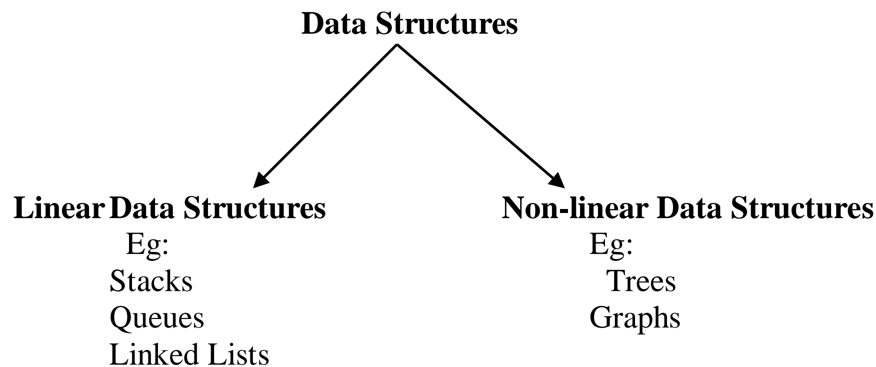
DATA STRUCTURES

Data Structure is a collection of data elements organized in a specified manner and accessing functions are defined to store and retrieve individual elements.

Data Structure is a study of different methods of Organizing the data and possible operations on these structures.

Difference between Data types and Data Structures :

Data type is collection of finite set of permitted values, and a set of operations on these values. Where as, Data Structure is a study about the Organizing the related data to enable us to work efficiently with data, exploring the relationships with the data.



STACK :

A Stack is an ordered collection of items into which new items can be inserted and from which items can be deleted at one end, called the ***Top*** of the stack. That is the removal or addition of elements can take place only at the top of the stack. So stack is expanded and shrink with a passage of elements or items.

A stack is defined as a data structure, which operates on a **Last In First Out** (**LIFO**) basis. It is generally used to provide temporary storage space for values. Stack most commonly used as a place to store local variables, parameters, and return addresses when a function is called. This method is very useful in recursion process.

Operations on Stack:

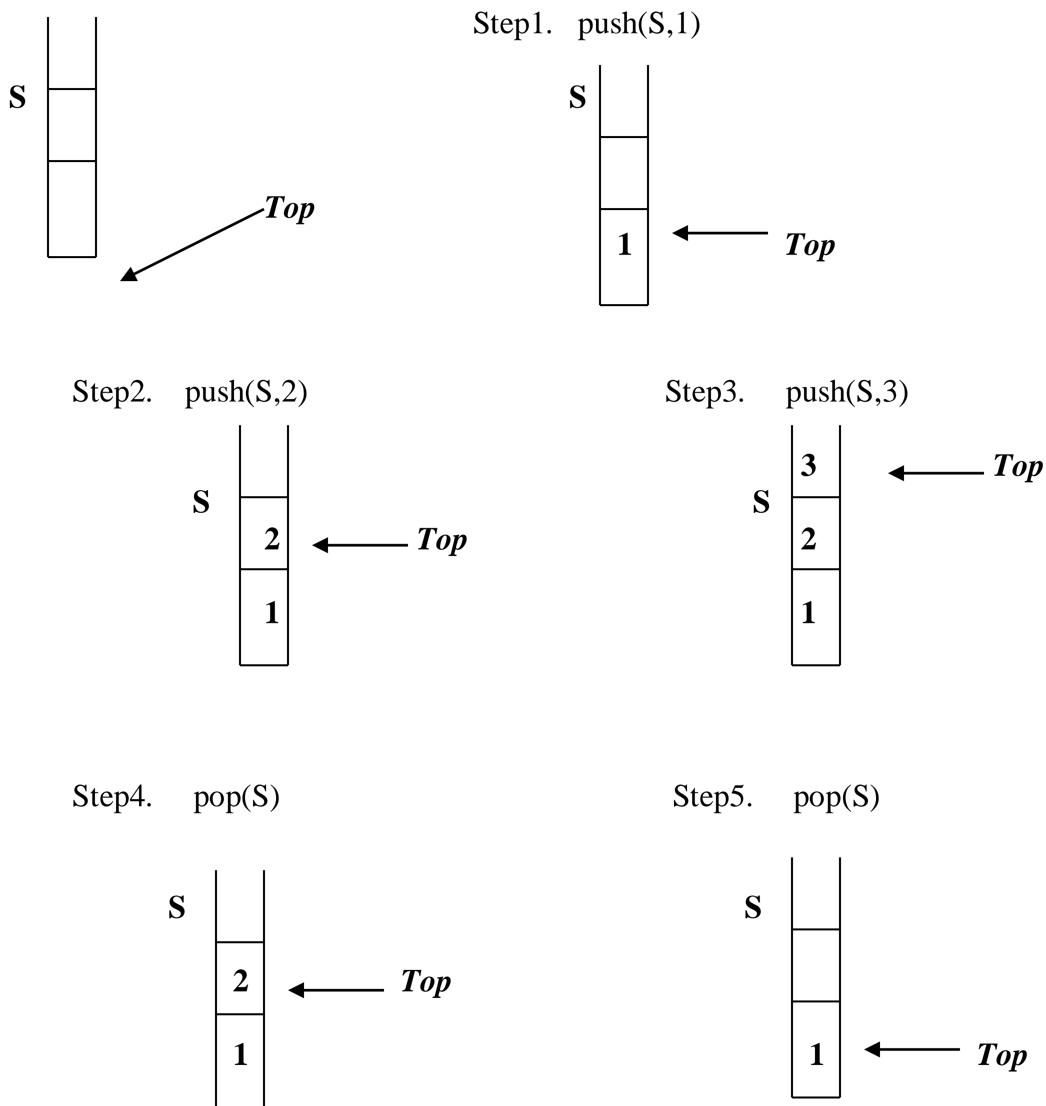
There are several primitive operations. We can define the following necessary operations on Stack.

1). **Create(S)** to create S as an empty Stack
2). **Push(S, i)** to insert the element i on top of the Stack.
3) **Pop(S)** to remove the top element of the Stack and to return the removed element as a function return value.
4) **Top(S)** to return only the top element of the Stack.
5) **Is_Empty(S)** to check whether the Stack is Empty or not ?. If it is empty this function returns value ***one*** i.e. **1**, otherwise returns ***zero*** i.e. **0**
6) **Is_Full(S)** to check whether the Stack is Full or not ?. If it is full this function returns value ***one*** i.e. **1**, otherwise returns ***zero*** i.e. **0**

The Stack Pointer keeps track of the current position of the Stack. When an element is placed or added on the Stack, it is said to pushed on the Stack. When an element is removed from the Stack, it is said to be popped off the Stack. Two additional terms or Pre-conditions almost always used with Stacks are Overflow and Underflow. Overflow occurs when we try to push one more element or more elements on a Stack which is already ***Full***. Where Underflow occurs when we try to pop an element from the Stack which is already ***Empty.***

Representation of Stack

Let us take an Empty Stack whose name is **S** and its capacity is **3**. i.e. This stack maintains only up to 3 elements. The ***Top*** pointer points the top element of the Stack.



Let us consider S is an Array to represent Stack, Top is a Stack Pointer (Index), Max is the Maximum size of an Array S, Item is a value to be store or delete.

Algorithm for Create a Stack.

CREATE(S, Top)

Step 1. Top ← 0
Step 2. Return

Algorithm for Delete an item

POP(S, Top)

Step 1. If (Top = = 0) then
Print ' Stack is Underflow'
Return .
Step 2. item ← S[Top]
Step 3. Top ← Top –1
Step 4. Return(item).

Algorithm for Insert an item

PUSH(S, MAX, Top, item)

Step 1. If (Top = = MAX) then
Print ' Stack Overflow '
Return.
Step 2. Top ← Top + 1
Step 3. S[Top] ← item
Step 4. Return.

Algorithm for Access Top element

TOP(S, Top)

Step 1. If (Top = = 0) then
Print ' Stack is Underflow '
Return
Step 2. Return(S[Top])

Algorithm For Overflow Condition

Is_FULL(S, Top, Max)

Step1. If (Top = = MAX)
Return(1)
Else
Return(0)

Algorithm For Underflow Condition

Is_EMPTY(S, Top)

Step1. If (Top = = 0)
Return(1)
Else
Return(0)

Implementation of Stack

There are two ways to implement the Stacks. They are

1). Using Arrays Implementation
2). Using Linked List Implementation.

Since Stacks are dynamic data Structures, it can grow & shrink during their operations. However, Stacks can be implemented using arrays by specifying a maximum size. So the size of an array is fixed at the time of its declaration itself. We can fix one end of the array as bottom of Stack. The other end of array may be used as a top of the Stack. By using this top end we can pushed and popped the items. We need another field to maintain the index of top of that Array.

Finally we can declare a stack as a structure, containing two members. ***Element*** is an array to maintain elements of the Stack is the first member and the Second one is an integer data type ***top***, to indicate the current top element index of the Stack

Because an array size is fixed, in the array (linear) representation of a stack, only a fixed number of elements can be pushed onto the stack. If in a program the number of elements to be pushed exceeds the size of the array, the program might terminate in an error. We must overcome this problem.

Similar to the linear representation, in a linked representation stackTop is used to locate the top element in the stack. However, there is a slight difference. In the former case, stackTop gives the index of the array. In the latter case, stackTop gives the address (memory location) of the top element of the stack.

Applications Of Stacks

Recursion is the ability of a model to call itself. It is an important facility in many programming languages which uses stacks. Another important application of Stacks is expression evaluation. In general, simple arithmetic expressions can be represented in three ways : Infix, Prefix, Postfix. Consider the binary expression **A + B .** Here ***A*** and ***B*** are two operands and + is a binary operator(An operator is a binary operator if it is associated with two operands)
The three notations are :

A + B	Infix Notation	
+ A B	Prefix Notation	(Polish)
A B +	Postfix Notation	(Reverse Polish)

In Prefix notation the operator precedes the two operands, in Infix notation the operator in between the two operands and in Postfix notation the operator follows both operands.

In any arithmetic expressions the five arithmetic operations Addition, Subtraction, Multiplication, Division and Exponentiation are represented by the following five operators + , **-** , ***** , **/** and **$** (for our convenience $ represents Exponentiation).

Operator - Precedence Rules

Rank	Operator	Associativity
1	( )	Left to Right
2	$	Right to Left
3	* and /	Left to Right
4	+ and -	Left to Right

Note : The Prefix form of a Complex expression is not the mirror image of Postfix form.

The following are some of the examples of expressions in three notations

Infix	Prefix	Postfix
A + B	+ A B	A B +
A + B – C	– + A B C	A B + C –
(A + B) * (C – D)	* + A B – C D	A B + C D – *
((A+B)*C–(D–E))$(F+G)	$–*+ABC–DE+FG	AB+C*DE – –FG+$
A – B / (C * D $ E)	– A / B * C $ D E	A B C D E $ * / –

Eg :-
Conversion **A $ B * C – D + E / F / (G + H)** *from Infix notation to Prefix notation*

Step1. G + H → **+ G H**
A $ B * C – D + E / F / **+ G H**

Step2. A $ B → **$ A B**
$ A B * C – D + E / F / **+ G H**

Step3. $ A B * C → *** $ A B C**
*** $ A B C** – D + E / F / **+ G H**

Step4. E / F → **/ E F**
*** $ A B C** – D + **/ E F** / **+ G H**

Step5. / E F / + G H → **/ / E F + G H**
*** $ A B C** – D + **/ / E F + G H**

Step6. * $ A B C – D → **– * $ A B C D**
– * $ A B C D + **/ / E F + G H**

Step7. – * $ A B C D + / / E F + G H → **+ – * $ A B C D / / E F + G H**

Result : **+ – * $ A B C D / / E F + G H**

Conversion **A $ B * C – D + E / F / (G + H)** *from Infix notation to Postfix notation*

Step1. G + H → **G H +**
A $ B * C – D + E / F / **G H +**

Step2. A $ B → **A B $**
A B $ * C – D + E / F / **G H +**

Step3. A B $ * C → **A B $ C ***
A B $ C * – D + E / F / **G H +**

Step4. E / F → **E F /**
A B $ C * – D + **E F /** / **G H +**

Step5. E F / / G H + → **E F / G H + /**
A B $ C * – D + **E F / G H + /**

Step6. A B $ C * – D → **A B $ C * D –**
A B $ C * D – + **E F / G H + /**

Step7. A B $ C * D – + E F / G H + / → **A B $ C * D – E F / G H + / +**

Result : **A B $ C * D – E F / G H** + / +

Conversion the Expression from Infix to Postfix:

First we scan the Complete Infix expression as a string and select one empty Stack. Next we check each and every character from the first character of the Infix string to last character. While the checking process is going on we insert the character immediately into the Postfix string if the current identified character is an operand, Otherwise (current character is an operator) we keep that operator in the Stack. Before storing that current operator in Stack, we POP the operator from the Stack and insert into the Postfix string as long as the Stack is not empty and the priority of Top operator in the Stack is greater than or equal to the priority of current operator. Later we PUSH the current operator into the Stack even if the above compound condition is failed. This process is completed once the checking is reach to the last character.

Sometimes the Stack still contains some operators after that checking process. If Stack remains some operators, POP the operator from the Stack until Stack is empty and insert into the Postfix string.

Algorithm

Step1. Clear the Stack **S** .
Step2. Read the Complete Infix expression
Step3. ***symbol*** ← First Character in Infix expression
Step4. While(***symbol*** is not a End of Character)

```
    {
        If symbol is an Operand
            Add symbol to the Postfix string
        Else
          {
            While( ! Is_Empty(S) && Precedence( Stack_Top(S) , symbol) )
              {
                Topsymb ← Pop(S)
                Add Topsymb to the Postfix string
              }
            Push( S , symbol)
          }
        symbol ← Next Character in Infix expression
    }
```

Step5. While(! Is_Empty(**S**))

```
    {
        Topsymb ← Pop(S)
        Add Topsymb to the Postfix string
    }
```

Eg:- Infix String A + B * C

Symb	Stack Contents	Output (Postfix String)
A	Empty	A
+	+	A
B	+	A B
*	+ *	A B
C	+ *	A B C
End of Input	Empty	A B C * +

Evaluation Of Postfix Expression

First we scan the complete Postfix expression as a string and select one empty Stack. In Postfix expression, each operator refers operation on the previous two operands. So that we start with first character of Postfix string and check each & every character until the last character. If the current character is an operand then put that operand into the Stack by PUSH. Otherwise (i.e. it is an operator and it refers to the Top two operands) we get the top two operands separately by POP, perform the indicated operation on the two operands and PUSH that result on to the Stack. The result will be available for use as an operand of the next operator.

If the checking reach to the last character, the above process completed and the Stack contains only one operand that is the result of the Postfix expression

Algorithm

```
Step1.  Clear the Stack S.
Step2.  Read the Complete Postfix expression.
Step2.  symbol ← First Character in Postfix expression.
Step3.  While(symbol is not a End of Character )
            {
                If(symbol is an Operand )
                        Push(S, symbol)
                Else
                  {
                        X2 ← Pop(S)
                        X1 ← Pop(S)
                        R  ← Result of applying Symb to X1 & X2
                        Push(S, R)
                  }
                symbol ← Next character in Postfix expression
            }
Step4.  Return( Pop( S ) ).
```

Check Whether the Arithmetic expression is properly balanced or not?

To indicate the boundaries of sub expressions, we some times use BRACKETS (parentheses, brackets and braces). So we must be care take to balance all these BRACKETS properly in the given expression. Otherwise the expression may be corrupt.

Eg:- { [a * b - (b + c)] * [sin(x-y)] } – (x – y)

To determine, whether the arithmetic expression is properly balanced or not we must check that a right counter part ')' or '**]**' or '**}**' exist for each left '(' or '**{**' or '**[**' in a proper order. To do this we use a Stack and start search from the first character to the last character for the given expression. Whenever we encountered a LEFT - BRACKETS *(* left parentheses '(' or left braces'**{**' or left bracket '**[**' *)* , we push onto the Stack. Whenever we encountered a RIGHT - BRACKETS *(* right parentheses ')' or right bracket '**]**' or right braces '**}**' *)* , we pop the top symbol from the Stack and check to see whether the encountered RIGHT - BRACKET and Stack top LEFT - BRACKET are same type or not?. We continue if they are same type, otherwise we stop the process.

The expression has properly balanced, if the Stack is empty by the time we get to the end of the expression and all pairs of matched BRACKETS were of the same type. Otherwise, the BRACKETS are not balanced properly.

Algorithm

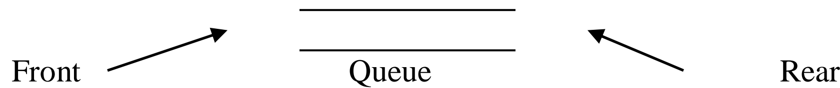
```
Step1.  Matching ← True
Step2.  Clear the Stack S.
Step3.  Read symbol as a first character from the input string.
```

```
Step4.  While (symbol is not a end of input string & Matching is True )  do
           {
                 If ( symbol is ‘ ( ‘   or   symbol is  ‘ [ ‘   or  symbol  is  ‘ { ‘  )
                         Push( S ,symbol )
                 Else if ( symbol is ‘ )‘   or   symbol is  ‘ ] ‘   or  symbol  is  ‘ } ‘  )
                    {
                         If  (  Is_Empty(S) )
                            {
                            Matching   ←  False
                            Write “More Right Brackets than Left Brackets“ && Return
                             }
                         Else
                            {
                                 topsymb  ←   Pop(S)
                                 If ( ! Equal_Type( topsymb , symbol ) )
                                    {
                                         Matching  ←  False
                                         Write “ Miss matched Brackets “  &&   Return
                                    }
                            }
                    }
                 symbol    ←    Next  character in the input string.
           }
Step5.  If ( Is_Empty(S) )
                 Write “ Brackets are Properly Balanced”
        Else
                 Write “ More Left Brackets than Right Brackets “
```

QUEUE

A Queue is an ordered group of elements in which elements are added one end (known as ***Rear***) and elements are deleted from the other end (known as ***Front***). There fore this data Structure is commonly known as **FIFO** (First In First Out) list.

Eg:- A line at a Bank or at a Bus stop or at a Ticket counter.

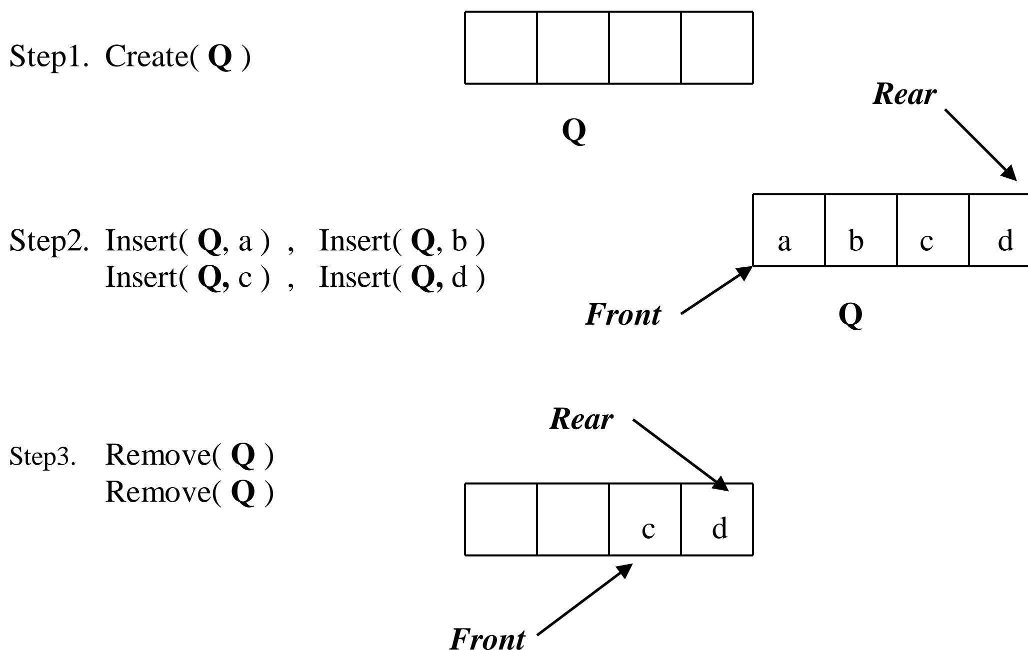


The following are the Primitive Operations on Queue.

1. CreateQ(**Q**)	Creates an empty Queue **Q.**
2. Insert(**Q**, x)	Insert an item x at the rear of the Queue **Q.**
3. Remove(**Q**)	Delete the front element from the Queue **Q.**
4. Is_empty(**Q**)	It checks whether the Queue contain any elements or not ? and it returns ***true*** if it is empty, otherwise it returns ***false***.
5. Is_full(**Q**)	It checks whether the Queue completely filled or not ? and it returns ***true*** if it is full, otherwise it returns ***false***.

The insert operation can always be performed, since there is no limit to the number of elements a Queue may contain i.e. In theory, there does not exist a ***Queue Overflow***. But in practice, implementation of a Queue have a maximum ***Queue Size***. The remove operations, however, can be applied only if the Queue is not empty. Otherwise we may get a situation ***Under Flow***.

Let us create an empty Queue **Q** with capacity **4** and perform Queue Operations. The following steps represents the Diagrammatic representation of Queue.



Let us assume **Q** is an Array to represent Queue. The first and last elements are indicated by front & rear. MAX is the Maximum size of an Array **Q** and Item is a value to be store or delete

Algorithm for to Initialize a Queue

CreateQ(Q)

Step1. Rear ← 0
Step2. Front ← 1
Step3. Return

Algorithm for to Insert an item into a Queue

InsertQ(Q, item, Rear, MAX)

Step1. If (Rear= = MAX) then
{
Write "Queue Overflow".
Return
}
Step2. Rear ← Rear + 1;
Step3. **Q**[Rear] ← ***item***
Step4. Return.

Algorithm for to Delete an item from the Queue

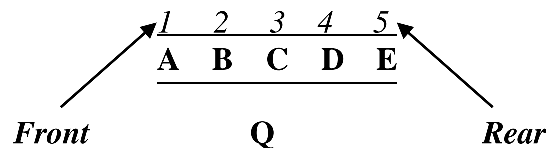
RemoveQ(Q, Rear, Front)

Step1. If (Front > Rear) then
{
Write " Queue Underflow".
Return.
}
Step2. ***item*** ← **Q**[Front]
Step3. Front ← Front + 1
Step4. Return(***item***)

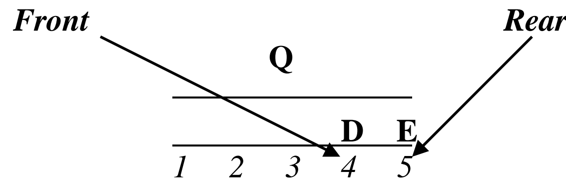
Note: Total Number of elements in Queue is **(Rear + 1) – Front**.

Drawback in Linear Queue :

To be explain this concept, let us consider a Queue **Q** with capacity **5** and insert the first five Alphabets (A, B, C, D, E.) onto the Queue **Q**. The Queue is completely full and the bellow figure represents the state of the Queue **Q**.



Suppose, if we delete the first three elements (A, B, C.) from the Queue **Q** then the Queue contains only two elements (D & E). Rear still points to the same element but Front points to an element D. The state of the Queue is :



Now we try to add some elements onto the Queue, since the Queue is not filled . But according to **InsertQ** *algorithm* we can't add any element. Because the rear reach to the last position of the queue. Even the queue contains some free positions, we can't insert elements onto the queue. This is the ***drawback*** of **Linear Queue**.

There are some methods to overcome this drawback . In one method at the time of deletion, after delete an element we check whether the Queue is empty or not?. If it is empty then immediately we reinitialized the queue. i.e. we set front & rear positions to the initial positions as in **CreateQ** *algorithm.*

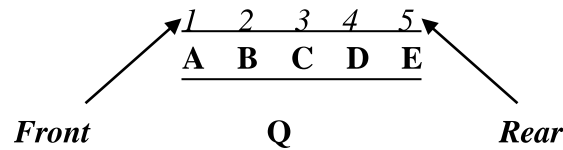
The Modified Algorithm for to Delete an item from the Queue

Remove(Q, Rear, Front)

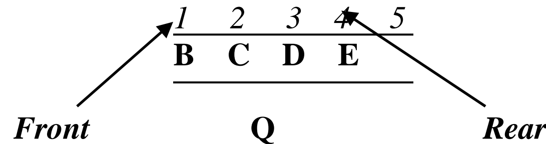
```
Step1.  If ( Front  > Rear)  then
            {
                  Write " Queue Underflow".
                  Return.
            }
Step2.  item  ← Q[Front]
Step3.  Front  ← Front + 1
Step3.  If( Front = = Rear) then
            {
                  Front   ← 1
                  Rear    ← 0
            }
Step4.  Return( item )
```

In second method we implement the queue like a Stack, where we fixing one end of the Stack Similarly, in Queue, we fix the Front of the Queue so that it is always representing the first element of the array. This means whenever we delete an element, the Queue is shifted to the beginning of the Array. ***i.e.*** The deleted first position is occupied by second element, and the second position is occupied by third element and so on up to last element. The below Steps & figures represents the process of this method

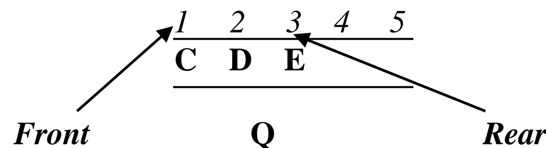
Step1. Create a Queue **Q** and Insert the first five Alphabets (A, B, C, D, E.).



Step2. Remove(**Q**)

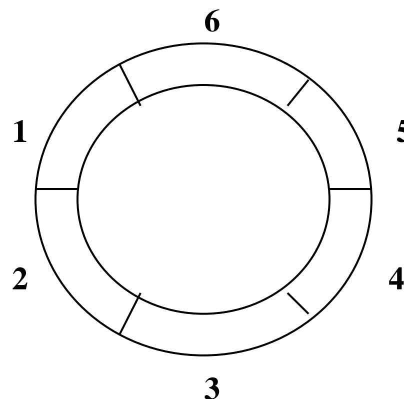


Step3. Remove(**Q**)



This Procedure could be successfully implemented, but it is too inefficient. Because, if we want to delete an element or some elements from a large Queue then the processing required to remove first element and shifting all the elements towards its left will be a costly affair.

Another solution to this problem is to use the array holding the Queue as a **Circular array**. This means we imagine the first element of the array as immediately following its last element. ***i.e.*** even though the last position of the array is occupied, a new element can be added at the first position of the array. The diagrammatic representation of Circular Queue is :

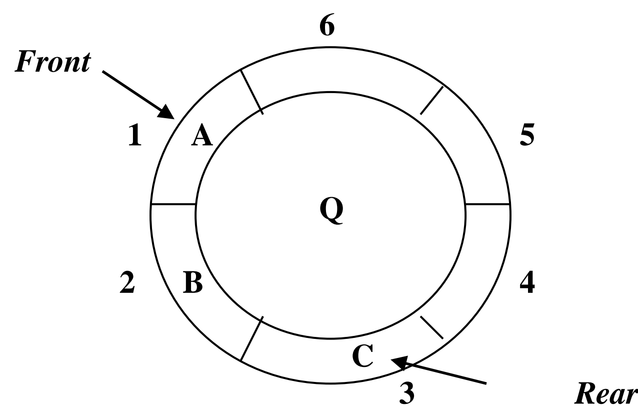


CICULAR QUEUE

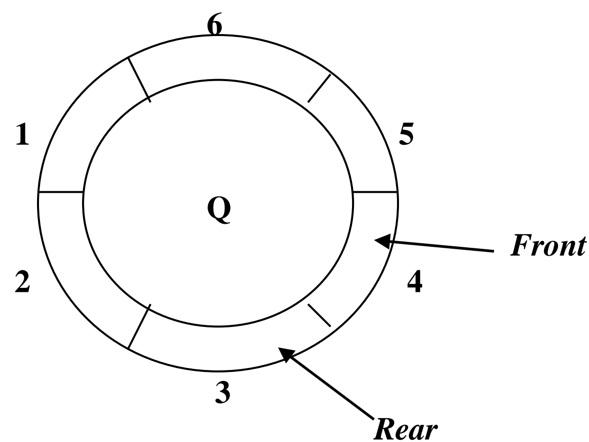
Circular Queue is a Queue, in which the last element comes just before the first element and here also elements are added through ***Rear*** and deletions performed through ***Front***. Generally in Circular Queue, we can't fix the first and last positions, because the ***Front*** & ***Rear*** are circularly rotated. While inserting elements on Circular Queue, as long as the current position of rear is not same as the ***last position*** (***MAX*** **th** position) of Circular Queue, rear is incremented by one and insert the new item in ***Rear*** **th** position. Whenever rear is located in ***last position***, instead increment rear, reset the rear to the first position of Circular Queue. The same procedure followed for ***Front*** while deleting elements from circular Queue.

Let us consider an Empty Circular Queue and observe the situations while performing Insertions & deletions. The following diagram represents its states.

Step1. Create**CQ(Q)** & Insert the first three Alphabets (A, B, C).



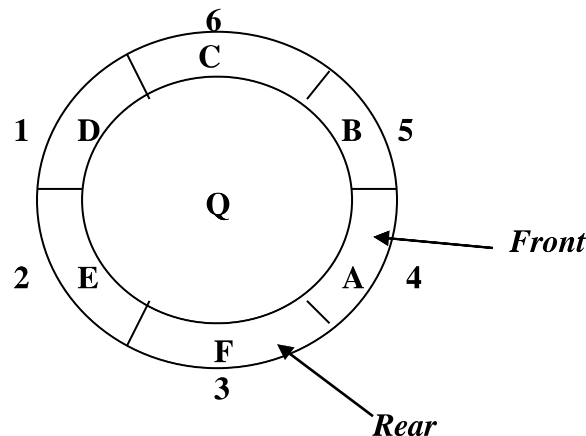
Step2. Delete all the elements from the Circular Queue.



In this Step ***Rear*** position is **3** and ***Front*** position is **4.**

i.e. ***Rear = 3*** and ***Front = 4***

Step3. Insert elements on Circular Queue, until it gets full.



In this Step ***Rear*** position is **3** and ***Front*** position is **4.**
i.e. ***Rear = 3*** and ***Front = 4***

What we have observed is, the ***Front*** & the ***Rear*** have the same values ***4 & 3*** respectively in both the steps **Step2** and **Step3**. But **Step2** represents ***Underflow*** situation where as the **Step3** represents ***Overflow*** situation. So it is very difficult to determine the Queue state, when the Queue is empty & when it is full.

To overcome the above problem, we use one ***Dummy*** position in Circular Queue that is always pointed by ***Front***. The last element is pointed by ***Rear*** but the first element is the next immediate element of ***Dummy*** position. So the first element in Circular Queue is the immediate next element of ***Front***. That means ***Front*** points to recently deleted item position but not the first element position.

Algorithm for to Insert an item into a Circular Queue
InsertC_Q(Q, item, Rear, MAX)

```
Step1.  If ( Rear= = MAX)   then
                Rear ← 1
        Else
                Rear ← Rear + 1
Step2.  If ( Front = = Rear )   then
           {
                Write " Circular Queue is Overflow "
                If ( Rear = = 1 )  then
                        Rear ← MAX
                Else
                        Rear ← Rear – 1
                Return.
           }
Step3.  Q[Rear] ← item
Step4.  Return.
```

Algorithm for to Delete an item from the Circular Queue

RemoveC_Q(Q, Rear, Front)

Step1. If (Front = = Rear) then
{
Write " Queue Underflow".
Return.
}
Step2. If (Front = = MAX) then
Front ← 1
Else
Front ← Front + 1

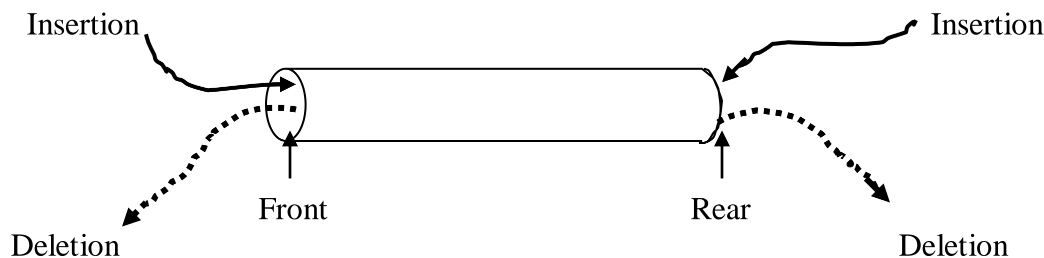
Step3. ***item*** ← **Q**[Front]
Step4. Return(***item***)

In Time Sharing Operating System, **N** number of Users are connected to the Computer System and all the Users share the System with a common ***time – slice*** *or* ***time Quantum***. Here CPU is switching from one *User Job* to another *User Job* when the allotted time – slice is completed and this Process is going on from first User to last User in a ***circular manner***. So this is One of the Application of Circular Queue.

Double Ended Queue

A Double Ended Queue, or simple pronounced as ***Dee-Q***, is a Queue in which the elements can be inserted and deleted at both ends of a Queue. This is very flexible structure when compared to Stacks and Queues.

The conceptual view of Double Ended Queue is

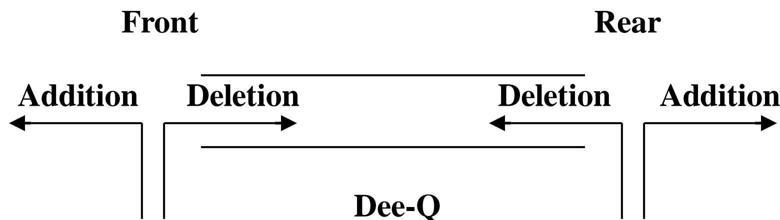


Dee-Q can also maintained in circular fashion. We can have several other types of Queues by limiting or relaxing some constraints. Two out of them are frequently used:

1. Input – restricted Double Ended Queue
2. Output – restricted Double Ended Queue

The Input– restricted Double Ended Queue is a Double Ended Queue in which deletions are allowed at both ends but insertions are allowed only at one end. The Output– restricted Double Ended Queue is a Double Ended Queue in which insertions are allowed at both ends but deletions are allowed only at one end.

Note:- In this ***Dee-Q,*** Rear and Front initialized to Zero position. Rear points to last item position in the List but Front points to recently deleted item position. i.e. Front points to preceded element of the first item in the List.



Let us assume **Q** is an Array to represent ***Dee-Q***. The MAX is the Maximum size of an Array **Q** and Item is a value to be store or delete. Front in Left side and Rear in Right side of the ***Dee-Q***. The following algorithms represent the operations of ***Dee-Q***.

Front Side Operations

Addition

```
Step1  If ( Front = = 0 )
          {
             Write " Addition is not Possible at Front Side"
             Return
          }
Step2  Q[ Front ]  ←  Item
Step3  Front  ←  Front –1
Step4  Return
```

Deletion

```
Step1  If ( Front = = Rear )
          {
             Write " Deletion is not Possible at Front Side"
             Return
          }
Step2  Front  ←  Front + 1
Step3  Item  ←  Q[ Front ]
Step4  Return(Item)
```

Rear Side Operations

Addition

```
Step1  If ( Rear = = MAX )
         {
            Write " Addition is not Possible at Rear Side"
            Return
         }
Step2  Rear ← Rear + 1
Step3  Q[ Rear ] ← Item
Step4  Return
```

Deletion

```
Step1  If ( Front = = Rear )
         {
            Write " Deletion is not Possible at Rear Side"
            Return
         }
Step2  Item ← Q[ Rear ]
Step3  Rear ← Rear –1
Step4  Return(Item)
```

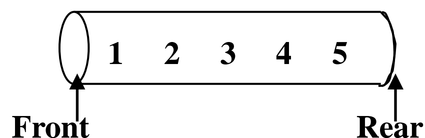
PRIORITY QUEUE

A Priority Queue is a Queue that contains items that have some predefined ordering. Unlike the usual removal of the first item, when an item is removed from a Priority Queue, the item with highest priority is removed.

To implement Priority Queue we follow two methods. The First method is while inserting items onto the Priority Queue the items should be stored in their corresponding positions on the basis of their priority. In Second method at the time of insertion items are stored in Queue manner but at the time of deletion the deleted item should be the highest priority item.

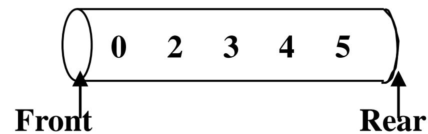
For example, let us assume that an integer Priority Queue has ***small integer with the highest priority*** and added the items into the Priority Queue **4, 3, 1, 5, 2** sequentially. Now we observe the two different approaches.

According to First method, the state of Priority Queue is given bellow. Here the item to be deleted is pointed by **Front,** which has the highest priority than all other items

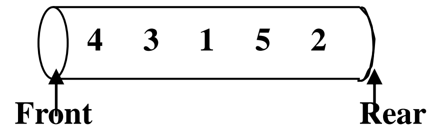


and also the first inserted element than all other elements. The item to be deleted from this Priority Queue is **1**. If the number **0** is added to the Queue after the removal of **1**,

then the **0** will be in the front of the Queue, even though it is last item to be added to the Queue. Now the new state of this Priority Queue is



According to Second method, new item inserted at **Rear** side but **Front** does not point to the deleted item position (of course **Front** points to First inserted item position but not deleted item position). So the state of Priority Queue is given bellow. To be delete



an item from this Queue, first it search for an element which has the highest priority and then deletes that element.

Note:- If a new item is added to the Priority Queue which has the same value as in ***existing number*** of the Queue, then the new item has a lower precedence than the item already existing.

LINKED LISTS

In an Array successive node of the data objects are at a fixed distance apart. So there are some drawbacks in Arrays. They are

1) Let us consider an Array whose name is **List** with size ***20*** and it contains only ***10*** elements. They are **{ 1, 5, 8, 9, 12, 15, 55, 59, 68, 75 }**.

1	**5**	**8**	**9**	**12**	**15**	**55**	**59**	**68**	**75**										
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*

Suppose we want to add **10** to this List (in ascending order). Then we will have to move the elements **12, 15, 55, 59, 68, 75** one location up to make space for new element **10** at the proper place. Similarly, if we have to delete an element **55** from this List, we again have to move **59, 68, 75** one location down. This particular problem occurs when we perform insertions or deletions in the middle and we have to move many elements so as to maintain sequential representation of the list.

2) In Arrays storage may be wasted. This means we have to allot the Maximum size, but in some cases we can't reach to (use) maximum size.

3) In Arrays, once the memory size is allotted at its declaration, we can't increase or decrease its size during the execution of a program.

Using Linked Lists can solve all these problems. In Linked Lists, the item may be placed any where in memory instead of being located a fixed position in the case of sequential representation. In this Linked representation, we store the address of the next element with the Data item.

Linked List:

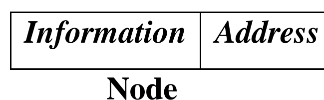
A linear collection of data items, each data item is connected with their adjacent data items. List is a Dynamic data structure and its size is variable. Items may be added and to it or deleted from it. That's why the Linked List is a very powerful and flexible Dynamic data structure.

Linked allocation:

In Linked lists the data items are logically adjacent but not physically. Each item is allocated space as it is added to the list. A link is kept with each item to the next item in the list. This type of allocation is called Linked Allocation.

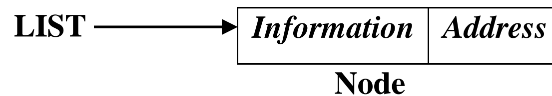
The item in the Linked List is called a **Node** . Each **item** or **node** in a linked list must contain at least two fields, one is ***information*** *field* and the other one is ***address*** *field* (address of the next item). The first field contains the actual element on the list which may be simple integer, character, a string or even a large record. The second field which is a pointer contains the address of the next node in the list used to access the next item.

The Diagrammatic representation of an item or node is



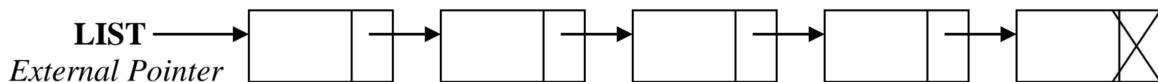
Node

The entire list can be maintain by an ***external pointer*** (**LIST**), pointing to the first node in the list. So we can access the first node through the external pointer.



Node

The second node through the *address field* (pointer) of the first node, the third node through the *address field* (pointer) of the second node etc… till the end of the list. The address field of the last node contains a special value known as NULL and it is diagrammatically represented by a symbol ⊠ . This is not a valid address, only tells us that we have reached the end of the list.



Drawbacks in Linked Allocation

1. No direct access to a particular element.
2. Additional Memory required for pointers (***address fields***).

   However, these limitations are not consider because of their powerful capabilities.

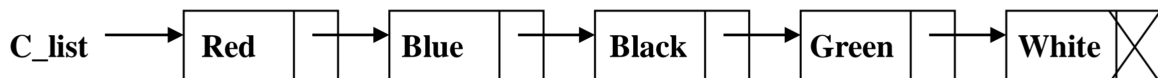
Self Referential Structure

It is sometimes desirable to include with in a structure one member that is a pointer to the parent structure data type. In general terms, this can be represented as

```
struct tag
  {
      data type      member 1;
      data type      member 2;
        .
        .
      struct tag     * member N;
  };
```

Where **member *N*** refers to the name of a pointer variable. Thus, the structure of type tag will contain a member that points to another structure of type tag. Such structures are known as Self Referential Structure.

Eg:-
```
struct link
   {
       char      color[10];
       struct link   *next;
   };
```



Where **C_list** is an External Pointer.

Operations on Linked List

There are five basic types of operations associated with list data abstraction.

1. To determine if the list is empty or not ?.
2. Add new elements any where in the list.
3. To check if a particular element is present in the list or not?
4. To delete a particular element from the list placed any where in the list
5. To print all the elements of the list.

If **P** is a pointer to a node, then the notations to be used in **Algorithm** are :

Node(**P**) refers to the node pointed by **P.**
Info(**P**) refers to the data part of that node pointed by **P**.
Next(**P**) refers to the Address part of that node pointed by **P**.
Info(***Next***(**P**)) refers to the data part of the next node which follows ***Node***(**P**)

We can initialize the list by making the external pointer ***NULL***.

LIST ← ***NULL***

We can check whether the list is empty or not? by checking whether the external pointer is NULL or not?. ***i.e.***

```
If LIST = = NULL
    Write " List is Empty"
Else
    Write " List is not Empty"
```

To visit or traverse or print the entire list from the first node to last node, we need a to use a temporary pointer, **T** known as traversal pointer.

```
T = LIST
While( T != NULL) do
    {
        write " Info( T ) "
        T ← Next( T )
    }
```

Inserting a node into Unordered Liked List

To add a new node containing data value ***x*** in the list, we need to follow the steps.

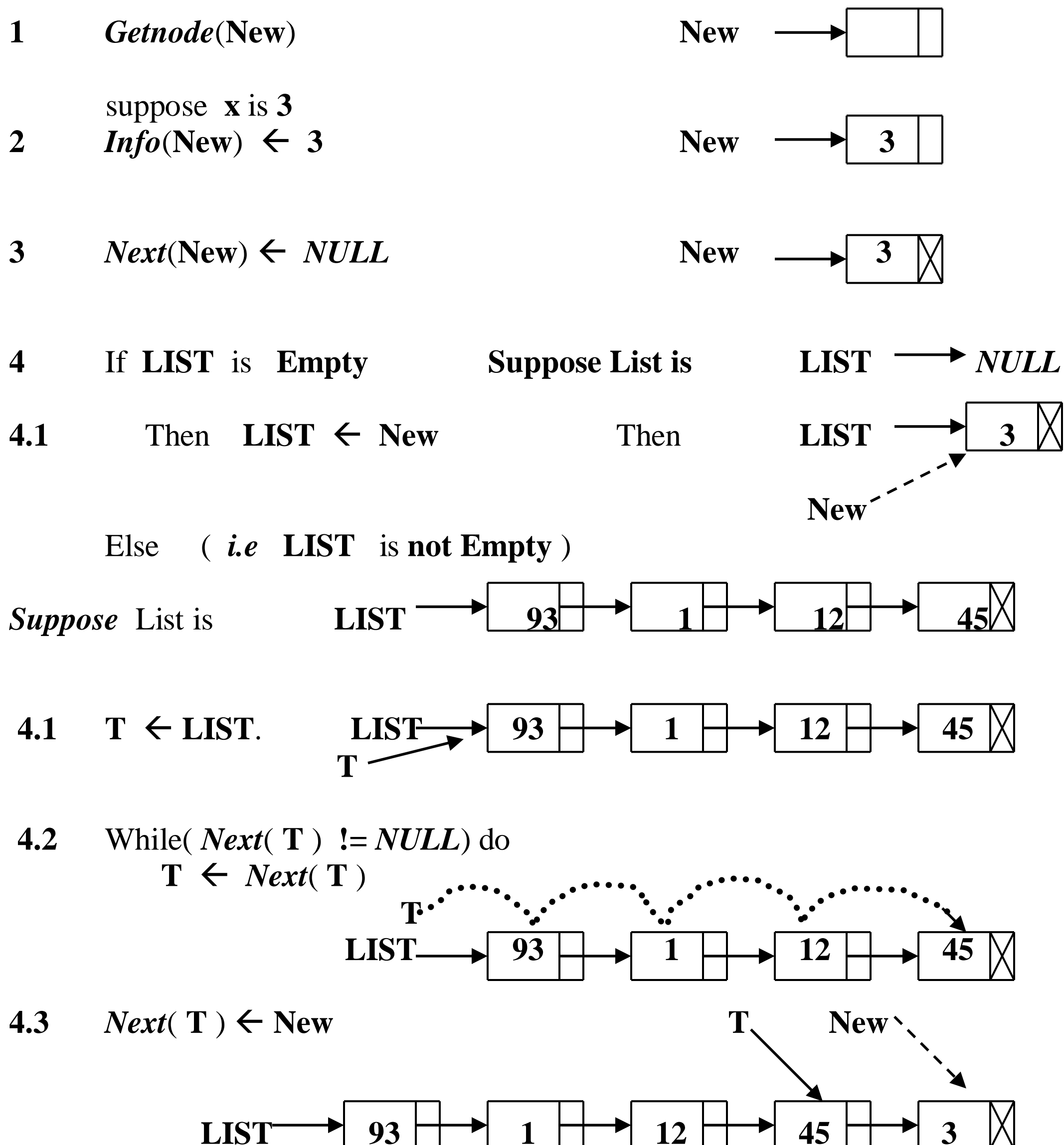
1. Get a new node, which is not in use.
2. Set the data field of the new node to ***x***.
3. Set the next field of the new node.
4. Set the pointer list point to the new node.

First we get a new node by using the notation ***Getnode***(**New**). This ***Getnode***(**New**) creates new node and sets the address of new node to a pointer **New**. Latter we assign the data value ***x*** in ***info*** *field* of new node and we set the ***NULL*** value to ***next*** *field* or ***address*** *field* of the new node. Now we have to decide where the new node to be inserted. Generally in Liked Lists, we insert new node in last position of the List. Before inserting a new node, first we have to check whether the List is empty or not? If the List is empty then we assign the new node address to List External pointer, there after new node is behave like a first node. Otherwise we reach the last node by ***traversing*** technique and add this new node to the last node of List.

To de this, we can write the following **Algorithm**.

Step ***1*** ***Getnode***(**New**)
Step ***2*** ***Info***(**New**) ← ***x***
Step ***3*** ***Next***(**New**) ← ***NULL***
Step ***4*** If(**LIST** = = ***NULL***) Then
Step ***4.1*** **LIST** ← **New**
Else
{
Step ***4.1*** **T** ← **LIST**
Step ***4.2*** While(***Next***(**T**) **!=** ***NULL***) do
T ← ***Next***(**T**)
Step ***4.3*** ***Next***(**T**) ← **New**
}

The diagrammatic representation of inserting a new node in Linked List is :



Insert a node into Order Liked List

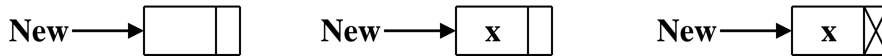
In Order Linked List all the Nodes are arranged in sequence (sorted order) on the basis of its information. So to insert a new node in this List, first we have to search proper position, and then add new node in that position. The position may be first one or middle one or last one in the List. The following Algorithm can handle and use different methods to insert a node at any position in the Linked List. In this Algorithm we consider two traversal pointers **Curr** & **Rear**. **Curr** visits the List from first node of the List until last node and **Curr** try to identify our required position to be insert a new node, where **Rear** stands behind **Curr**. **i.e.** while **Curr** is visiting, **Rear** points the previous node of **Curr**.

After Complete their visit, we check **Curr** & **Rear** states**.** If **Curr** & **Rear** does not point to any node then the List state is Empty and we insert ***first node***. If **Curr** does not point to any node but **Rear** points to some one (ie. Last node) then the new node to be inserted at ***last position***. If **Curr** points to some one(ie. First node) but **Rear** does not point any node then the new node to be inserted at ***first position***. If Curr & Rear point to two different nodes separately then the new node to be inserted in between **Curr** & **Rear**.

```
Step1          Getnode( New )
Step2          Info(New ) ← x
Step3          Next(New) ← NULL
Step4          Curr ← LIST
Step5          Rear ← NULL
Step6          While( Curr != NULL && x > Info(Curr) ) do
                 {
Step6.1              Rear ← Curr
Step6.2              Curr ← Next(Curr)
                 }
Step7          If( Curr = = NULL)
                 {
Step7.1              If( Rear = = NULL)
Step7.1.1                LIST ← New
                     Else
Step7.1.1                Next( Rear ) ← New
                 }
               Else
                 {
Step7.1              If( Rear = = NULL)
Step7.1.1              {
                         Next(New) ← Curr
                         LIST ← New
                       }
                     Else
                       {
Step7.1.1                Next( Rear ) ← New
Step7.1.2                Next( New ) ← Curr
                       }
                 }
```

The diagrammatic representation of inserting a new node in Order Linked List

By Steps**1,2&3.** ***Getnode***(**New**) ***Info***(**New**) ← *x* ***Next***(**New**) ← ***NULL***

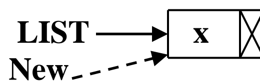


If List is Empty

Suppose the List is **LIST** → ***NULL***

By Steps**4&5** **Rear** points to ***NULL.*** Since List is ***NULL,*** **Curr** also points ***NULL***.
In Step**6** Since **Curr** is ***NULL***, **Control** does not enter into Step**6**.

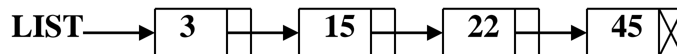
Case: I In this Case **Curr != *NULL*** and **Rear != *NULL***
So, whatever the value of **x**, we add the **New** to **LIST.**



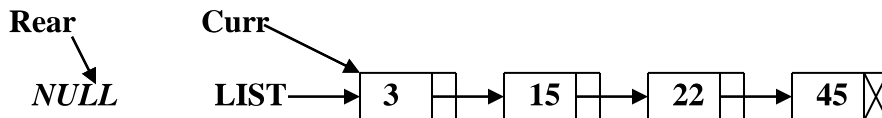
The new node is the first node in the List

***If* List is not Empty**

Suppose List is



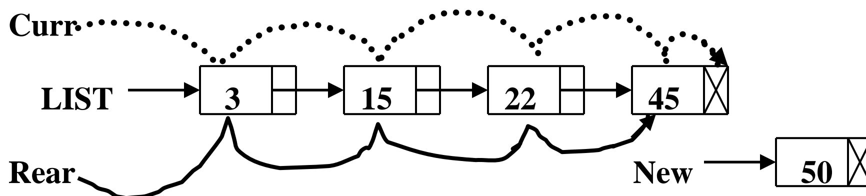
By Steps**4&5**



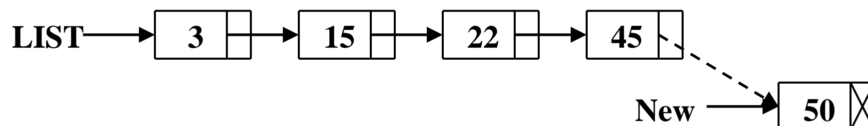
***Let us take* X *value is* 50**

The new node is **New** → **50**

After Step**6, t**he States of **Curr** & **Rear** are



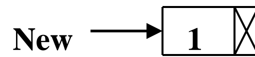
Case: II In this Case **Curr** = = ***NULL*** and **Rear != *NULL***
So, we add **New** as the next node of **Rear**



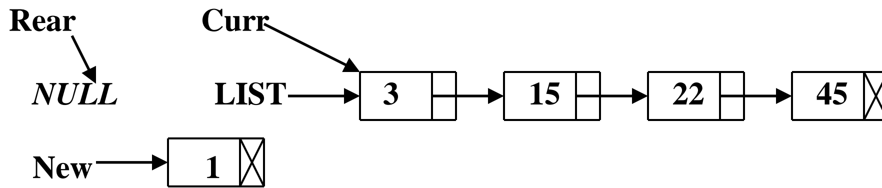
, ***The new node is added at last position in the List***.

***Let us take* X *value is* 1**

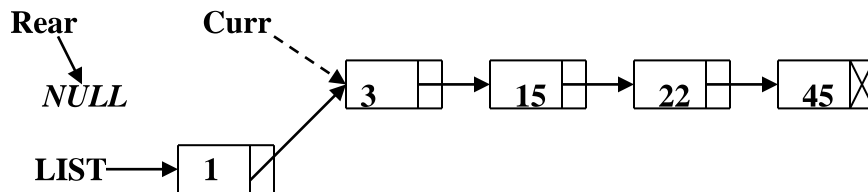
The new node is



After Step**6, t**he States of **Curr** & **Rear** are



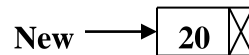
Case: III In this Case **Curr ! =** ***NULL*** and **Rear = =** ***NULL***
So, we add **New** as the first node of **LIST**



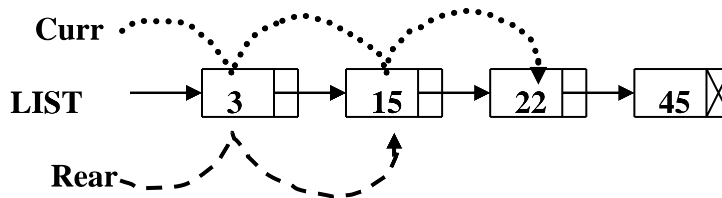
The new node is added at first position in the List.

***Let us take* X *value is* 20**

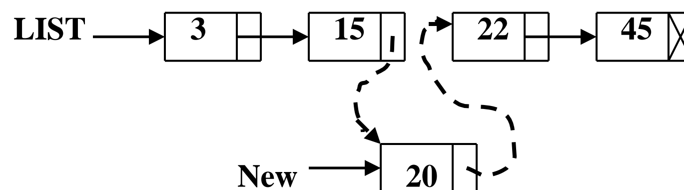
The new node is



After Step**6, t**he States of **Curr** & **Rear** are



Case: IV In this Case **Curr !=** ***NULL*** and **Rear !=** ***NULL***
So, we add **New** in between **Rear** & **Curr**



The new node is added in middle of the List

Delete a node from Order / Un order Liked List

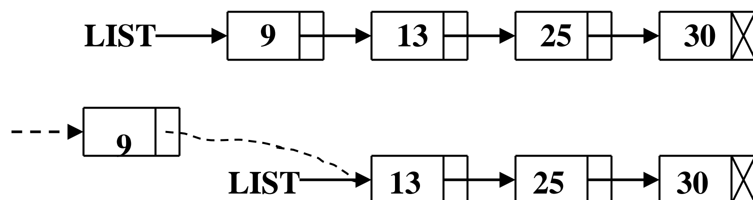
There may be nodes in the List, which are no longer required. Deletion of such nodes is an essential operation on a Linked List. At the time of deletion, we may get any one of the following four situations.

1. List is Empty and tries to delete a node.
2. What node we want to be deleted that node does not exist in the List.
3. The node to be deleted is a first node in the List.
4. The node to be deleted is a last node or in the middle of List.

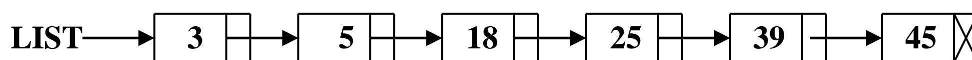
When we want to delete a node from the List, some times the List may contain no one node. In this case we have to print a message " **List is Empty** & **We can't delete a Node**".

In some cases even the List is not empty, a node we want to be deleted may not exist in the List. Whenever that case is arrived, we have to print a message " **Node does not exist in the List** ".

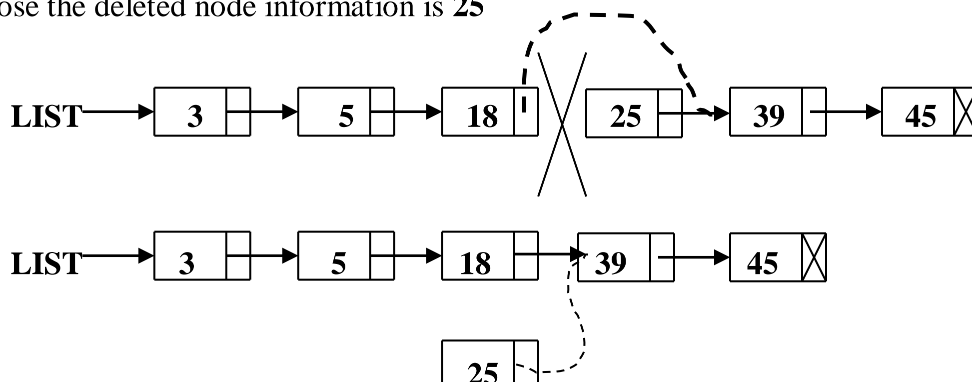
If the node to be deleted is the first one in the List, we are simply required to move the ***External pointer*** from the first node to the second node of List. Since the External pointer has changed its pointing location from deleted (first) node to its next (second) node, there after no way is there to visit that deleted node. This can be represented by the following diagrammatic manner.



If the node to be deleted is in the middle or in last position of the List, we consider two pointers **Curr** & **Rear**, **Curr** starts with the ***first node*** of List and searches for ***deleted node***. Where **Rear** always points to the previous node of **Curr**. Once **Curr** points to the deleted node position we have to do two things, first we disconnect the link in between a pair of nodes **Rear** & **Curr**, later we have to provide a connection in between **Rear** & **Next node of Curr**. The diagrammatic representation of this deletion is



Suppose the deleted node information is **25**



Suppose is **x** the deleted node information and the List is either Ordered or Un order Liked List then the algorithm deletes the node by the following Steps.

Algorithm

```
Step1        Curr  ← LIST
Step2        Rear  ← NULL
Step3        While( Curr  ! =  NULL  &&  Info(Curr)  ! =  x)  do
               {
Step4              Rear ← Curr
Step5              Curr ← Next(Curr)
               }
Step6        If( Curr  = = NULL)
               {
                   If( Rear = = NULL)
                           Write "List is Empty & We can't delete a Node "
                   Else
                           Write " The Node does not exist in the List "
               }
             Else
               {
                   If( Rear = = NULL)
                     {
                           Write " The Deleted Node is First node "
                           LIST ← Next( LIST)
                     }
                   Else
                     {
                           Write " Deleted Node is either Middle or Last Node "
                           Next( Rear ) ← New
                           Next( New )  ← Curr
                     }
                   Write " The Deleted Node is Curr "
               }
```

In this Algorithm, initially we assign **LIST** to **Curr** and **Rear** to NULL and **Rear** Always points to the **Curr** previous node. **Curr** begins with first node and search for deleted node until either **Curr** gets **NULL** or **Curr** ***information*** matched with x value by Step**3.** Once Step**3** process completed, the deleted node pointed by **Curr** and we delete that node logically but not permanently. Since, we are only disconnecting the link in between original Linked List and **Curr (**deleted node). But **Curr** (deleted node) still alive in the memory even we disconnect the link. So it is logically deleted but not permanently. To be delete permanently we should compulsory use a notation ***freenode***(**Curr**). This ***freenode***(**Curr**) operation can free (for reuse) the node which is pointed by **Curr** and put back into available free space **Memory**. So the Last Step in this Algorithm is

Step**7** ***free***(**Curr**)

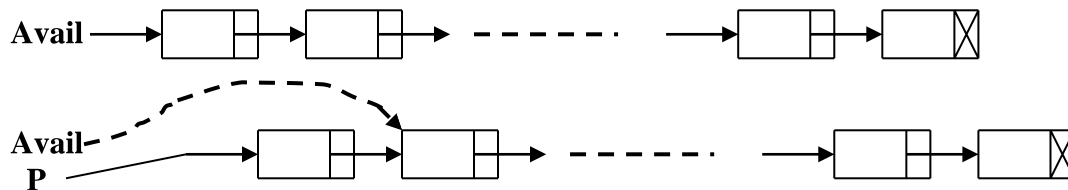
Avail List :

The List of available nodes in the **Memory** is called **Avail List**. It is a finite pool of available nodes , these available nodes are existing as a Linked list and it is pointed by an ***External*** pointer called ***Avail***. The Last node in this List is ***NULL***.

Getnode()

Getnode() supplies a node (empty node) from the **Avail List**. We can write the following algorithm to provide a new node

Step**1** If (**Avail** = = ***NULL***) then
Write " **Avail List is Empty** "
Else
{
P = **Avail**
Avail = ***Next***(**Avail**)
Return(**P**)
}

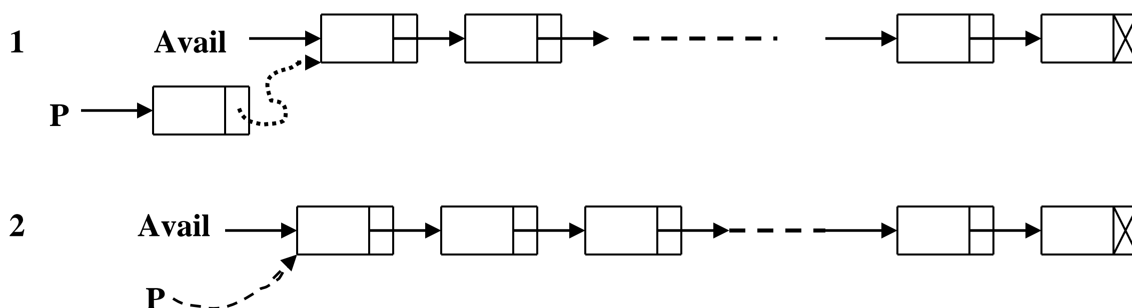


Freenode( )

In Linked List, suppose a particular node we want to be deleted, we disconnect that node link with existing List. This is only logical deletion, but it is not a permanent deletion. We can delete that node permanently by ***Freenode()***. Because this ***Freenode()*** frees the deleted node and return back to the **Avail** List. The Algorithm for ***Freenode()*** is

Step**1** ***Next*** (**P**) = **Avail**
Step**2** **Avail** = **P**

This Algorithm works pictorially in two steps



Implementation of Linked List Using Dynamic Memory Allocation

In Array implementation of Linked Lists, a fixed number of nodes represented by an Array are established at the start of execution. Therefore the disadvantage here is we have to predict the number of nodes when a program is written which is not always possible. Another disadvantage is that we have to allocate the declared number of nodes throughout the execution of the program, whereas the program may not be actually using those many number of nodes.

The solution is Dynamic allocation of nodes to the program. That is, storage must be allocated to a program only when it is required, and it must be released, when it is no longer in use. Now, we will implement the list using Dynamic memory allocation.

We can implement a list using the concept of pointers in **C**. A node of Linked list can be defined as follows.

```
struct  nodetype
   {
        int  info;
        struct nodetype   *next;
   };
typedef  struct nodetype  N;
```

Here, a node consists of an ***information*** *field* and a ***pointer*** to the next node in the list rather than an integer as in the case of Array implementation.

C has the capability of Dynamically allocating memory space, modifying the previously allotted memory space and freeing a memory space by the following Memory Allocation Functions.

Function	**Task**
malloc	Allocates requested size of bytes and returns a pointer to the first byte of the allocated space.
calloc	Allocate space for an array of elements, initializes them to zero and then returns a pointer to the memory.
Free	Frees previously allocated space.
realloc	Modifies the size of previously allocated space.

Allocating a Block of Memory

The **malloc** function reserves a block of memory of specified size and returns a pointer of type ***void***. This means that we can assign it to any type of pointer. The general format is

Ptr = (***cast-type*** *) **malloc**(***byte-size***);

Ptr is a pointer of type ***cast-type***. The **malloc** returns a pointer (of ***cast-type***) to an area of memory with size ***byte-size***.

Eg:-   ptr = (N *) malloc (sizeof(N));

Here **N** is *synonym* for struct nodetype, contain two members ***info*** (2 bytes) and ***next*** *pointer* (2 bytes) occupies totally **4** bytes. So **4** bytes reserved by **malloc** function and return the first byte address of reserved space to a pointer ptr of type **N**.

The **calloc** is another memory allocation function that is normally used for requesting memory space at run time for storing derived data type such as arrays and structures. While **malloc** allocates a single block of storage space, **calloc** allocates multiple blocks of storage, each of the same size, and then sets all bytes to zero. The general format is

Ptr = (***cast-type*** *) **calloc**(***n***, ***byte-size***);

This function allocates contiguous space for **n** blocks, each of size ***byte-size***.

Note:-

If there is not enough memory space by either **malloc** or **calloc** function, a NULL pointer is returned. So, have to check whether the requested memory is allotted or not? by The following condition:

```
if( ptr = = NULL)
  {
     printf("\n Memory is not allotted");
     exit(0);
  }
```

Altering the Size of a Block

We can change (either **increase** or **decrease**) the memory size already allocated with the help of the function **realloc**. This process is called the reallocation of memory. Suppose we allocate the Block of Memory by the following statement.

Ptr = **malloc**(sizeof(***old-size***));

The general format to increase or decrease the Memory Block is

Ptr = **realloc**(Ptr,***new-size***);

This function allocates a new memory space of size ***new-size*** to the pointer **Ptr** and returns a pointer to the first byte of the new memory block. The ***new-size*** may be larger or smaller than the ***old-size***.

Releasing the Used Space

When we no longer need the data we stored in a block of memory, and we do not intend to use that block for storing any other information, we may release that block of memory for future use, using the **free** function. The general format is

free(**ptr**);

Where **ptr** is a pointer to a memory block which has already been created by **malloc** or **calloc**.

We can, therefore, use **malloc** function in the place of ***getnode()*** function and use **free** function in the place of ***freenode()*** function

Header Node

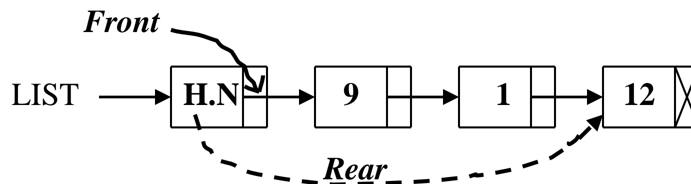
Some times it is desire to keep an extra node at the front of a List. Such a node doesn't represent an item in the List and is called a **Header Node** or a **List Header**. The ***Information field*** of such Header node might be unused but the ***Next field*** maintains the first node address. More often the information of such a node could be used to keep Global Information about the Entire List.

Eg:-

One of the applications of Header node is the information portion of the Header node contains the number of nodes (not including Header) in the List. In this Structure the Header must be adjusted the total number of nodes when we apply addition or deletion operation on the List. We can directly obtained the total number of nodes without traversing the entire List.

By using this application, we can develop Data Structure Stack easily. Since Header node information field maintains the number of elements that the Stack contains.

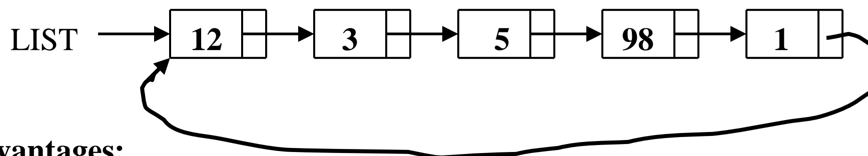
Another application is we can simply develop the Data Structure Queue. Until now, two external pointers, **Front** & **Rear**, were necessary for a List to represent a Queue. However, now only a single external pointer is sufficient to maintain a Queue that is Head node. Because the Header node next field behave like a **Front** and information field of Header node maintains last node address so that it is behave like a **Rear** .



Another possibility for the use of the information portion of a List Header is as a pointer to a **Curr** node in the List during a traversal process. This would be eliminate the node for an external pointer during traversal.

CIRCULAR LINKED LIST

In a Circular Linked List, instead of ***NULL*** pointer to the Last node of a List, it contains the address of its First node. In this List, starting from any given node, we can traverse all nodes merely by chaining throughout the List. When we are traversing a Circular List, we must be careful, as there is a possibility to get into an infinite loop if we can't able to detect the end of the List. To overcome this problem we keep an External pointer at the starting node and look for this External pointer as a stop sign. Actually, there is no First or Last node since the Structure is Circular. However, the External pointer continues to represent the List pointing to the First node.



Advantages:

1. Any node can be visited (traversed) starting from any other node in a List.
2. There is no need of NULL pointer to signal the end of the List and, hence all pointers contain valid addresses

Disadvantages:

To visit previous node from the current node, we must take one complete cycle.

DOUBLY LINKED LIST

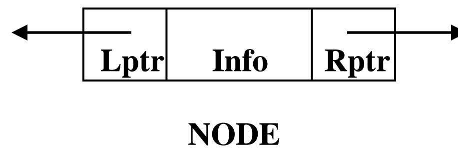
In a Doubly Linked List, each node contains two pointers (***previous*** & ***next***), one to its predecessor and another to its successor. i.e. each node contain the address of next node by ***next field*** and also address of previous node by ***previous field***. The ***previous*** and ***next*** pointers are often mentioned as ***prev*** & ***next*** or ***lpter***(left pointer) & ***rptr***(right pointer) respectively in a Structure definition. The Left pointer of First node and Right pointer of Last node always ***NULL***.

In this List we can traverse or search the entire List in both directions (Backward & Forward). That means, from any node in the List, we can visit any other node in the List. To reach Last node of the List from any other node, we simply follow the next pointers. To reach First node of the List from any other node, we follow the previous pointers.

The Structure Definition is:

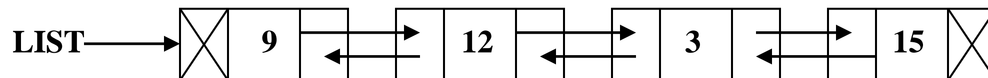
```
Struct  doubly_list
   {
        char   info;
        struct doubly_list   *lptr;
        struct doubly_list   *rptr;
   };
```

The Diagrammatic representation of Doubly Linked List Node is:



NODE

The Diagrammatic representation of Doubly Linked List



Advantages:

1. We can traverse the entire List in both directions.
2. In this List, there no need to refer to a key(node to be deleted) by using a pointer to the previous node at the time of deletion

Disadvantages:

1. When compare to single Linked List, the extra pointer in doubly Linked List occupies additional space and also increase the maintenance of insertions & deletions because there are more pointer to adjust.

TREES

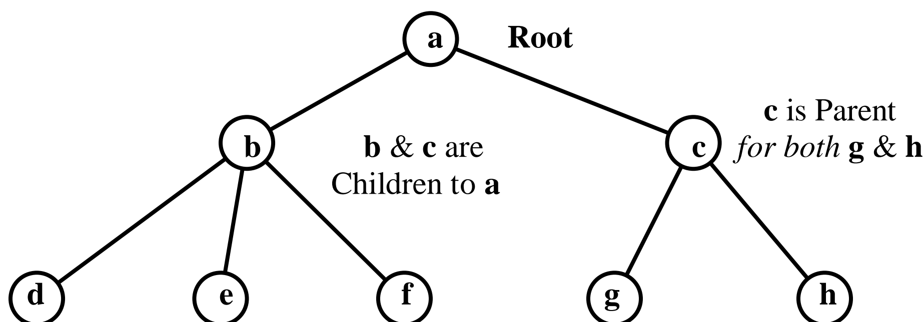
A tree is a finite set of one or more nodes such that there is a *specially designated node* called the **Root**. The remaining nodes are partitioned into **N >= 0** disjoint sets $\mathbf{T_1, T_2, \ldots T_N}$, where each of these sets is also a Tree. $\mathbf{T_1, T_2, \ldots T_N}$ are called sub trees of the Root.

A tree is a data Structure in which all the nodes are connected while directed *arcs*.

A Tree consists of one or more nodes (*vertices*), which are connected by branches (*edges*). Each node can have the data and its associated information. A tree with **N** nodes has **N-1** branches.

A Tree contains a unique First element known as the **Root**, which is shown at the top of the tree Structure.

A node, which points to other nodes in a tree is said to be ***Parent*** node. Each node in a tree pointed by parent is called ***Children***. The root of the tree is a parent of all the other elements in a tree.



In this tree there are ***8 nodes***, that's why it contains ***7 branches***.

A node may or may not have children. A node, which doesn't have any child, called the ***Leaf*** or ***Terminal*** node. These Leaf nodes are some times known as ***External*** nodes, Non-Terminal nodes are known as ***Internal*** nodes.

The child nodes of same parent are said to be ***Siblings***. All these are placed at the same level. All nodes at a particular level are said to be a same ***generation***.

A node is an ***ancestor*** of another node in the tree if it is the parent of that node or the parent of some other ancestor of that node.

A node is said to be ***descendent*** of another node if it is the child of that node or the child of some other descendent of that node.

A ***path*** in a Tree is a list of distinct nodes in which successive nodes are connected by branches in the Tree. The ***length*** of a particular path is the number of branches in that *path.*

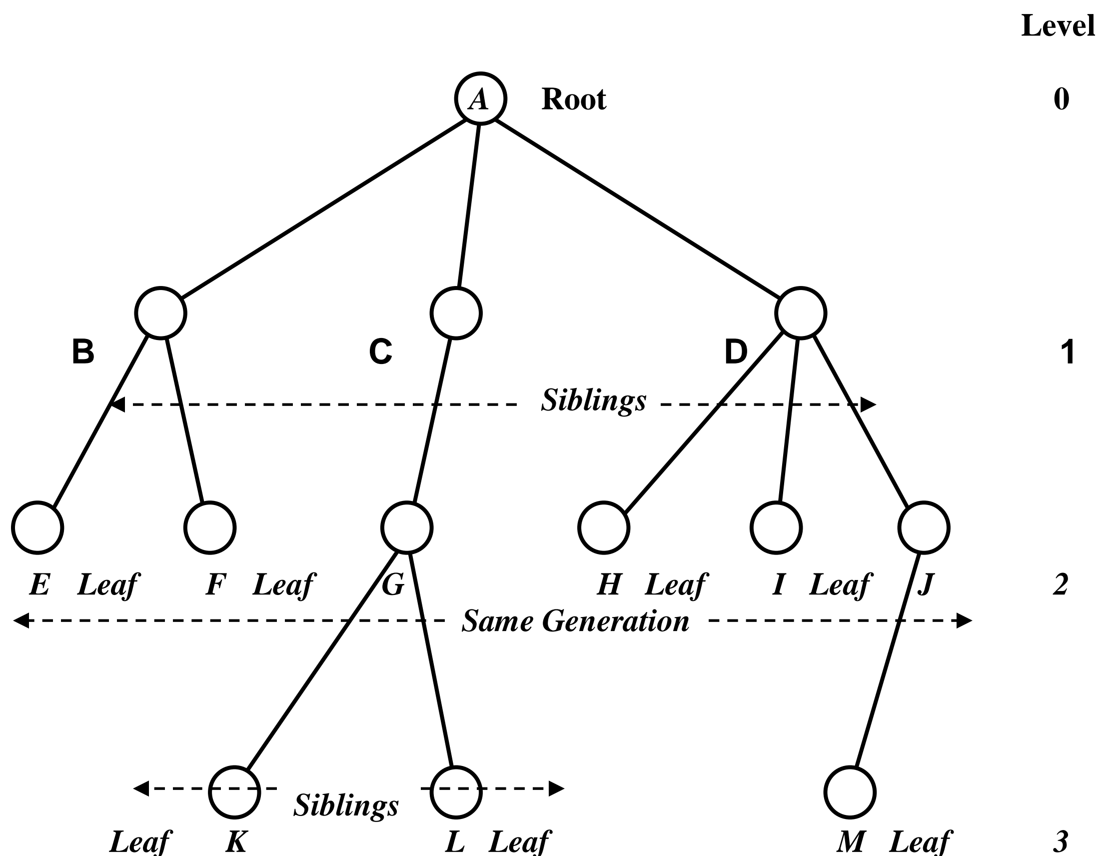
The ***Degree*** of a node of a tree is the number of children of that node. The Degree of Leaves is Zero.

The maximum number of children a node can have is often referred to as the ***Order*** of a Tree. Note that the ***Degree*** is referred for a **node** and ***Order*** is referred for entire **Tree**.

The ***height*** (***depth***) of a tree is the Maximum level of a Tree.

A ***sub tree*** is a sub set of a Tree that is itself a Tree.

A ***Forest*** is a set of **N** >= **0** disjoint Trees. The notation of a ***forest*** is very close to that of a Tree because if we remove the Root of a Tree we get a Forest. For example in the given Tree if we removed Root node that contain information **A** then we get a ***Forest*** with 3 Trees **B, C, D** respectively.



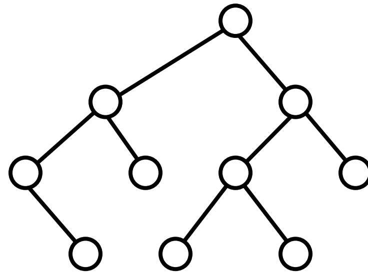
BINARY TREE

A ***Binary Tree*** is a tree in which each node has at most two children, a Left child and a Right child. Thus, the order of Binary Tree is **2**.

A ***Binary Tree*** is either empty or consists of the following steps

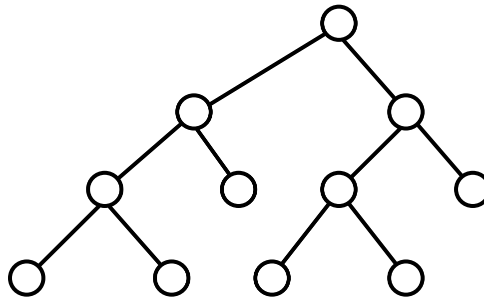
1. A node (called the **Root** node) and
2. Left and Right Sub trees

Note:- Both the sub trees(left & right) are themselves Binary Trees.

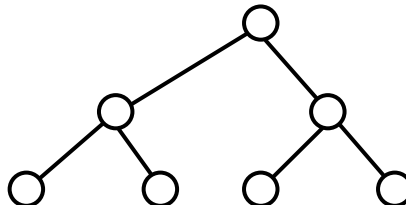


Note:- Binary Trees are not symmetric structures. i.e interchange of Right and Left sub trees results a new different Binary Tree.

A Binary Tree is called ***Strictly Binary*** Tree if every non-leaf node in the Binary Tree has non-empty Left and Right sub trees.



A Binary Tree is called ***Complete Binary*** Tree if every non- leaf node has left and right sub trees and all these leaves are at the same level.



Binary Tree Representation:

A Binary Tree can be represented in two ways.

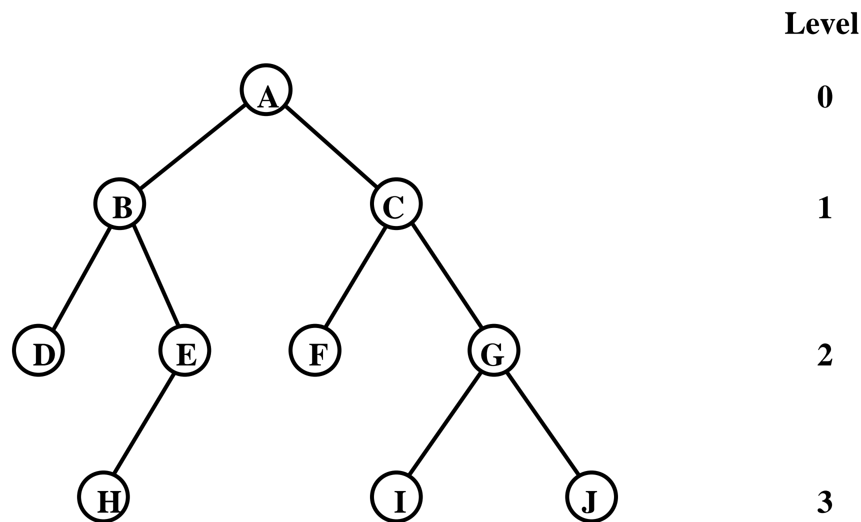
1. Sequential (Array) representation.
2. Linked (Pointer) representation

Implementation of Binary Trees using Arrays.

We can implement Binary Trees in arrays using one-dimensional array.

a. The Root of Binary Tree is stored in the First location (**0th** index)
b. If a node is in **Jth** location (**Jth** index)of Array then its left child is stored in the **2Jth** location (**2J +1** index) and its right child is stored in the **2J+1** location (**2J+2** index).

Suppose if a Tree conations a depth **D** then, the size of the array to store the tree nodes is equal to $2^{D+1} - 1$.



The above Tree depth is **3**, So the number of nodes we required is **15**

A	B	C	D	E	F	G			H				I	J
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**

Advantages:

1. No overhead of maintaining any pointers.
2. We can easily identifies the Parent node for a specified child i.e the Parent is
   located at (***children location / 2***) area. Here the division is integer division.

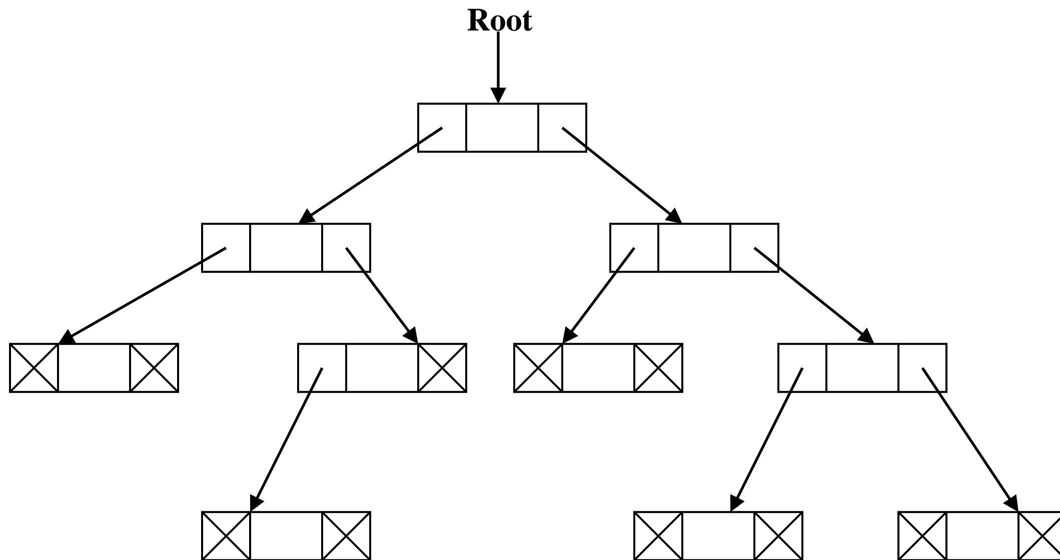
Note:- For Indexes, Parent location = (child location – 1) / 2

Disadvantages:

1. Growing and shrinking of a tree can't efficiently manage.
2. Wastage of Memory, when the array is not properly filled.

Linked Representation:

Linked representation of Trees in memory is implemented using pointers. Since each node in a Binary Tree can have maximum two children, anode in a linked representation has two pointer fields for both Left and Right child, and one Information field. If a node does not have any child, the corresponding pointer field is ***NULL*** pointer. The following figure shows the Linked representation




```
struct  tree_node
        {
            char   info;
            struct  tree_node   *left, *right;
        };
```

A Binary Tree with N nodes contains N+1 Null Pointers. The above Tree has **10** Nodes, so it contains **11** ***NULL*** pointers

Disadvantages:

1. Wasted space due to Null Pointers at all the Leaf nodes.
2. Finding a Parent of a particular node is difficult.

Binary Search Tree:

A Binary search tree is a Binary tree, which is either empty or contains a node whose key satisfies the following conditions:

1. The key in the Left child of a node (if any) precedes the key in the Parent node.
2. The key in the Right child of a node (if any) succeeds the key in the Parent node.
3. The Left and Right sub trees of the Root are again Binary search Trees.

Note:- The given assumptions are for only non-duplicate values. To accommodate the duplicate values we can modify the definition so that the key in the Right child of a node can be greater than or equal to the key in the Parent node.

Algorithm for search information* X *in the Binary Search Tree

```
P ← Root
While ( P != NULL ) and ( info(P) != X ) do
        If ( info(P) > X ) then
                P ← Left( P )
        Else
                P ← Right( P )
If ( P = = NULL ) then
        Write " X is not Present in the Tree."
Else
        Write " X is Present in the Tree."
```

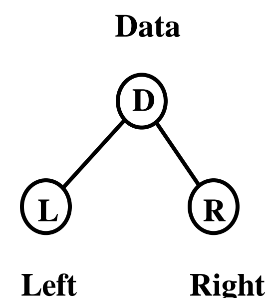
Traversing of Binary Tree:

Traversing a Tree means that processing it so that each node is visited exactly once. A Binary Tree can be traverse in number of ways. The most common three Traversals are In-Order, Pre-Order and Post-Order.

Recursive Procedures for In-Order, Pre-Order and Post-Order traversals are:

Pre-Order:

Step1: Visit the Root of a Tree
Step2: Pre-Order Traverse of Left Sub Tree
Step3: Pre-Order Traverse of Right Sub Tree.



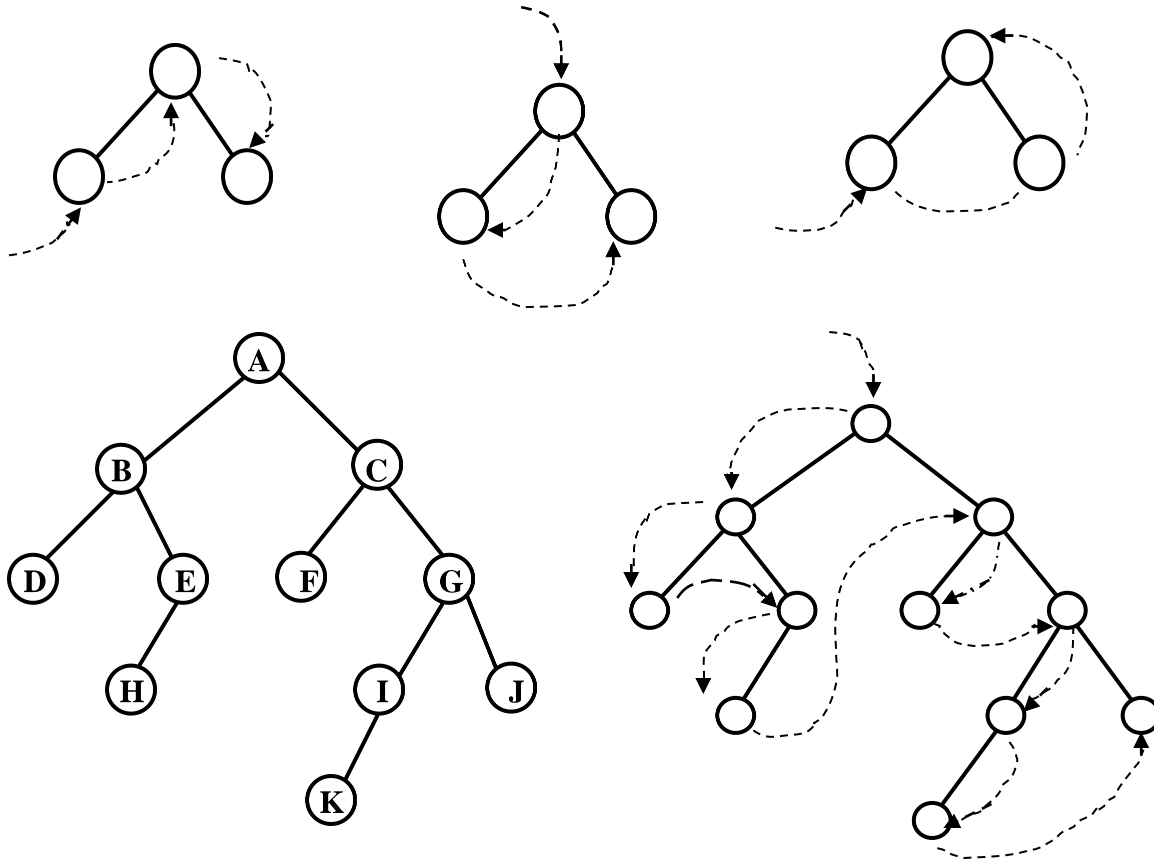
In-Order:

Step1: In-Order Traverse of Left Sub Tree
Step2: Visit the Root of a Tree.
Step3: In-Order Traverse of Right Sub Tree.

Post-Order:

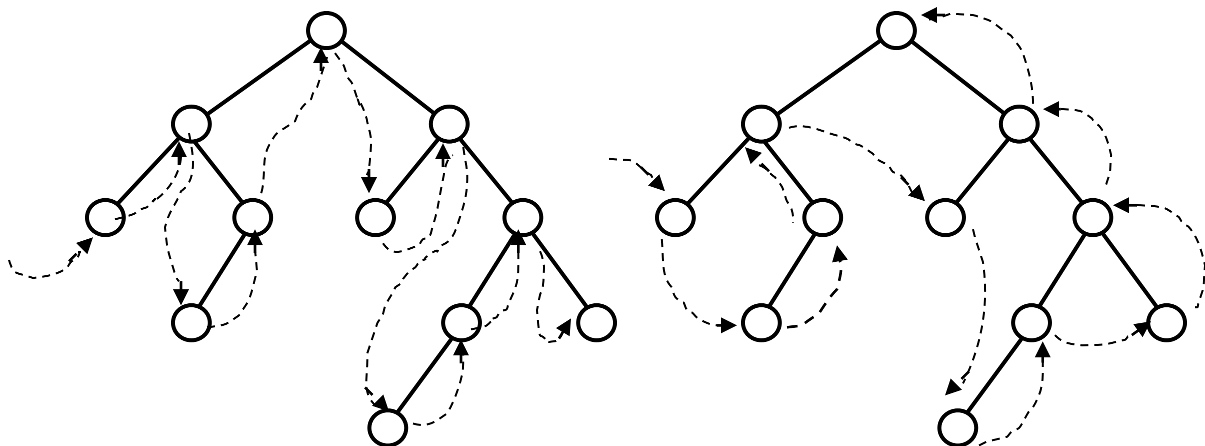
Step1: Post-Order Traverse of Left Sub Tree
Step2: Post-Order Traverse of Right Sub Tree.
Step3: Visit the Root of a Tree.

In-Order Traverse **Pre-Order Traverse** **Post-Order Traverse**



Sample Binary Tree

Pre-Order Traversal Binary Tree
A, B, D, E, H, C, F, G, I, K, J



In-Order Traversal Binary Tree
D, B, H, E, A, F, C, K, I, G, J

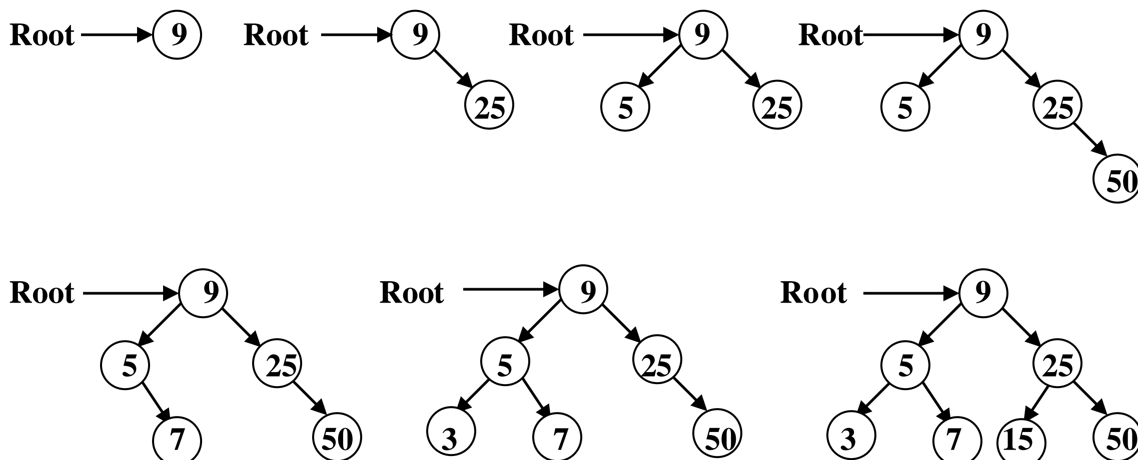
Post-Order Traversal Binary Tree
D, H, E, B, F, K, I, J, G, C, A

Algorithm to insert a new node onto Binary Search Tree

Step1: ***Getnode*(New)**
Step2. ***Left*(New) ← *NULL***
Step3. ***Right*(New) ← *NULL***
Step4. ***Info*(New) ← X**
Step5. **Curr ← Root**
Step6. **Rear ← *NULL***
Step7. While(**Curr != *NULL*) do**
    {
        Rear ← Curr
        If (***Info*(Curr)** > **X**) then
            Curr ← *Left*(Curr)
        Else
            Curr ← *Right*(Curr)
    }
Step8. If (**Rear** = = ***NULL***) then
        Root ← New
    Else
    {
        If (***Info*(Rear)** > **X**) then
            ***Left*(Rear) ← New**
        Else
            ***Right*(Rear) ← New**
    }

In a Binary Search Tree, the New node will always be inserted at its proper position in the Binary Search Tree as a Leaf Node. Let us take an empty Binary Search Tree, and the following diagrams represents the state of Binary Tree when we insert the data **9, 25, 5, 50, 7, 3, 15** in a sequential order.

Initially, the Binary Tree State is **Root** ⟶ ***NULL***

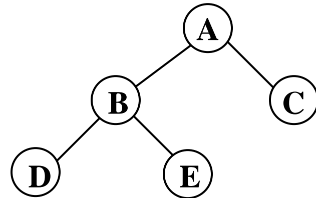


Delete a Node from the Binary Search Tree:

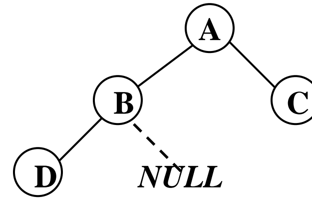
The method to delete a node depends on the specific position of the node in the tree. The algorithm to delete a node can be subdivided into **4** different cases.

***<u>Case 1</u>*:**

If the node to be deleted is a Leaf node, we only need to set the appropriate link of its Parent to Null, and dispose of the node, which is deleted.



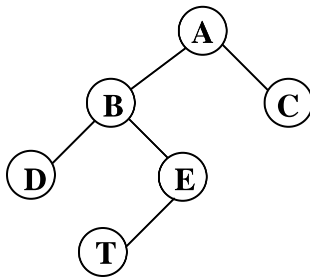
Suppose, the Node to be Deleted is **E**

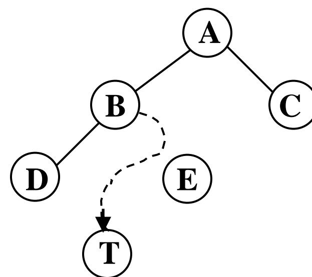


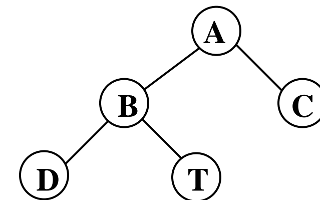
After Delete the Node **E**

***<u>Case 2</u>*:**

If the node to be deleted has only Left child, we need to adjust the Link from the Parent of the deleted node to point to the Left child of the node we intend to delete. Thus we can dispose of the deleted node.





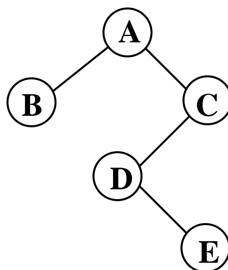


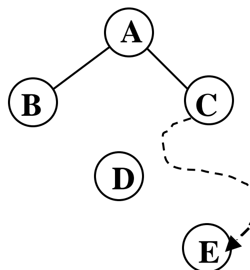
Suppose, the Node to be Deleted is **E**

After Delete the Node **E**

***<u>Case 3</u>*:**

If the node to be deleted has only Right child, we need to adjust the Link from the Parent of the deleted node to point to the Right child of the node we intend to delete. Thus we can dispose of the deleted node.



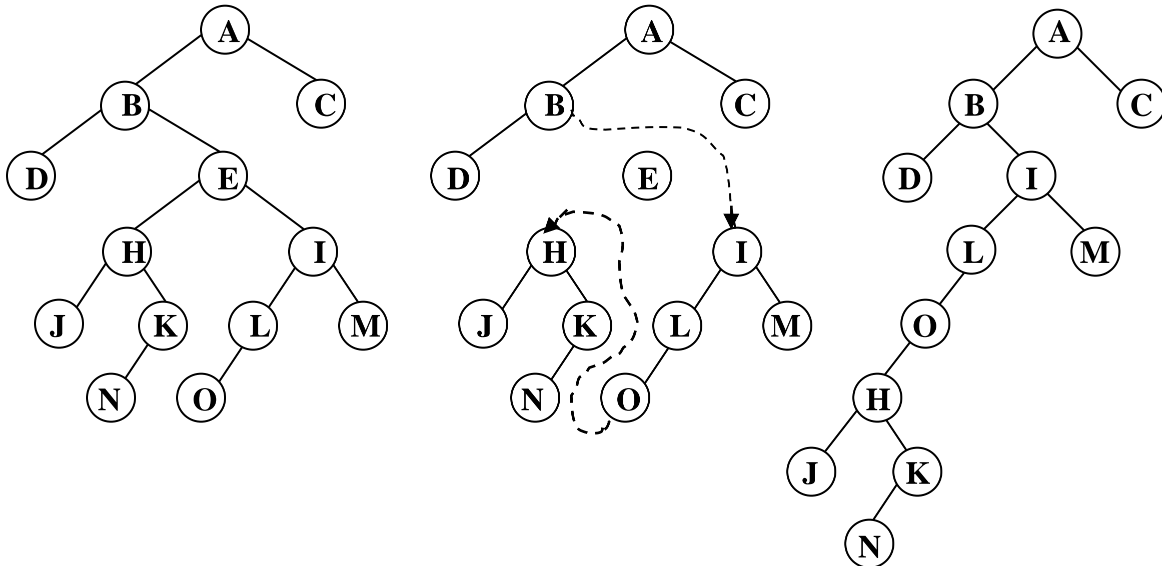


Suppose, the Node to be Deleted is **D**

After Delete the Node **D**

***Case* 4:**

If the node to be deleted has two children is most complicated problem. We attach one of the sub trees of the nodes to be deleted to the parent and then hang the other sub tree onto the appropriate node of the first sub tree. Let us attach right sub tree to the parent node and then hang the left sub tree onto a proper node of the right sub tree. Since every key in the left sub tree is less than every key in the right sub tree. Therefore, we must attach the left sub tree as far to the left as possible. This proper place can be found by going left until an empty left sub tree is found.



Suppose, the Node to be Deleted is **E** After Delete the Node **E**

Algorithm to delete a Node from the Binary Search Tree

Initialize the traversal pointers

Step1. **Curr** ← **Root**

Step2. **Rear** ← **NULL**

Search for the deletion place

Step3. While (***Info***(**Curr**) **!= X** **&&** (**Curr** **!=** ***NULL***))

{

Rear ← **Curr**

If (***Info***(**Curr**) > **X**)

Curr ← ***Left***(**Curr**)

Else

Curr ← ***Right*** (**Curr**)

}

Step4. If(**Curr** = = ***NULL***)

Write " We are trying to delete non existing node."

Else if (***Left***(**Curr**) **!=** ***NULL*** **&&** ***Right***(**Curr**) **!=** ***NULL***)

{

Temp ← ***Right*** (**Curr**)

```
                while( Left(Temp ) != NULL )
                        Temp <- Left (Temp )
                Left (Temp) <- Left(Curr)
                Connect <- Right(Curr)
            }
        Else if ( Left(Curr) = = NULL && Right(Curr) = = NULL )
            {
                Connect <- NULL
            }
        Else if ( Left(Curr) != NULL && Right(Curr) = = NULL )
            {
                Connect <- Left (Curr)
            }
        Else
            {
                connect <- Right (Curr)
            }
Step5.  If (Rear = = NULL )
                Root <- Connect
        Else if ( Info(Rear) > Info(Curr ) )
                Left(Rear) <- Connect
        Else
                Right(Rear) <- Connect
Step6.  Freenode(Curr)
```

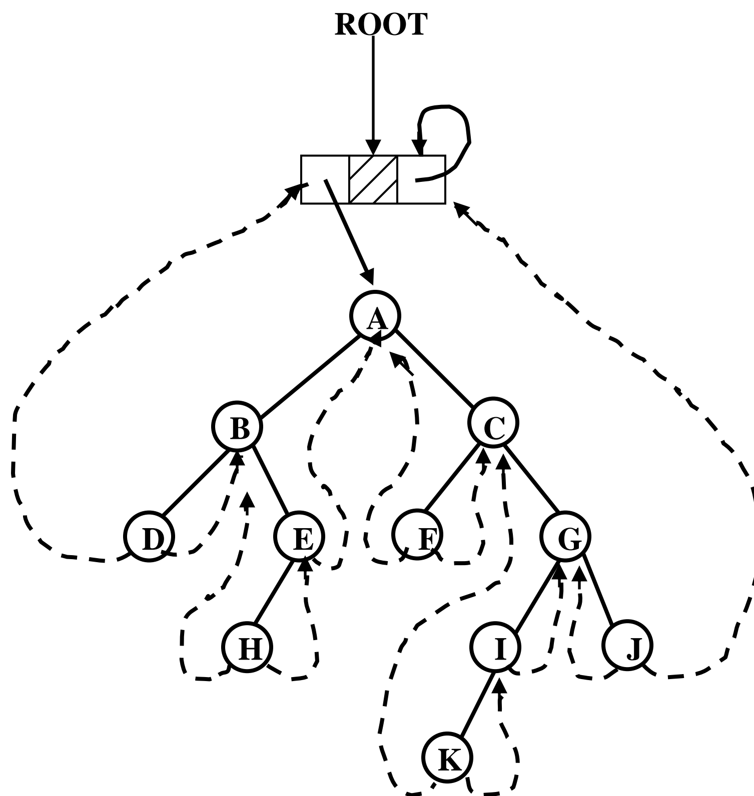
Treaded Binary Tree

To traverse a tree in a particular order, we have to know how to access the next node in that order. Moreover, we have seen that the linked representation of Binary Tree has a drawback of having NULL pointers at several nodes. This space can be utilized efficiently by replacing the values with ***threads***. Threads are nothing but pointers which contain some additional useful information.

Generally, the threading of a Binary Tree corresponds to a particular traversal order. For example, for In-Order traversal of a threaded tree, if the right pointer of a node was originally NULL, now it points to the In-Order successor of that node in the tree. Similarly, if the left pointer of the node was originally NULL, it now points to In-Order predecessor of that node. However the left pointer of first node and right pointer of last node will contain the NULL value.

Trees can also have a special node, called header node, as we have seen with Linked Lists. That is, in a tree which has header node, the pointer **ROOT** points to header node rather than the Root node. If a header node is there in a threaded binary tree, the left pointer of first node and right pointer of last node, with respect to In-Order

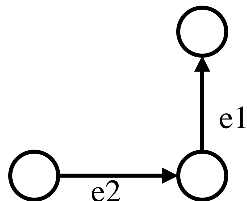
traversal. The following diagram represents the Threaded Binary Tree. Threads are generally indicated by dotted lines.



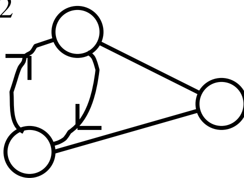
GRAPH THEORY

A graph **G = (V , E)** consists of a non-empty set, **V** called the set of ***nodes*** or ***vertices*** or ***points*** and a set of ***edges*** **E** in which each edge is associated to an ordered pair of nodes or unordered pair of nodes.

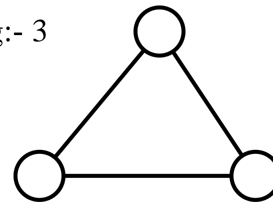
Eg:- 1



Eg:- 2

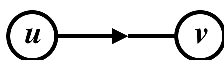


Eg:- 3



Ordered Pair of Nodes:

A pair of nodes ***u*** and ***v*** in a graph is called an **ordered pair** of nodes if ***u*** is connected to ***v*** with some directions. Ordered pair of nodes is denoted by **< *u* ,*v* >**

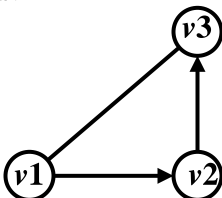


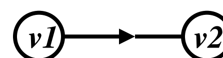
Unordered Pair of Nodes:

A pair of nodes in a graph ***u*** and ***v*** is called **unordered pair** of nodes if ***u*** is connected to ***v*** and without any direction. These are denoted by ***(u , v)***

Adjacent nodes:

A pair of nodes in a graph which are connected by an edge are called **adjacent nodes**.

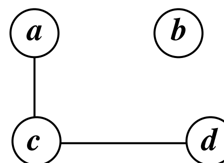




Insolated Node:

A node in a graph, which is not connected any one of the other nodes is called an isolated node.

.

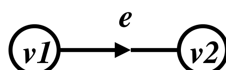


In this graph, ***x*** & ***y*** are Isolated Nodes

In this graph, ***b*** is Isolated Node.

Directed edge:

An edge in a graph is called a directed edge if it is associated to a pair of ordered nodes.



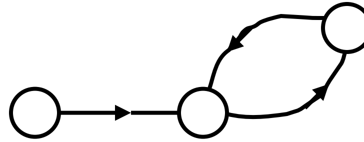
Here, **e** is ***directed edge***

Undirected edge:

An edge in a graph is called undirected edge if it is associated to unordered pair of nodes.

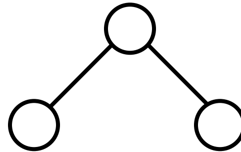
Directed graph:

A graph in which every edge is a directed edge is called a directed Graph.



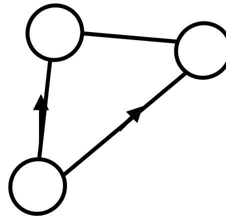
Undirected graph:

A graph is called undirected if each edge is not a direct edge.



Mixed Graph:

A graph in which some edges are directed and some edges are undirected is called a mixed graph.



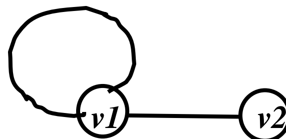
Consisting only one-way street in a map of a city is called directed graph. In a Map of a city consisting of two- way streets only is called an un-directed graph. In a Map of a city consisting of two- way and one- way streets is called Mixed Graph.

Incident:

In a graph if an edge ***e*** is associated to an ordered or unordered pair of nodes ***u*** and ***v*** then we say that the ***e*** is incident with ***u*** and ***v***.

Loop or Sling:

In a graph an edge connecting a node to it self is called a loop or sling.

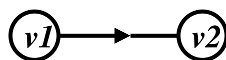


Initiating or originating Node:

In a Dia-graph a node at which an edge starts or begins is called initiating or originating node.

Terminating Node:

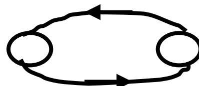
In a Dia-graph a node at which an edge ends is called terminating node.



Here, ***v2*** is terminating node

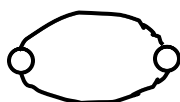
Distinct Edges:

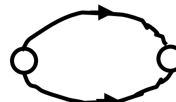
In a diagraph if two nodes are connected in two opposite directions the corresponding edges are called distinct nodes.



Parallel edges:

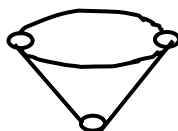
In a graph if two edges are associated to the some ordered or unordered pair of nodes then those edges are called parallel edges.





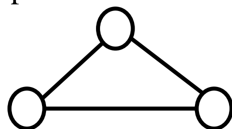
Multi graph:

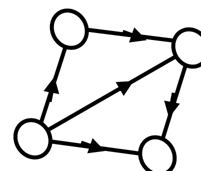
A graph is called a multi graph if it contains some parallel edges.



Simple Graph:

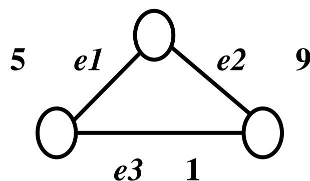
If in a graph there is no more than one edge between any pair of nodes we say that the graph is a simple graph.





Weighted Graph:

A graph in which weights are assigned to each edge is called a weighted graph.



e1 ***weight*** is **5**

e2 ***weight*** is **9**

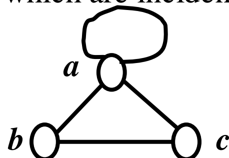
e3 ***weight*** is **1**

Eg1. A graph representing a system of pipelines in which the weights are assigned indicates the amount of some commodity transferred through the pipe is an example for a weighted graph.

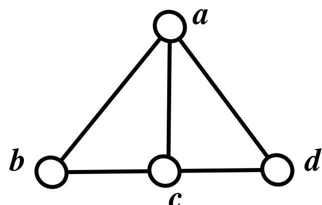
Eg2. Map of a city in which each street is assigned weights according to the traffic to the density is weighted graph.

Degree of a Node:

Number of edges, which are incident to a node and self - loop counted twice is called a degree of a node.



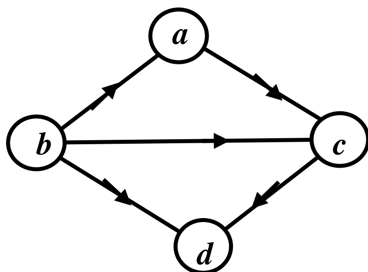
Degree of node ***a*** is **4**
Degree of node ***b*** is **2**
Degree of node ***c*** is **2**



Degree of node ***a*** is **4**
Degree of node ***b*** is **2**
Degree of node ***c*** is **3**
Degree of node ***d*** is **2**

Out Degree of a Node:

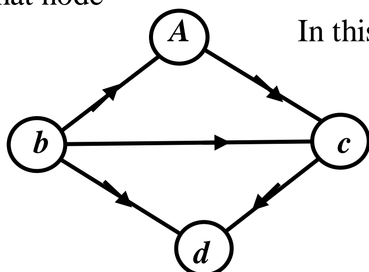
The no. of edges originating at a node in a Dia-graph is called the out-degree of that node.



In this graph, ***a*** out-degree is **1** , ***b*** out-degree is **3**
c out-degree is **1** , ***d*** out-degree is **0**

In Degree of a Node:

The no. of edges terminating at a node in a Dia-graph is called the in-degree of that node

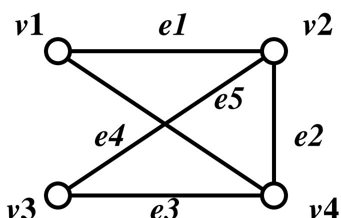


In this graph, ***a*** in-degree is **1** , ***b*** in-degree is **0**

c in-degree is **2** , ***d*** in-degree is **2**

Path in an undirected graph:

Path in an undirected graph is a sequence of edges in which any two successive edges have a common node.



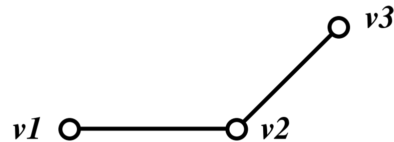
e1, e2 and ***e3*** is a **path.**

e1, e2, e3 and ***e4*** is a **path.**

e1 and ***e5*** is a **path.**

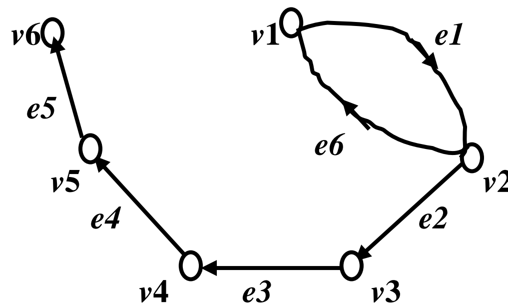
Simple Path:

A path in a graph is called a ***Node Simple*** if it travels to distinct nodes.



Path in a diagraph:

In a diagraph path is a sequence of edges in which terminal node of an edge e is same as initial node of its successive edge. ***e1, e2, e3, e4*** and ***e5*** is a **path.**



Length of a Path:

In a directed or undirected graph the length of a path is the no. of edges in a path.

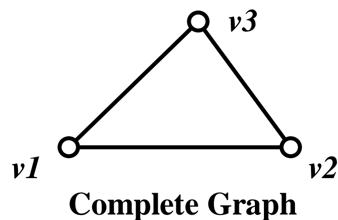
For example the length of the path from *node* ***v3*** to *node* ***v6*** is **3**.

Edge Simple:

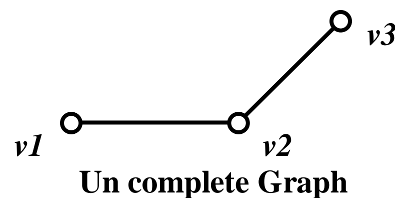
A path is called an edge simple if all the edges in the path are distinct.

Complete undirected graph:

An undirected graph is said to be complete if there is an edge common to each pair of nodes.



Complete Graph

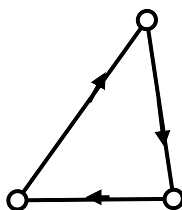


Un complete Graph

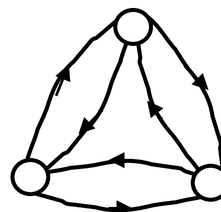
Complete Diagraph:

A diagraph is said to be complete if there is an edge common to each pair of nodes. A diagraph is complete if there us an edge from ***u*** to ***v*** for each pair **< *u* , *v* >**

Un complete Graph



Complete Graph



Graphs Representation

To represent a graph we have to represent two things: nodes and edges. Graphs are generally represented either in sequential representation or linked representation. Sequential representation uses a two-dimensional array where as linked representation uses a linked list. We refer graph means directed graphs unless specified otherwise.

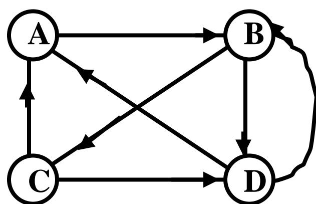
Sequential Representation

A graph is conveniently represented by a matrix (two-dimensional array) called adjacency matrix(or incidence matrix). A graph containing n nodes can be represented by a matrix containing ***n*** rows and ***n*** columns.

Adjacency matrix

The Adjacency matrix **A** of graph **G =(V, E)** with n nodes is an ***n x n*** matrix such that: $\mathbf{A_{ij}} = \mathbf{1}$ if there is an edge between $\mathbf{v_i}$ and $\mathbf{v_j}$, otherwise $\mathbf{A_{ij}} = \mathbf{0}$

Essentially adjacency matrix represents which vertices are adjacent, or rather, which two vertices have an edge. For example, the given bellow figure shows directed graph and its adjacency matrix.



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

Each position of adjacency matrix represents whether one node is connected to another (value 1 or true) or not (value 0 or false). Note that it encodes the direction of the edges. For example, in adjacency matrix there is **1** in ***row* B**, ***column* A**, since there is an edge from **B** to **A**. But there is no edge in the reverse direction from ***row* A** to ***column* B**, so that in adjacency matrix the correspondent position (***row* A** , ***column* B**) contains **0**.

Note that if a matrix contains only **0**s and **1**s then it is called bit matrix or Boolean matrix. The total number of **1**s in the matrix represents the *total number of edges* in that directed graph.

The adjacency matrix for a undirected graph is a symmetric matrix. Because an edge (A, B) means that both (A, B) and (B, A).

An adjacency matrix has certain disadvantages:

(a) Graphs with few edges would have a lot of wasteful zeroes in the adjacency matrix. That is, the corresponding adjacency matrix is sparse.
(b) Insertion and deletion of nodes is difficult

Linked Representation

The linked representation, using linked lists, maintain two kinds of lists:

1 A **NODE LIST** and

2 An **EDGE LIST**.

NODE LIST

In Node List, each node corresponds to a node in Graph **G**, and it contains the following three fields:

(a) ***NODE*** – contains the value of the node in the graph

(b) ***NEXT*** - contains a pointer that points to next node in the node list and

(c) ***ADJACENT*** - contains pointer that points to the first node of adjacency list of the node

Each node in the node list can have other fields, which may contain the information associated with the node.

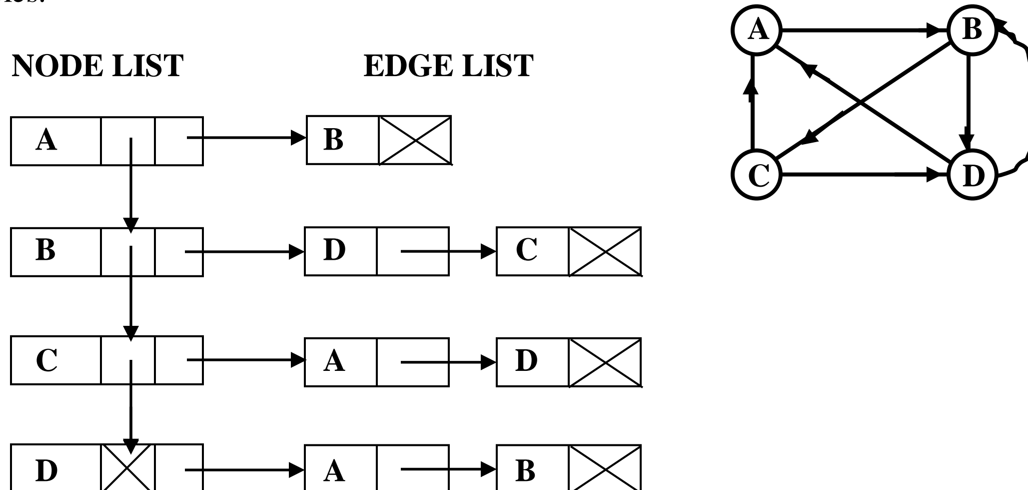
EDGE LIST

In Edge List, each node corresponds to an edge of graph **G**. A node in the edge list contain two fields:

(a) **TERMINAL** – contains a node that is an adjacent node to a node in the **NODE LIST**

(a) **PTR** – contains a pointer that points to next node in the adjacency list, which are all initiated from same node in the graph.

Like node list, edge list also may have any other information. One such important field is the weight field, if an edge is associated with weight. In the adjacency list representation, each edge appears in two lists and thus the space usage essentially doubles.



Graph Traversal

Traversal of a graph means that systematically visiting all nodes exactly once in the graph. The two important traversal methods are **breadth-first search** and **depth-first search.** The graph traversal start at an arbitrary vertex since there is no node as special. Assume that each node in a graph will be in one of three states while traversing the graph***: ready*** **state*,*** ***waiting*** **state*,*** ***visited*** **state.**

Breadth-First Traversal

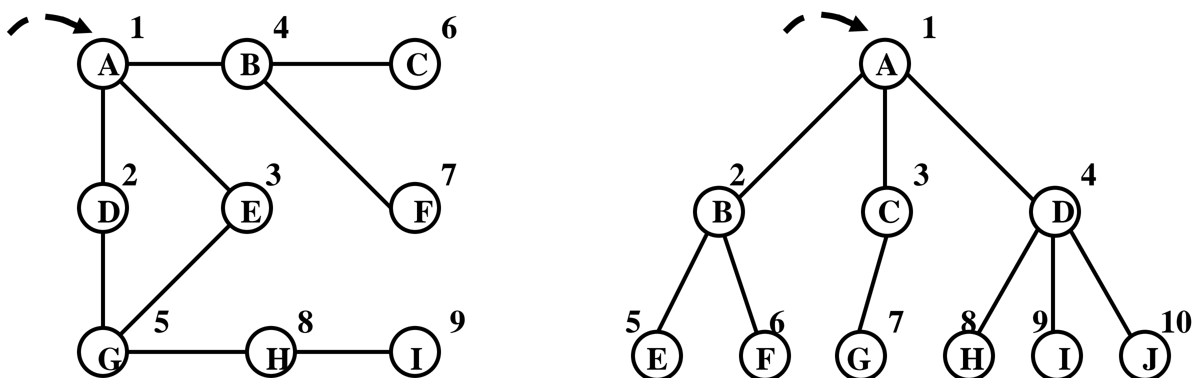
This strategy is much similar to level-by-level traversal of an ordered tree. Breadth-first search operates by processing nodes in layers. The breadth-first search can begin at any arbitrary node. The nodes, which are adjacent to the start node, are processed first, and proceeds to adjacent nodes of that nodes just visited. This process starts until all the nodes are visited. Here, if the traversal just visited a node **A**, then it next visits all the nodes adjacent to **A**, keeping the nodes adjacent to these in ***waiting*** list to be traversed after all nodes adjacent to **A** have been visited.

A data structure **Queue** is used to place all waiting nodes. This queue is also convenient to keep the track of nodes that are already visited, so that, a node is visited only once.

The general Breadth-first traversal algorithm is as follows:

Step**1**. All nodes are initialized as *ready* state and initialize **Queue** to empty
Step**2**. Begin with any node, which is in ready state and put into Queue. Mark the status of that node to *waiting*.
Step**3**. While Queue is not empty do
{
Step**3.1**. Delete the first node ***K*** from Queue and process it. Mark the status of that node to *visited*.
Step **3.2**. Add all the adjacent nodes of ***K,*** which are in ready state to the rear side of the Queue and mark the status of those nodes to *waiting*.
}
Step**4**. If the graph still contains nodes, which are in *ready* state then go to step**2**.
Step**5**. Return

The step**6** in the above algorithm handle those nodes, which does not have a path from starting node to them.



The above diagrams show the visiting nodes in **BFS**. Numbers on the nodes indicates the sequence of nodes visited in **BFS**.

Depth-first Traversal

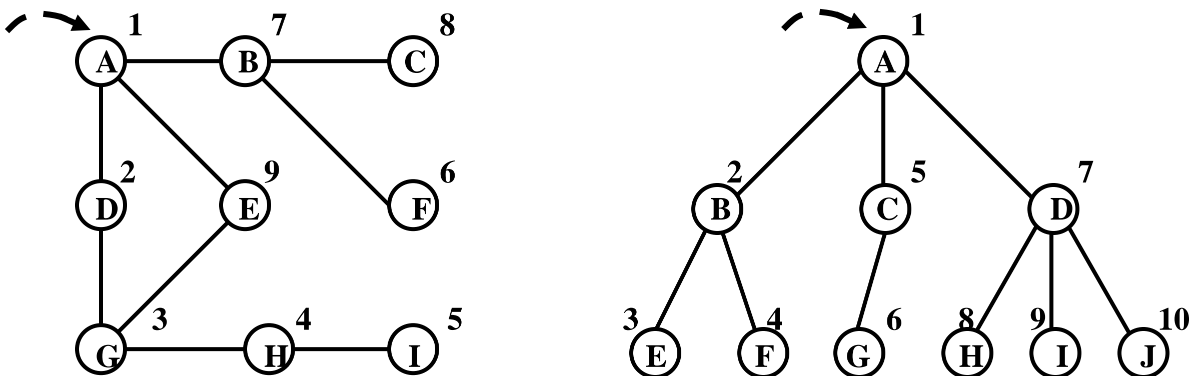
The depth-first traversal of a graph is much similar to preorder traversal of an ordered tree. The traversal of a graph start at any arbitrary node, say **A**. Suppose **B, C, D** and **E** be the nodes adjacent to **A**. Then, we will next visit **B** and keep **C, D** and **E** waiting. After visiting **B**, we traverse all the vertices to which it is adjacent before returning to traverse **C, D** and **E**.

In the depth-first traversal, we backtrack on a path once it reached the end of that path. We consider the data structure **Stack** instead of a queue as in breadth first traversal.

The general Depth-first traversal algorithm is as follows:

Step**1**. All nodes are initialized as *ready* state and initialize **Stack** to empty
Step**2**. Begin with any node, which is in ready state and push into Stack. Mark the status of that node to *waiting*.
Step**3**. While Stack is not empty do
{
Step**3.1**. Pop the top node ***K*** from Stack and process it. Mark the status of that node to *visited*.
Step **3.2**. Push all the adjacent nodes of ***K,*** which are in ready state in to the the Stack and mark the status of those nodes to *waiting*.
}
Step**4**. If the graph still contains nodes, which are in *ready* state then go to step**2**.
Step**5**. Return

Note that, the depth-first traversal continues progressively deeper (downwards) in a recursive manner.



The above diagrams show the visiting nodes in **BFS**. Numbers on the nodes indicates the sequence of nodes visited in **BFS**.

Spanning Tree

There is a common problem associated with weighted graphs, that of finding a *minimal spanning* tree for each connected component.

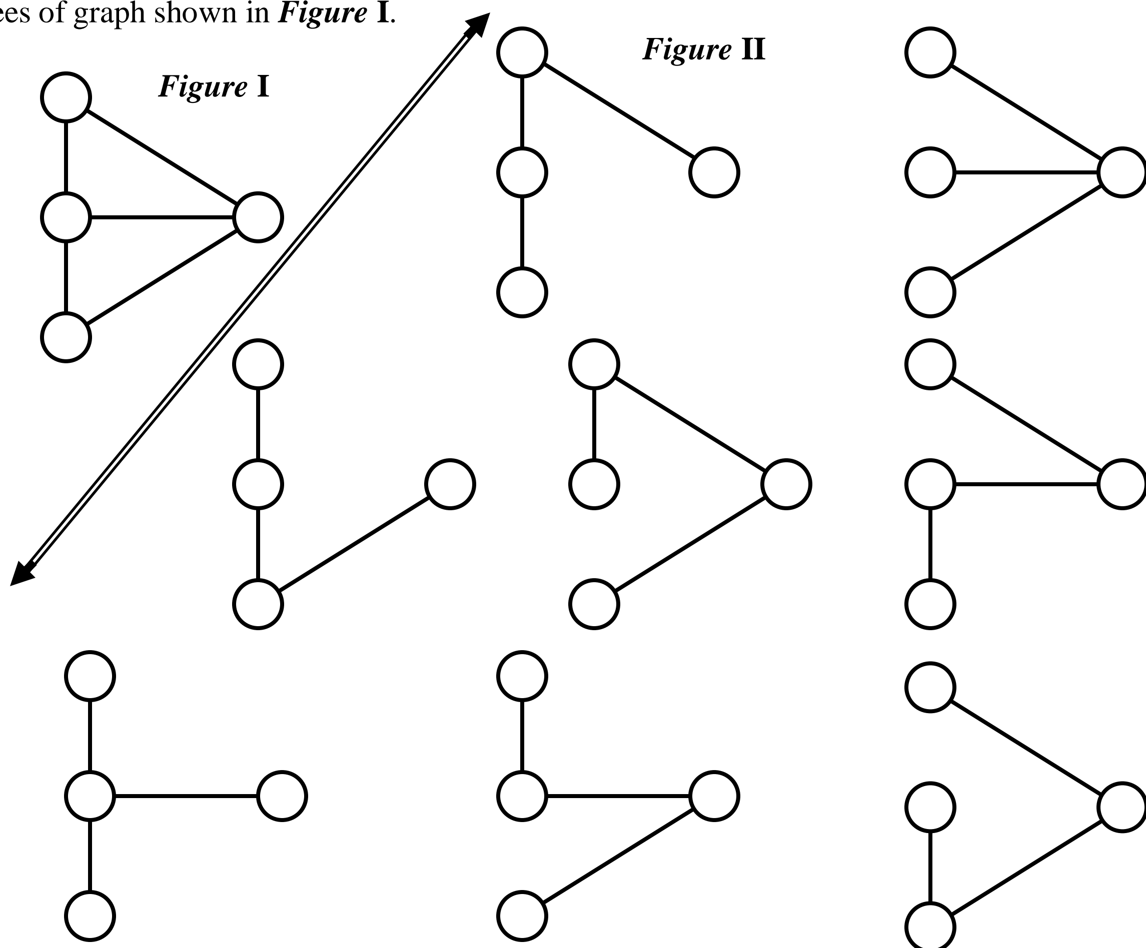
A ***spanning tree*** for an undirected graph G is a graph T consisting of the nodes of G together with enough from G such that:

1. There is a path between each pair of nodes in T.
2. There are no simple cycles in T.

It should be clear that if G is connected if and only if there is spanning tree for G. Thus, there is a spanning tree for every connected component of a graph.

If a graph G = (V, E) contain N nodes, then the spanning tree for that graph contains N-1 edges. The edges of spanning tree are subset of E.

In general, it is possible to construct different spanning trees for a graph, G. For example, the given bellow ***Figure* I** is a tree and ***Figure* II** shows the possible spanning trees of graph shown in ***Figure* I**.



For any spanning tree we could pick any node as the root and that would provide a parent-child relationship for the nodes connected by each edge.

A **minimal spanning tree** for a weighted graph G is a spanning tree such that the sum of its weights is less than or equal to the sum of the weights of every other spanning tree for G. That is in a minimal spanning tree the sum of weights of the edges is as small as possible.