form node A to node B.
r example, In Fig1 nodes K and L are adjacent nodes.

Isolated Node:

A node which is not connected to any other node is called isolated node. OR A node having no edge at all is called isolated node. In Fig1, node P is an isolated node.

Path: The sequence of edges lying between two nodes is called a path. The no of edges in a path is called length of that path.



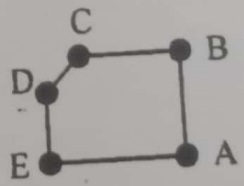
all edges running from origin to destination

In the above graph, the path between node A and C is having two edges i.e. AB, BC. Thus its length is 2.

Cycle: A Path from a node X to itself is called cycle.

Connected Graph:

A graph G is said to be connected if there is a simple path between two nodes of the graph. Following graph is a connected graph.

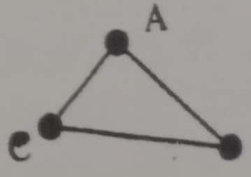


Complete Graph:

A graph G is said to be complete if all of its nodes are adjacent to each other. All complete graphs are connected graphs but for a connected graph it is not necessary to be complete graph.

OR A graph in which each and every node is adjacent all the other nodes is called complete graph.

For example, the following graph is a complete graph.

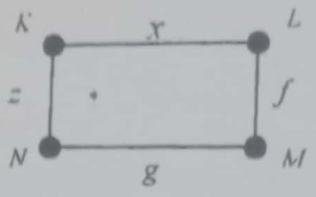


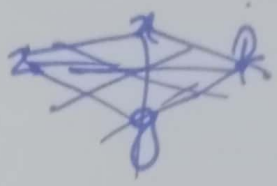
If there are n nodes in a complete graph then there would be n(n-1)/2 edges in the graph.

Labeled Graph:

A graph G is said to be labeled graph if all of its edge are labeled.

For example, the following graph is a Labeled graph.

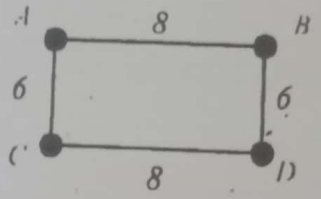




Weighted Graph:

A graph G is said to be weighted if all of its edges are assigned some non-negative numerical values. The numeric value is called weight of the edge.

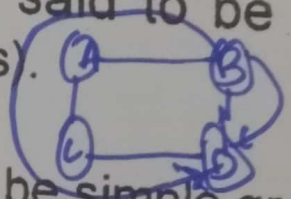
For example the following graph is a weighted graph.



Parallel edges:

Two edges e1 and e2 are said to be parallel if they connect two nodes A, B separately. The edges may be directed or undirected.

OR Two edges e1 and e2 are said to be parallel if they have the same endpoints (nodes).

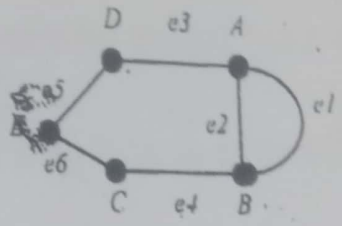


Simple Graph: A graph G is said to be simple graph if it has no parallel edges.

Multiple Graph or Multi-Graph:

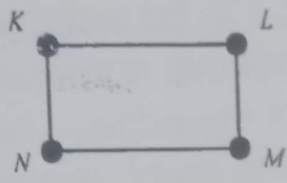
A graph G is said to be Multi-Graph if it has at least one pair of parallel edges. A Multi-graph is shown here in

which there exist parallel edges e1 and e2 between nodes A and B.



Undirected Graph:

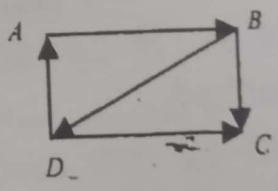
A graph G is said to be undirected if every edge of the graph is undirected.



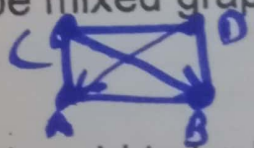
Directed Graph:

A graph G is said to be directed if every edge of th graph has a direction. The arrow head represents th direction of edges.

OR A graph is said to be directed if it has directed edge A directed graph is also called as Digraph. Here is a digrap



Mixed Graph: A graph G is said to be mixed graph if it h directed as well as undirected edges.



Isolated or Null Graph: A graph G is said to be Null gra if all of its nodes are isolated.

(A) (B)
(C) (D)

In-degree of a node:

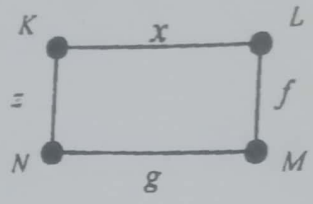
The total number of edges ending at a node is ca In-degree of that node. OR The total no. of edges toward

If there are n nodes in a complete graph then there would be n(n-1)/2 edges in the graph.

Labeled Graph:

A graph G is said to be labeled graph if all of its edge are labeled.

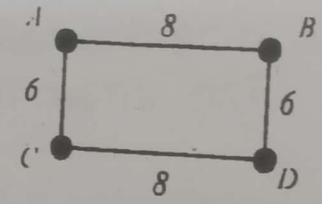
For example, the following graph is a Labeled graph.



Weighted Graph:

A graph G is said to be weighted if all of its edges are assigned some non-negative numerical values. The numeric value is called weight of the edge.

For example the following graph is a weighted graph.



Parallel edges:

Two edges e1 and e2 are said to be parallel if they connect two nodes A, B separately. The edges may be directed or undirected.

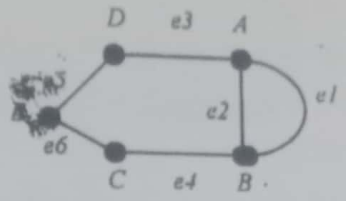
OR Two edges e1 and e2 are said to be parallel if they have the same endpoints (nodes).

Simple Graph: A graph G is said to be simple graph if it has no parallel edges.

Multiple Graph or Multi-Graph:

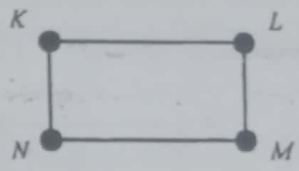
A graph G is said to be Multi-Graph if it has at least one pair of parallel edges. A Multi-graph is shown here in

which there exist parallel edges e1 and e2 between nodes A and B.



Undirected Graph:

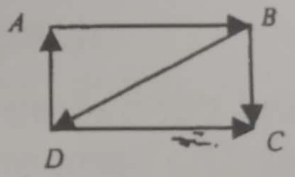
A graph G is said to be undirected if every edge of the graph is undirected.



Directed Graph:

A graph G is said to be directed if every edge of the graph has a direction. The arrow head represents the direction of edges.

OR A graph is said to be directed if it has directed edges. A directed graph is also called as Digraph. Here is a digraph.



Mixed Graph: A graph G is said to be mixed graph if it has directed as well as undirected edges.

Isolated or Null Graph: A graph G is said to be Null graph if all of its nodes are isolated.

In-degree of a node:

The total number of edges ending at a node is called In-degree of that node. OR The total no. of edges towards a

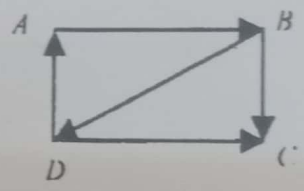
node is called in-degree of that node. It is represented by InDeg(N) where N reprents a node.

In the above mentioned Digraph the In-degree of nodes are given below.

InDeg(A) = 01
InDeg(B) = 01
InDeg(C) = 02
InDeg(D) = 01

Out-Degree of a node:

The total no. of edges drawn from a node is called out-degree of that node. OR The total number of edges starting form a node is called Out-degree of that node. It is represented by OutDeg(N) where N shows a node.



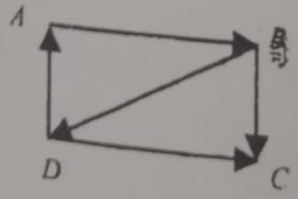
Here,

OutDeg(A) = 01
OutDeg(B) = 02
OutDeg(C) = 00
OutDeg(D) = 01

Degree of a node:

The in-degree and out-degree of a node 'N' is called degree of that node. OR The number of edges contained by a node 'N' is called the total degree of that node. It is represented by Deg(N) where N shows a node.

Consider the following graph.



Here,

Deg(A) = 2

Deg(B) = 3
Deg(C) = 2
Deg(D) = 3

Regular graph: A graph G is said to be Regular graph if all the nodes have the same degree.

Balanced graph: A graph G is said to be Balanced graph if the in-degree and out-degree of any node is equal.

Initiating and terminating nodes: A node N is called initiating node if the edge is drawn/directed from it to another node M. The node M in this case is called terminating node.

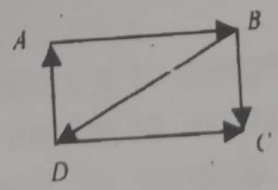
Sink and Source: A node in the directed graph is called Sink if it has zero out-degree and maximum in-degree. A node in the directed graph is called Source if it has zero in-degree and maximum out-degree.

Pendent node: The node whose total degree is 1 is called Pendent node.

Adjacency List:

It is a table or list which contains the adjacent nodes of every node of a graph.

Consider the following graph.

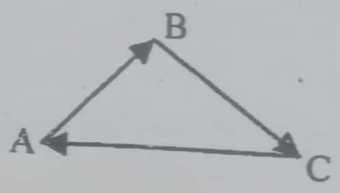


The adjacency list of the above graph is as under;

Node	Adjacent Node
A	B
B	C, D
C	No node
D	A, C

Adjacency Matrix and its Usage:

A matrix MXN having entries 1's and 0's representing the connection between nodes of a graph, is called adjacency matrix. The 1 shows a connection, whereas the 0 shows no connection. It is always a square matrix i.e. if there are n nodes then the order of the adjacency matrix would be nXn. We can find out Path Matrix from Adjacency matrix from which one can prove the number of paths between the nodes. Lets take a directed graph and to find out adjacency and path matrixes.



	A	B	C
A	0	1	0
B	0	0	0
C	1	0	0

Now the adjacency matrix thus formed is;

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

Path Matrix:

A matrix P is called Path or Reach-ability matrix if

$$P = \begin{cases} 1 \text{ if there exists a path between two nodes.} \\ 0 \text{ if there is no path.} \end{cases}$$

Now for path matrix there are three nodes, so we will have to find out A^2 and A^3

$$A^2 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

And

$$A^3 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Now let path Matrix be P

So,

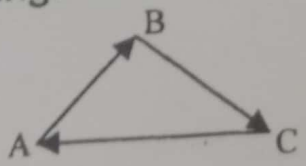
$P = A + A^2 + A^3$

Thus,

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

From the above Path matrix P we can see that there exists: One Path from A to B (as the first row and second column is 1), one Path from C to A (as the third row and first column is 1) and One Path from C to B (as the third row and second column is 1) .

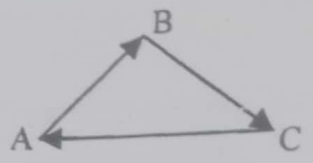
Look and check the digraph again. We can reach from one node to another using the paths proved by the Path Matrix P, i.e. results are matching.



The end

Adjacency Matrix and its Usage:

A matrix MXN having entries 1's and 0's representing the connection between nodes of a graph, is called adjacency matrix. The 1 shows a connection, whereas the 0 shows no connection. It is always a square matrix i.e. if there are n nodes then the order of the adjacency matrix would be nXn. We can find out Path Matrix from Adjacency matrix from which one can prove the number of paths between the nodes. Lets take a directed graph and to find out adjacency and path matrixes.



	A	B	C
A	0	1	0
B	0	0	0
C	1	0	0

Now the adjacency matrix thus formed is;

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

Path Matrix:

A matrix P is called Path or Reach-ability matrix if

$$P = \begin{cases} 1 \text{ if there exists a path between two nodes.} \\ 0 \text{ if there is no path.} \end{cases}$$

Now for path matrix there are three nodes, so we will have to find out A^2 and A^3

$$A^2 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

And

$$A^3 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Now let path Matrix be P

So,

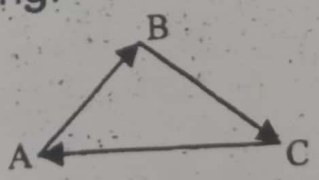
$P = A + A^2 + A^3$

Thus,

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

From the above Path matrix P we can see that there exists: One Path from A to B (as the first row and second column is 1), one Path from C to A (as the third row and first column is 1) and One Path from C to B (as the third row and second column is 1)

Look and check the digraph again. We can reach from one node to another using the paths proved by the Path Matrix P, i.e. results are matching.



The end