UNIT- V

PART-1

Trees and Graphs

Topics:-

Trees

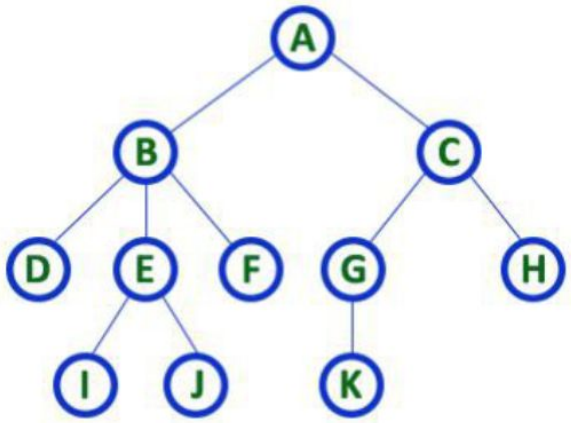
Tree terminology
Representation
Binary trees
Representation
binary tree traversals
binary tree operations

Graphs

Graph terminology
Graph representation
Elementary graph operations
Breadth First Search (BFS)
Depth First Search (DFS)
Connected components
Spanning trees

TOPIC 1: TREE INTRODUCTION AND TERMINOLOGY

A tree data structure is a non-linear data structure because it does not store elements in a sequential manner. In tree elements are arranged in multiple levels.



TREE with 11 nodes and 10 edges

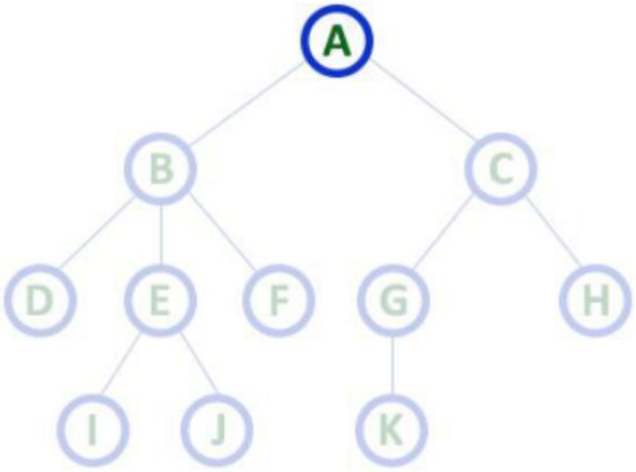
- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

Terminology

In a tree data structure, we use the following terminology...

1. Root

In a tree data structure, the first node is called as **Root Node**. Every tree must have a root node. We can say that the root node is the origin of the tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.

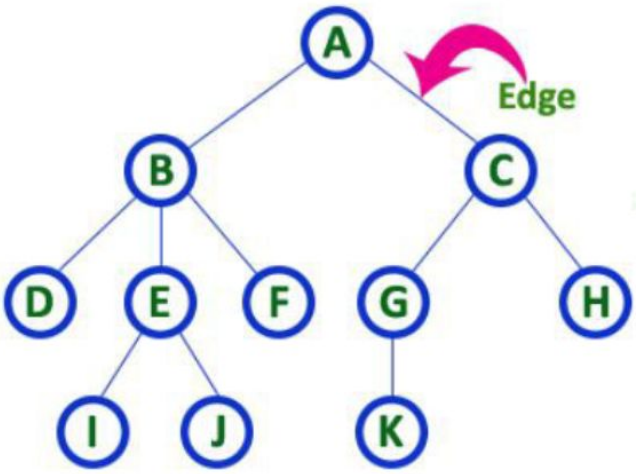


Here 'A' is the 'root' node

- In any tree the first node is called as ROOT node

2. Edge

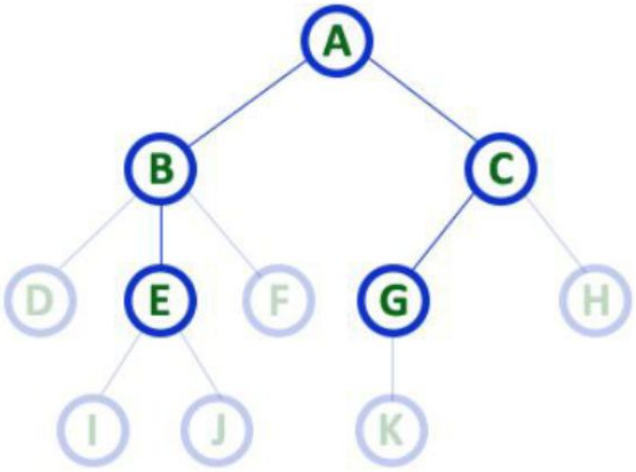
In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.



- In any tree, 'Edge' is a connecting link between two nodes.

3. Parent

In a tree data structure, the node which is a predecessor of any node is called as **PARENT NODE**. In simple words, the node which has a branch from it to any other node is called a parent node. Parent node can also be defined as "**The node which has child / children**".

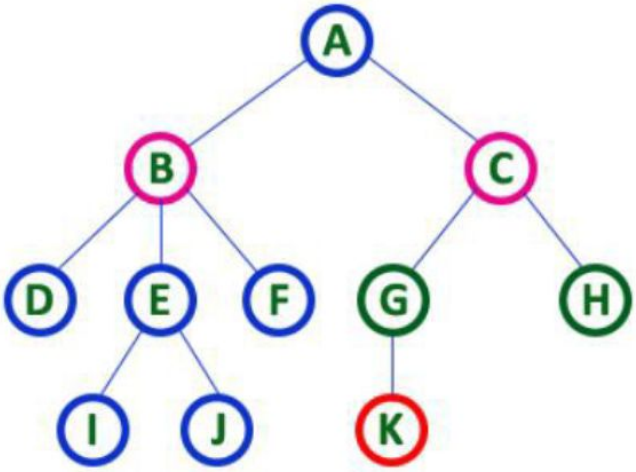


Here A, B, C, E & G are Parent nodes

- **In any tree the node which has child / children is called 'Parent'**
- **A node which is predecessor of any other node is called 'Parent'**

4. Child

In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.



Here B & C are Children of A

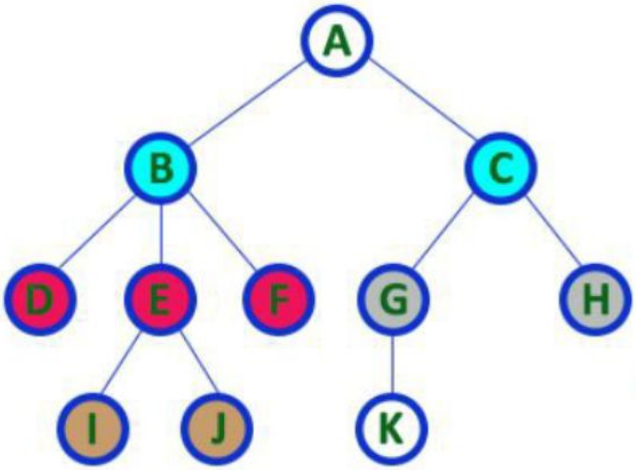
Here G & H are Children of C

Here K is Child of G

- **descendant of any node is called as CHILD Node**

5. Siblings

In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with the same parent are called Sibling nodes.



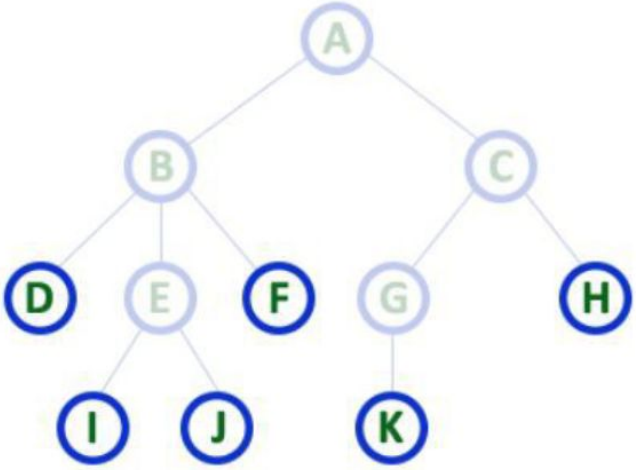
Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

- **In any tree the nodes which has same Parent are called 'Siblings'**
- **The children of a Parent are called 'Siblings'**

6. Leaf

In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child.

In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as '**Terminal**' node.



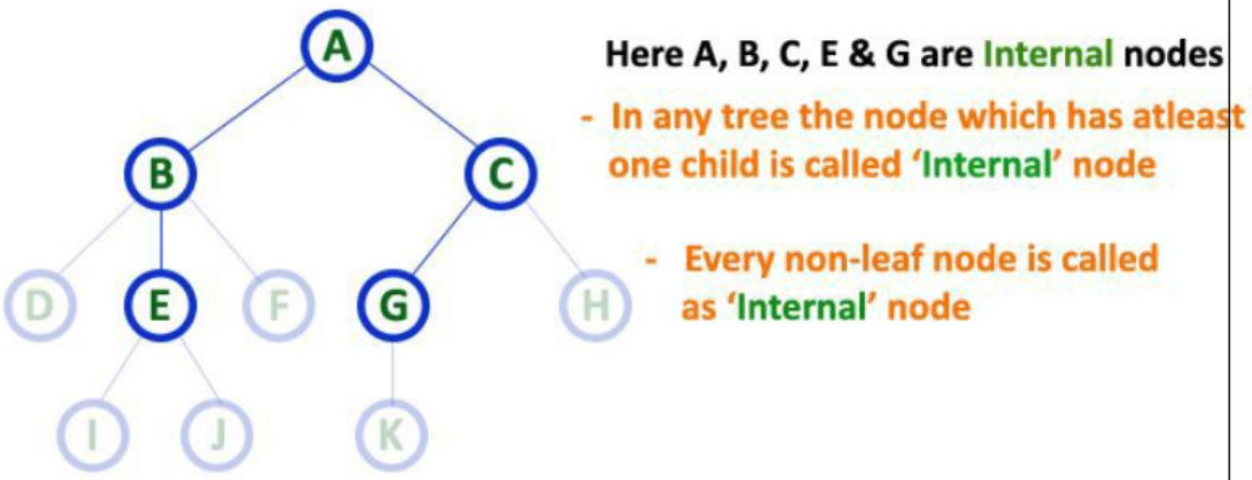
Here D, I, J, F, K & H are Leaf nodes

- **In any tree the node which does not have children is called 'Leaf'**
- **A node without successors is called a 'leaf' node**

7. Internal Nodes

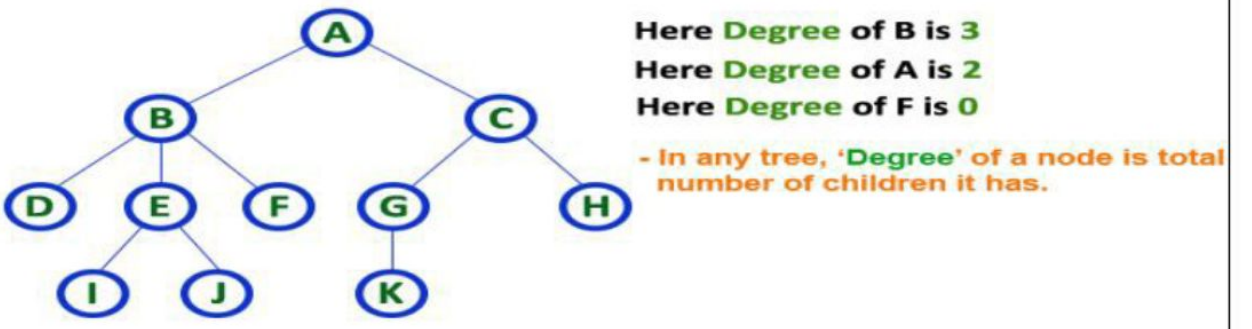
In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.

In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**. **The root node is also said to be Internal Node** if the tree has more than one node. Internal nodes are also called as '**Non-Terminal**' nodes.



8. Degree

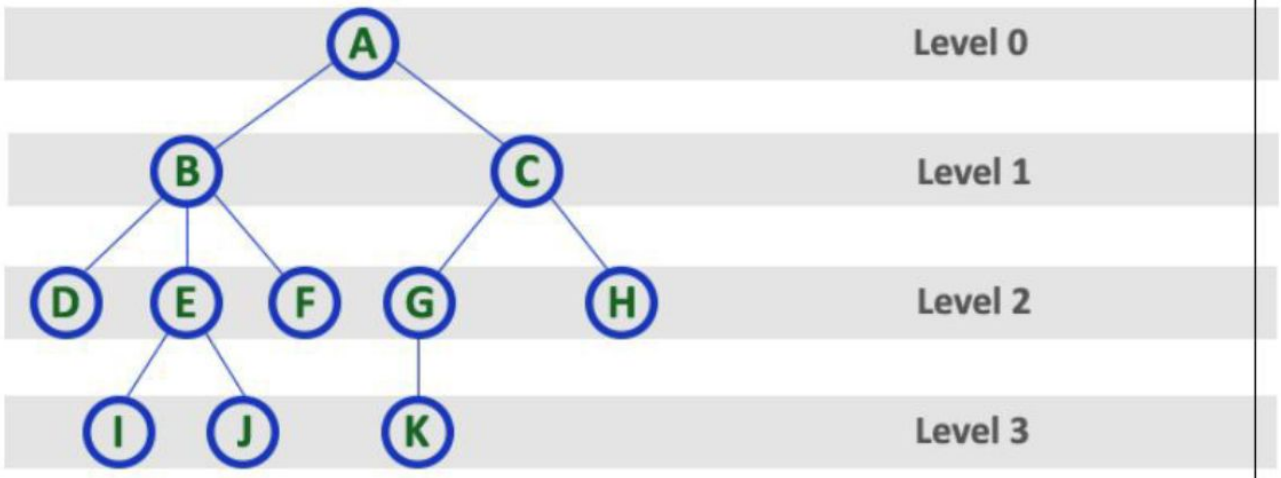
In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'



9. Level

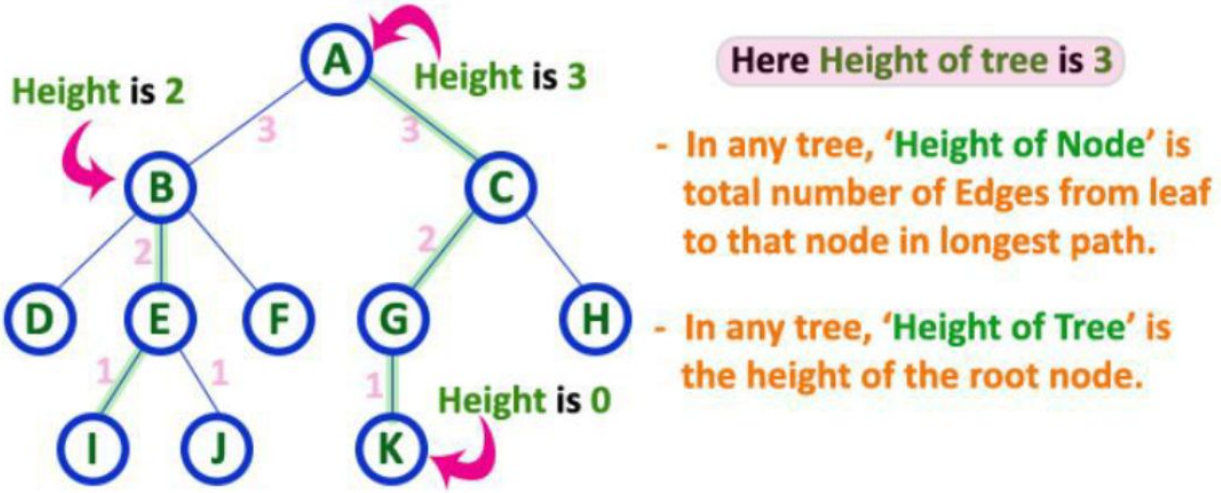
In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on...

In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



10. Height

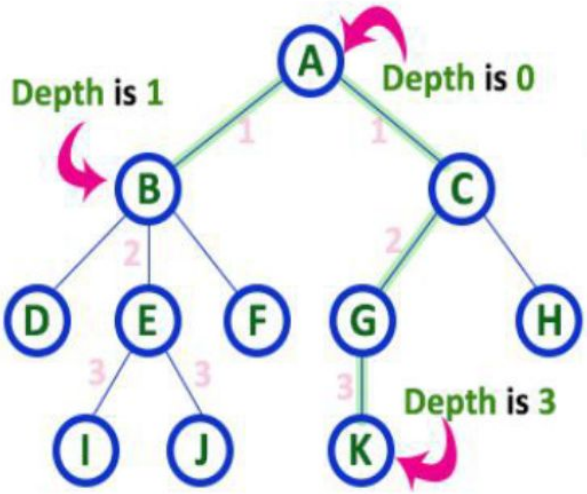
In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. In a tree, height of the root node is said to be **height of the tree**. In a tree, **height of all leaf nodes is '0'.**



11. Depth

In a tree data structure, the total number of egdes from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf

node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'.**

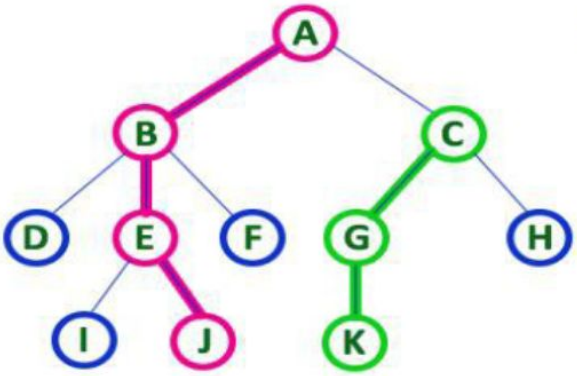


Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - J has length 4**.



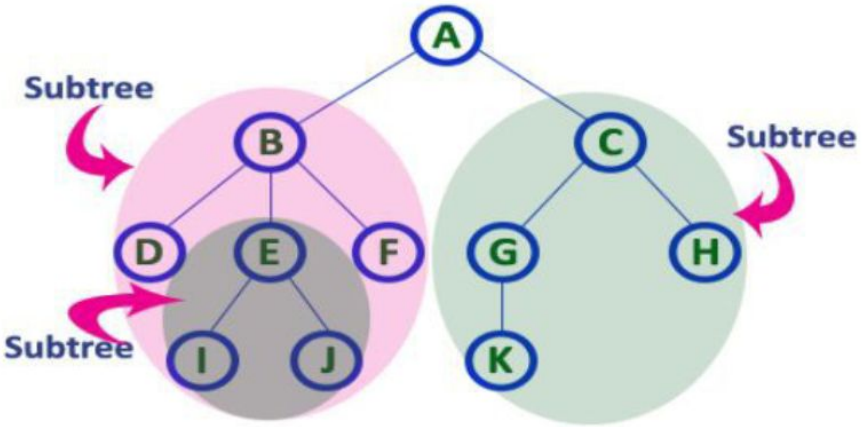
- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is
A - B - E - J

Here, 'Path' between C & K is
C - G - K

13. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



TOPIC 2: TREE REPRESENTATIONS

A tree data structure can be represented in three methods. Those methods are as follows...

1. **List Representation**
2. **Left Child - Right Sibling Representation**
3. **Degree two / Left child – Right child Representation**

1. List Representation

In this representation, we use two types of nodes one for representing the node with data called 'data node' and another for representing only references called 'reference node'. We start with a 'data node' from the root node in the tree. Then it is linked to an internal node through a 'reference node' which is further linked to any other node directly. This process repeats for all the nodes in the tree.

2. Left Child - Right Sibling Representation

In this representation, we use a list with one type of node which consists of three fields namely Data field, Left child reference field and Right sibling reference field. Data field stores the actual value of a node, left reference field stores the address of the left child and right

reference field stores the address of the right sibling node. Graphical representation of that node is as follows...

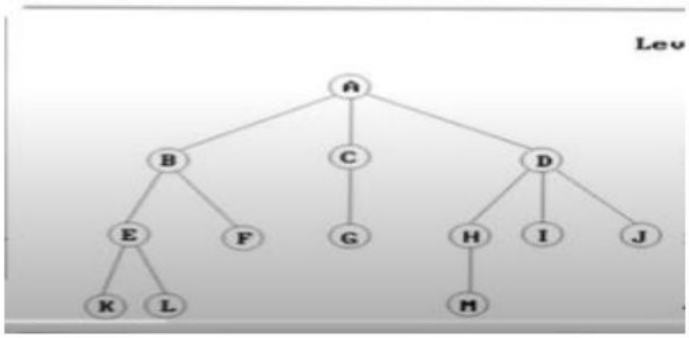
In this representation, every node's data field stores the actual value of that node. If that node has left a child, then left reference field stores the address of that left child node otherwise stores NULL. If that node has the right sibling, then right reference field stores the address of right sibling node otherwise stores NULL.

3. Degree two / Left child – Right child Representation

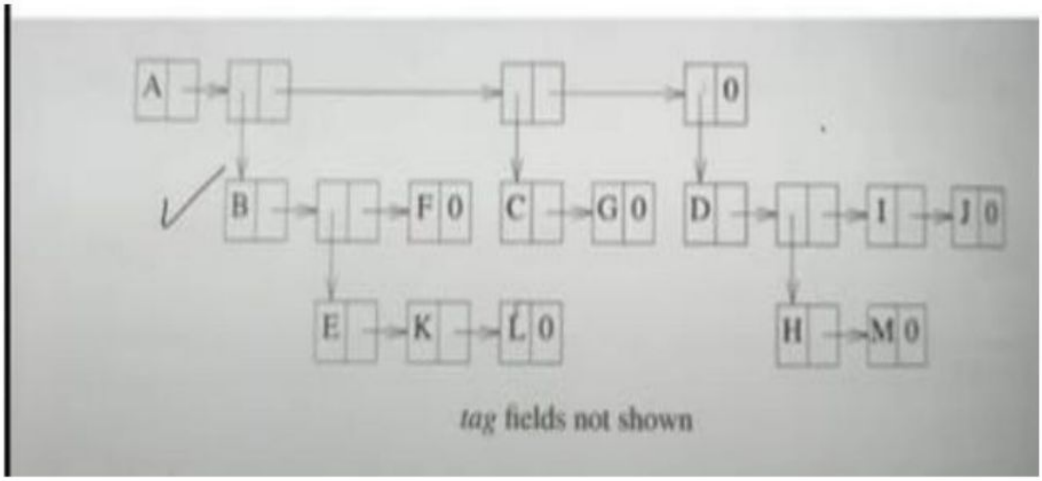
In a **degree-two representation**, the **two** children of anode are referred as left and right children. In a normal **tree**, every node can have any number of children. In this representation, rotate right sibling pointer in left child – right sibling tree clockwise by 45 degrees.

Note:- **Binary tree** is a special type of **tree data structure** in which every node can have a maximum **of 2** children.

Example:- Consider below tree:



1) List Representation:-



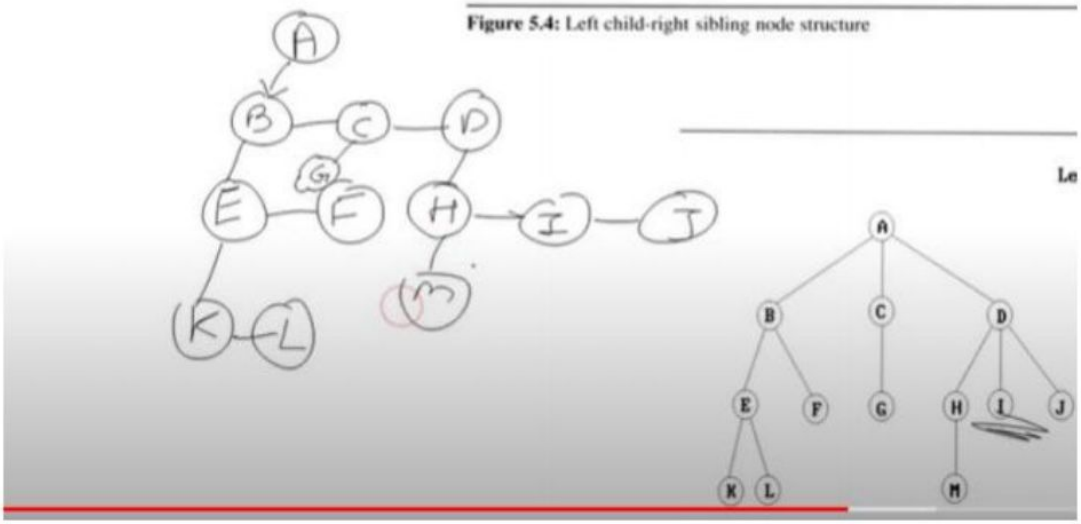
2) Left Child – Right Sibling Representation:-

Representation Of Trees (cont'd)

– Left Child- Right Sibling Representation

data	
left child	right sibling

Figure 5.4: Left child-right sibling node structure



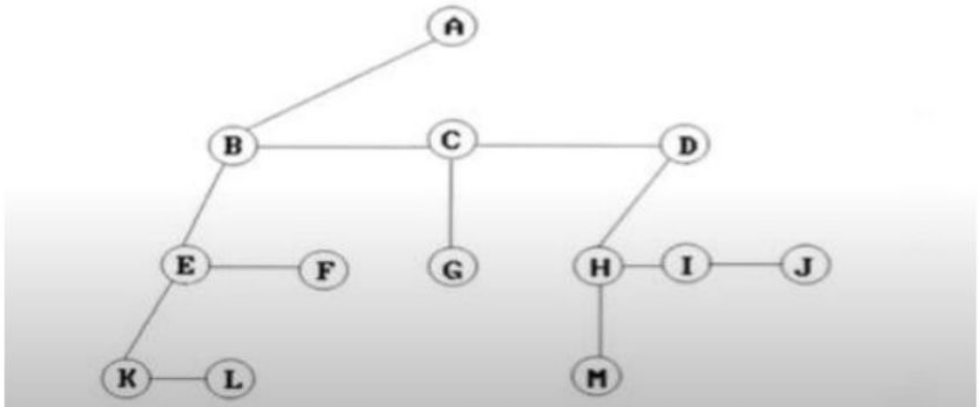


Figure 5.5: Left child-right sibling representation of a tree

3) Degree two / Left Child – Right Child Representation:-

Representation Of Trees (cont'd)

- Representation As A Degree Two Tree
- Rotate the right sibling pointers in left child right sibling tree clockwise by 45 degrees

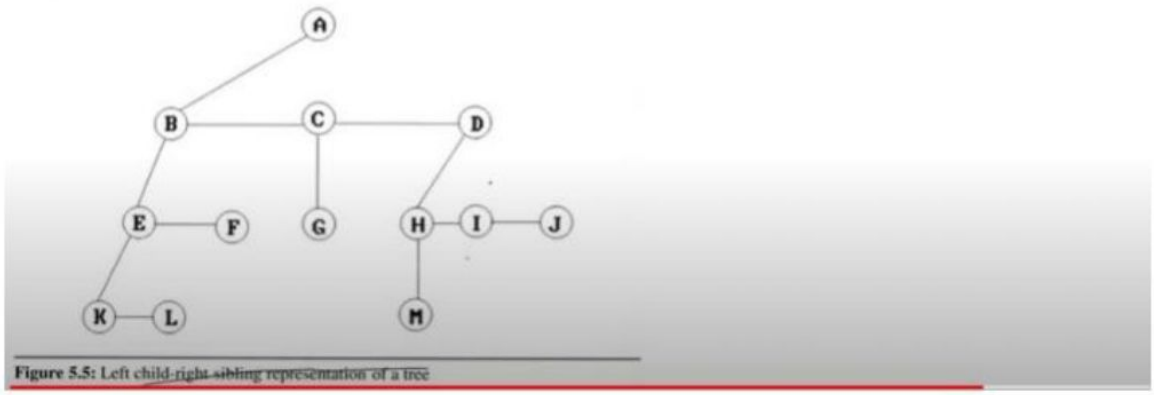


Figure 5.5: Left child-right sibling representation of a tree

Representation Of Trees (cont'd)

- Representation As A Degree Two Tree
- Rotate the right sibling pointers in left child right sibling tree clockwise by 45 degrees

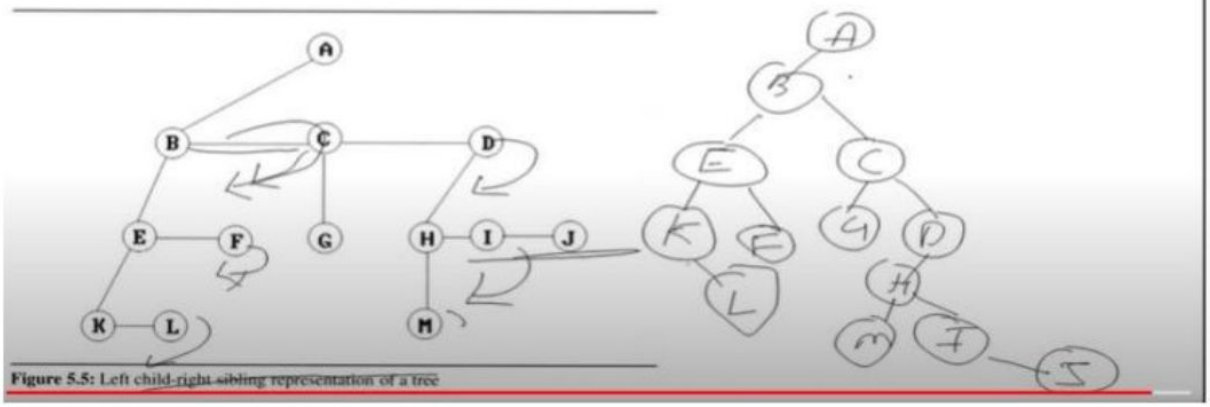
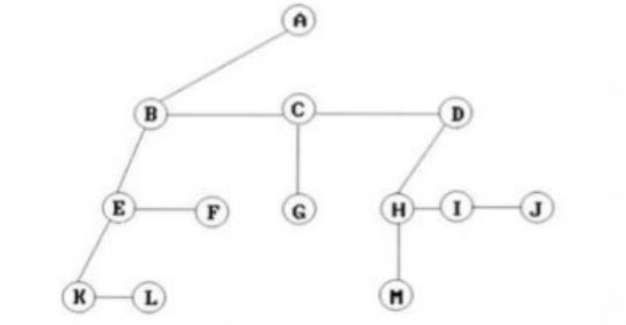


Figure 5.5: Left child-right sibling representation of a tree

Representation Of Trees (cont'd)

- Representation As A Degree Two Tree



igure 5.5: Left child-right sibling representation of a tree

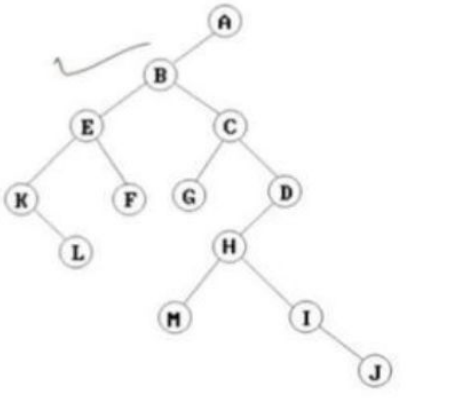
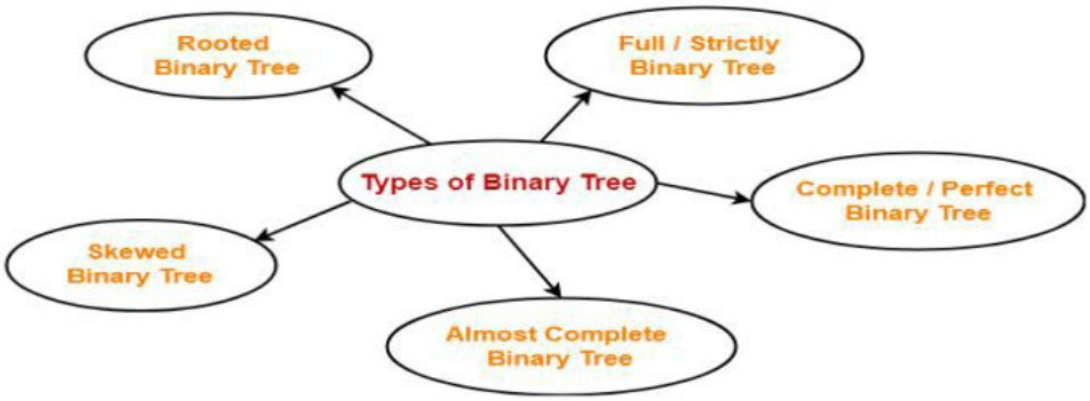


Figure 5.6: Left child-right child tree representation of a tree

TOPIC 3: BINARY TREES

A binary tree is a hierarchical data structure in which each node has at most two children generally referred as left child and right child.

Types of binary trees:-

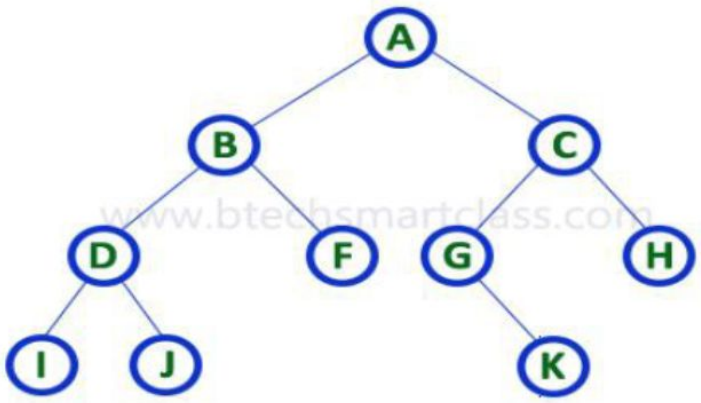


TOPIC 4: BINARY TREE REPRESENTATIONS

A binary tree data structure is represented using two methods. Those methods are as follows...

1. **Array Representation**
2. **Linked List Representation**

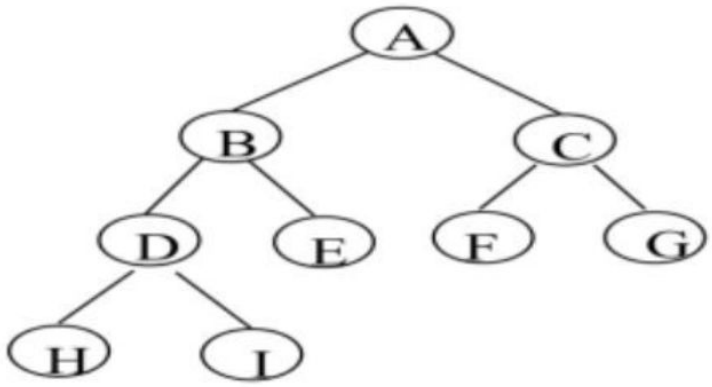
Consider the following binary tree...



1. Array Representation of Binary Tree

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.
To represent a binary tree of depth **'n'** using array representation, we need one dimensional array with a maximum size of **2n + 1**.
Case 1:-
If node is at i^{th} index , left child would be at [(2*i)+1] , right child would be at [(2*i)+2] and parent would be at [(i-1)/2].
(or)
Case 2:-
If node is at i^{th} index , left child would be at [(2*i)] , right child would be at [(2*i)+1] and parent would be at [(i/2)].

Example 1:-



Case 1:-

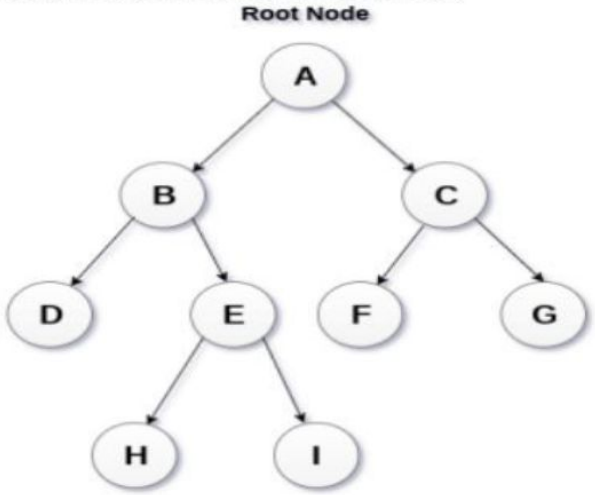
0	1	2	3	4	5	6	7	8
A	B	C	D	E	F	G	H	I

Case 2:-

1	2	3	4	5	6	7	8	9
A	B	C	D	E	F	G	H	I

Example 2:-
Note:- Before going to represent binary tree , first convert it to complete binary tree.

A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.

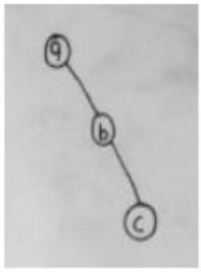


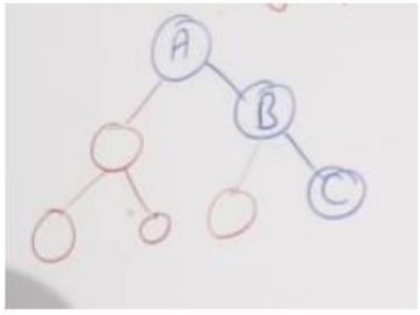
A	B	C	D	E	F	G			H	I
1	2	3	4	5	6	7	8	9	10	11

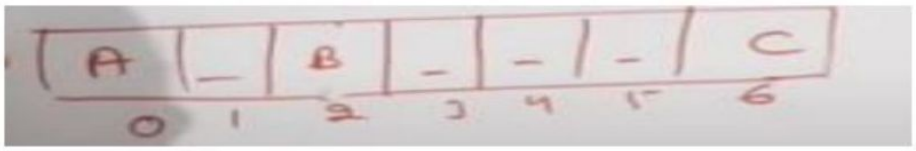
Sequential Representation of Binary Tree

0	1	2	3	4	5	6	7	8	9	10
A	B	C	D	E	F	G	-	-	H	I

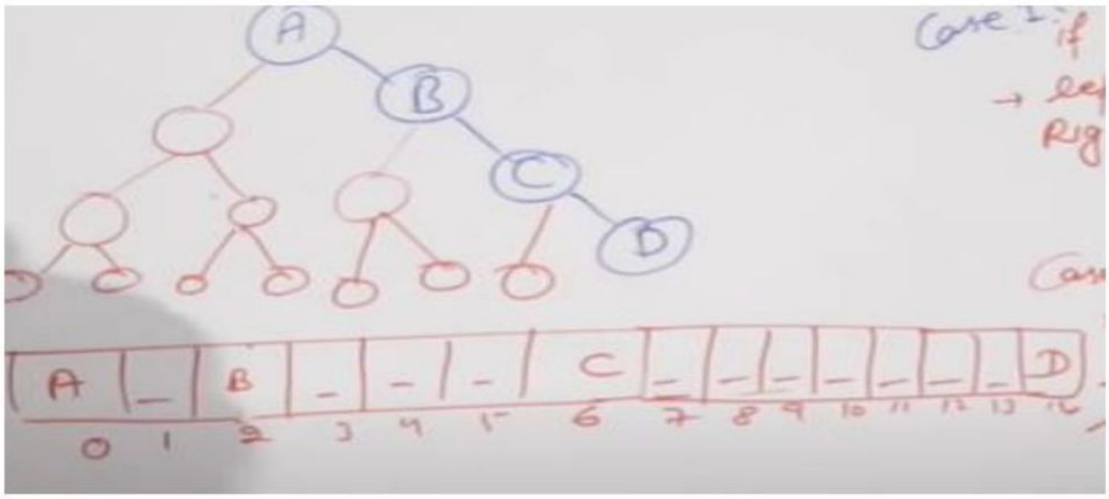
Example 3:-







Example 4:-



2. Linked List Representation of Binary Tree

We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.
In this linked list representation, a node has the following structure...

Example 1 :-

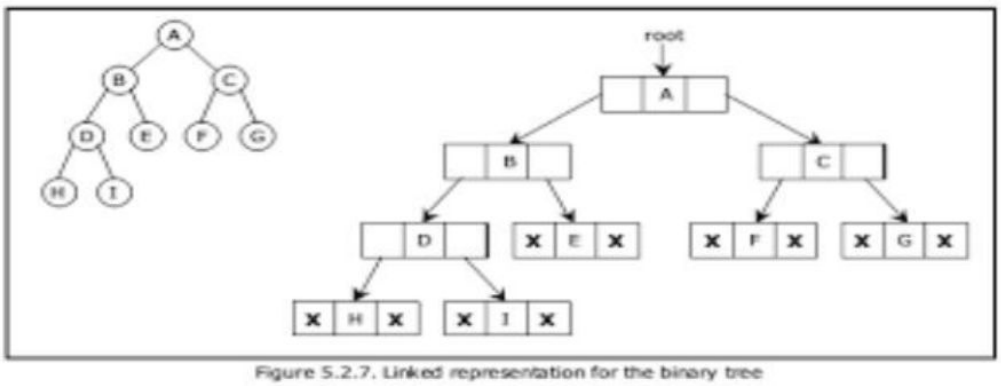
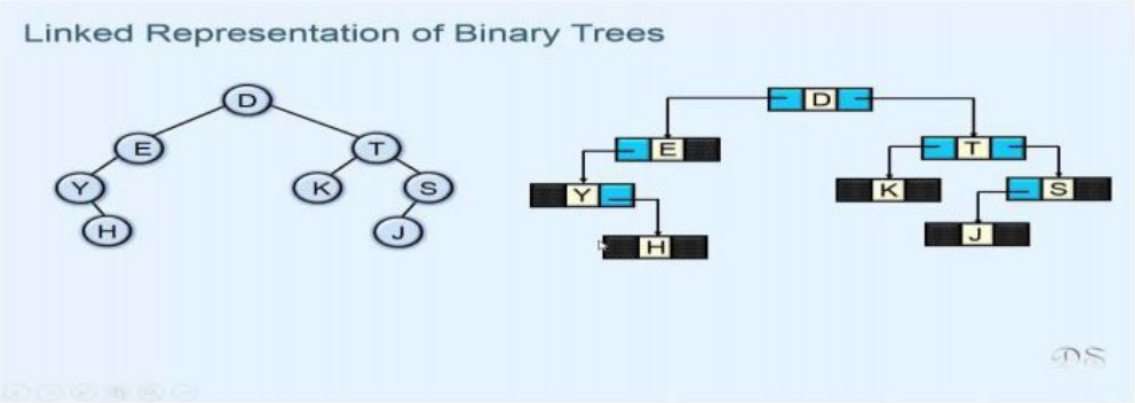
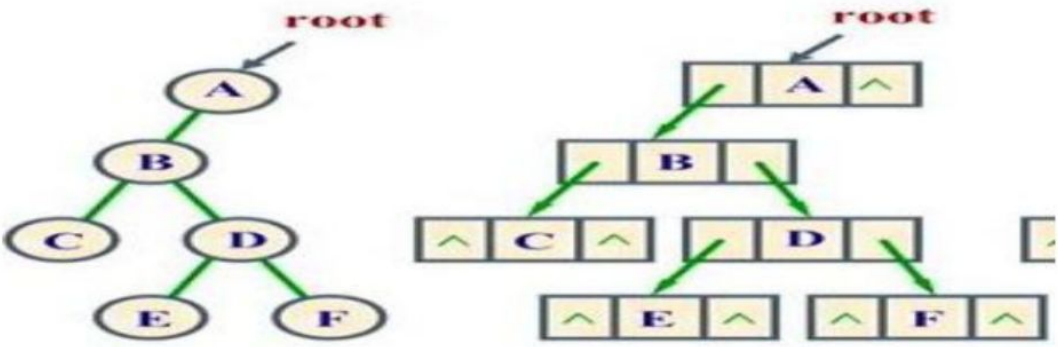


Figure 5.2.7. Linked representation for the binary tree

Example 2:-



Example 3:-



TOPIC 5: BINARY TREE TRAVERSALS

Binary Tree Traversal
Binary tree traversing is a process of accessing every node of the tree and exactly once. A tree is defined in a recursive manner. Binary tree traversal also defined recursively.

There are three techniques of traversal:

1. Preorder Traversal :- In Pre-Order traversal, the root node is visited before the left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.
2. Postorder Traversal:- In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.
3. Inorder Traversal:- In In-Order traversal, the root node is visited between the left child

and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting the right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

1. Preorder Traversal

Algorithm for Preorder traversal

Step 1 : Start from the Root.

Step 2 : Then, go to the Left Subtree.

Step 3 : Then, go to the Right Subtree.

2. Postorder Traversal

Algorithm for Postorder traversal

Step 1 : Start from the Left Subtree (Last Leaf).

Step 2 : Then, go to the Right Subtree.

Step 3 : Then, go to the Root.

3. Inorder Traversal

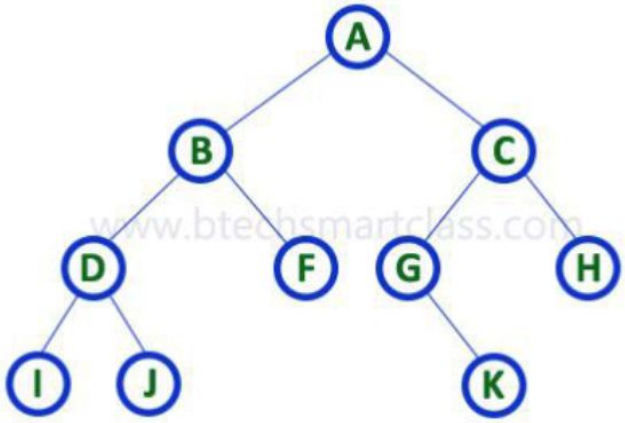
Algorithm for Inorder traversal

Step 1 : Start from the Left Subtree.

Step 2 : Then, visit the Root.

Step 3 : Then, go to the Right Subtree.

Example 1:-



In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C - H

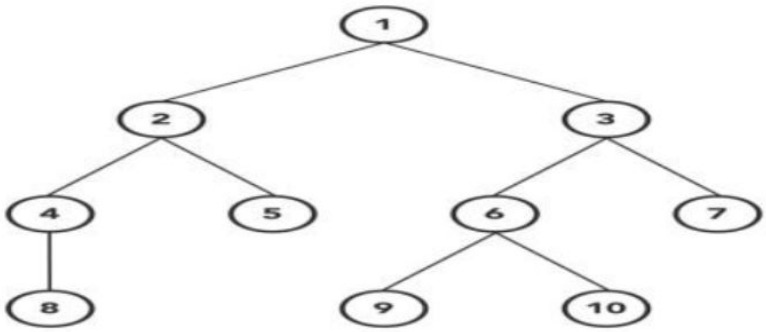
Pre-Order Traversal for above example binary tree is

A - B - D - I - J - F - C - G - K - H

Post-Order Traversal for above example binary tree is

I - J - D - F - B - K - G - H - C - A

Example 2:-



Inorder Traversal

8 4 2 5 1 9 6 10 3 7

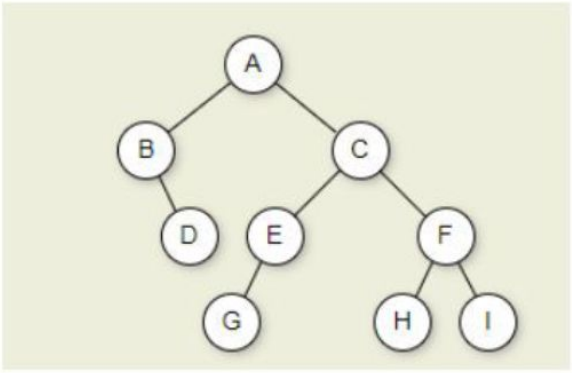
Preorder Traversal

1 2 4 8 5 3 6 9 10 7

Postorder Traversal

8 4 5 2 9 10 6 7 3 1

Example 3:-

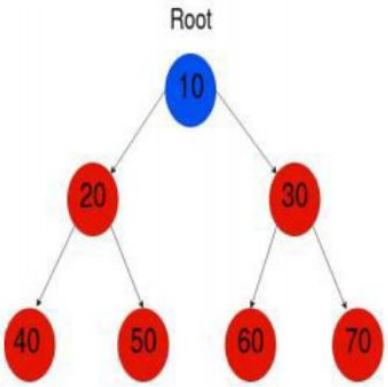


Inorder :- **B D A G E C H F I.**

Preorder:- **A B D C E G F H I.**

Postorder:- **D B G E H I F C A.**

Example 4:-



PreOrder Traversal :
10 -> 20 -> 40 -> 50 -> 30 -> 60 -> 70

InOrder Traversal :
40 -> 20 -> 50 -> 10 -> 60 -> 30 -> 70

PostOrder Traversal :
40 -> 50 -> 20 -> 60 -> 70 -> 30 -> 10

TOPIC 6: BINARY TREE OPERATIONS

There are many common operations performed on a binary tree –

Insertion:-Inserting an element into tree

Deletion:-deleting an element from tree

Search:- Searching whether given element is present in tree or not

Traversal:- Visiting all the nodes exactly once in tree in systematic way.

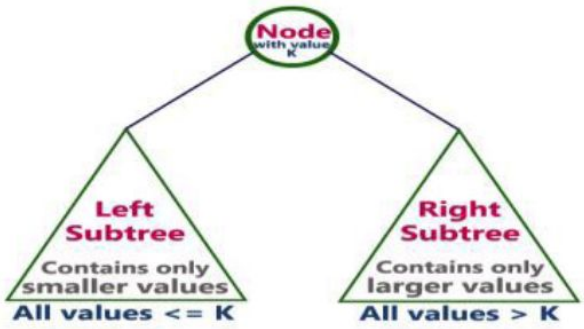
Binary Search Tree:-

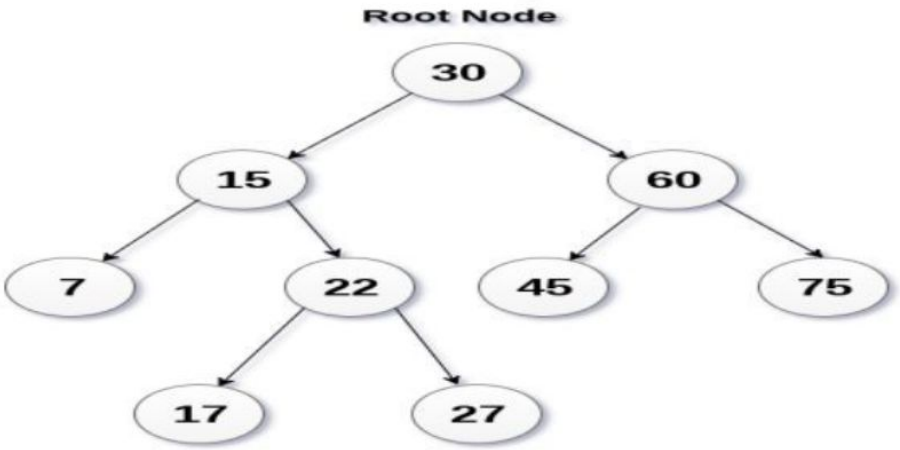
Binary Search Tree(BST): In computer science, a binary search tree (BST), also called an ordered or sorted binary tree. BST is binary tree that must satisfy the following properties

All elements of left sub tree should be less than root node

All elements to the right sub tree should be greater than root node

LST and RST itself binary search trees.





Binary Search Tree 0

Note:- Every binary search tree is a binary tree but every binary tree need not to be binary search tree.

Operations on a Binary Search Tree

The following operations are performed on a binary search tree...

1. Search
2. Insertion
3. Deletion

Search Operation in BST

The search operation is performed as follows...

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the value of root node in the tree.

Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.

Step 5 - If search element is smaller, then continue the search process in left subtree.

Step 6- If search element is larger, then continue the search process in right subtree.

Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node

Step 8 - If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.

Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Insertion Operation in BST

In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1 - Create a newNode with given value and set its left and right to NULL.
Step 2 - Check whether tree is Empty.
Step 3 - If the tree is Empty, then set root to newNode.
Step 4 - If the tree is Not Empty, then check whether the value of newNode is smaller or larger than the node (here it is root node).
Step 5 - If newNode is smaller than or equal to the node then move to its left child. If newNode is larger than the node then move to its right child.
Step 6- Repeat the above steps until we reach to the leaf node (i.e., reaches to NULL).
Step 7 - After reaching the leaf node, insert the newNode as left child if the newNode is smaller or equal to that leaf node or else insert it as right child.

Deletion Operation in BST

Deleting a node from Binary search tree includes following three cases...
Case 1: Deleting a Leaf node (A node with no children)
Case 2: Deleting a node with one child
Case 3: Deleting a node with two children

Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...
Step 1 - Find the node to be deleted using search operation
Step 2 - Delete the node using free function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...
Step 1 - Find the node to be deleted using search operation
Step 2 - If it has only one child then create a link between its parent node and child node.
Step 3 - Delete the node using free function and terminate the function.

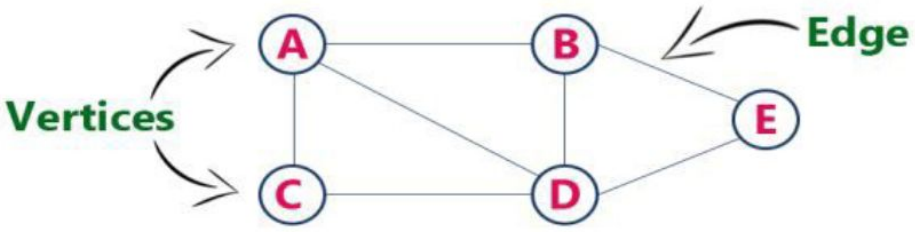
Case 3: Deleting a node with two children

We use the following steps to delete a node with two children from BST...
Step 1 - Find the node to be deleted using search operation
Step 2 - If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.
Step 3 - Swap both deleting node and node which is found in the above step.
Step 4 - Then check whether deleting node came to case 1 or case 2 or else goto step 2
Step 5 - If it comes to case 1, then delete using case 1 logic.
Step 6- If it comes to case 2, then delete using case 2 logic.
Step 7 - Repeat the same process until the node is deleted from the tree.

TOPIC 7: GRAPH TERMINOLOGY

Graph :- Graphs are non-linear data structures comprising a finite set of nodes and edges. The nodes are the elements and edges are ordered pairs of connections between the nodes. Generally, a graph is represented as a pair of sets (V, E). V is the set of vertices or nodes. E is the set of Edges.

Simple Definition of Graph:- Graph G can be defined as G = (V , E)
Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



Tree vs Graph

No.	Graph	Tree
1	Graph is a non-linear data structure.	Tree is a non-linear data structure.
2	It is a collection of vertices/nodes and edges.	It is a collection of nodes and edges.
3	Each node can have any number of edges.	General trees consist of the nodes having any number of child nodes. But in case of binary trees every node can have at the most two child nodes.
4	There is no unique node called root in graph.	There is a unique node called root in trees.
5	A cycle can be formed.	There will not be any cycle.
6	Applications: In Computer science graphs are used to represent Google maps Facebook Linkedin Twitter World Wide Web Operating System	Applications: searching Decision making machine learning game trees

Graph Terminology:-

1) Vertex

Individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, A, B, C, D & E are known as vertices.

2) Edge

An edge is a connecting link between two vertices.

Edges are three types.

1. **Undirected Edge -** An undirected egde is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).
2. **Directed Edge -** A directed egde is a unidirectional edge. If there is directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).
3. **Weighted Edge -** A weighted egde is a edge with value (cost) on it.

3) Undirected Graph

A graph with only undirected edges is said to be undirected graph.

4) Directed Graph

A graph with only directed edges is said to be directed graph.

5) Mixed Graph

A graph with both undirected and directed edges is said to be mixed graph.

6) End vertices or Endpoints

The two vertices joined by edge are called end vertices (or endpoints) of that edge.

7) Origin

If a edge is directed, its first endpoint is said to be the origin of it.

8) Destination

If a edge is directed, its first endpoint is said to be the origin of it and the other endpoint is said to be the destination of that edge.

9) Adjacent

If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, vertices A and B are said to be adjacent if there is an edge between them.

10)Incident

Edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

11)Outgoing Edge

A directed edge is said to be outgoing edge on its origin vertex.

12)Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

13)Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

14)Indegree

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

15)Outdegree

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

16)Parallel edges or Multiple edges

If there are two undirected edges with same end vertices and two directed edges with same origin and destination, such edges are called parallel edges or multiple edges.

17)Self-loop

Edge (undirected or directed) is a self-loop if its two endpoints coincide with each other.

18)Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

19)Path

A path is a sequence of alternate vertices and edges that starts at a vertex and ends at other vertex such that each edge is incident to its predecessor and successor vertex.

TOPIC 8: GRAPH REPRESENTATION

Graph data structure is represented using following representations...

1. **Adjacency Matrix**
2. **Incidence Matrix**

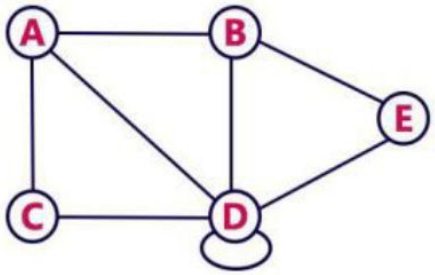
3. Adjacency List

Adjacency Matrix

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices. That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

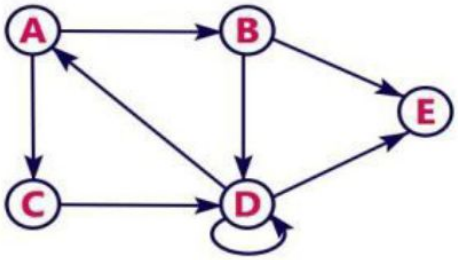
For example, consider the following

undirected graph representation...



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

Directed graph representation...



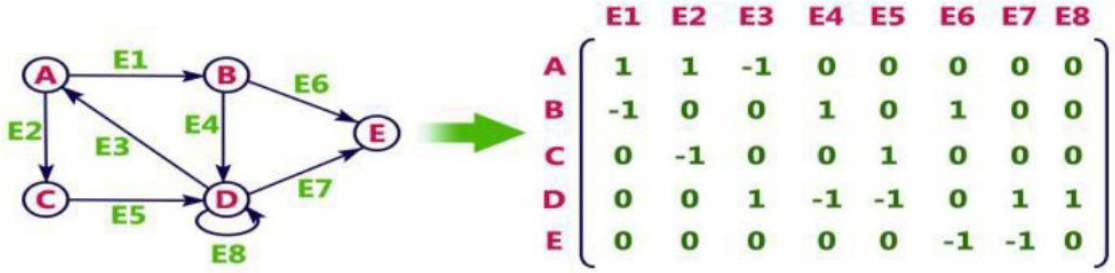
	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	1
C	0	0	0	1	0
D	1	0	0	1	1
E	0	0	0	0	0

Incidence Matrix

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of edges. That means graph with 4 vertices and 6 edges is represented using a matrix of size 4X6. In this matrix, rows represent vertices and columns

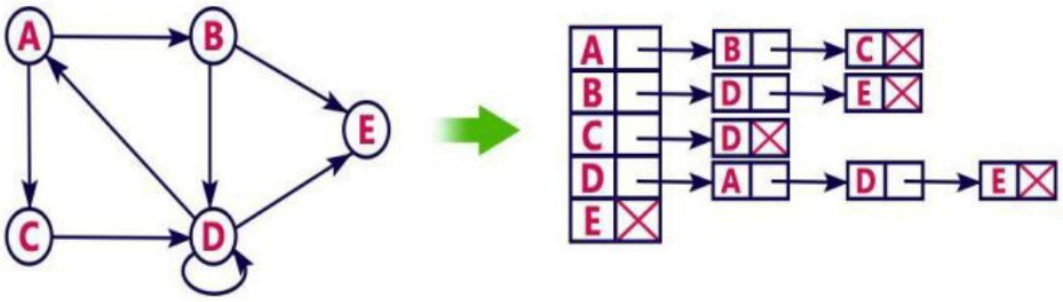
represents edges. This matrix is filled with 0 or 1 or -1. Here, 0 represents that the row edge is not connected to column vertex, 1 represents that the row edge is connected as the outgoing edge to column vertex and -1 represents that the row edge is connected as the incoming edge to column vertex.

For example, consider the following directed graph representation...

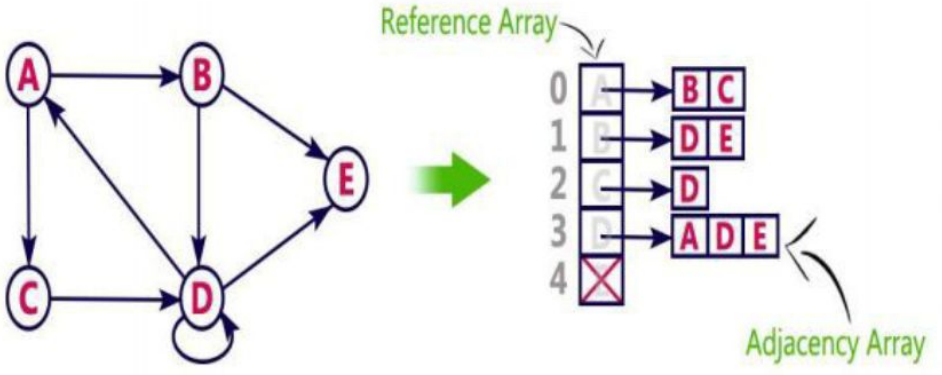


Adjacency List

In this representation, every vertex of a graph contains list of its adjacent vertices. For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using an array as follows..



TOPIC 9: ELEMENTARY GRAPH OPERATIONS

Given a graph G = (V E) and a vertex v in V(G)
Various graph operations are:-

1) Traversal - visiting all vertices in G exactly once.There are two graph traversal techniques.
   - Depth First Search (DFS)
   - Breadth First Search (BFS)
2) Connected components
3) Spanning tree

TOPIC 10: BREADTH FIRST SEARCH

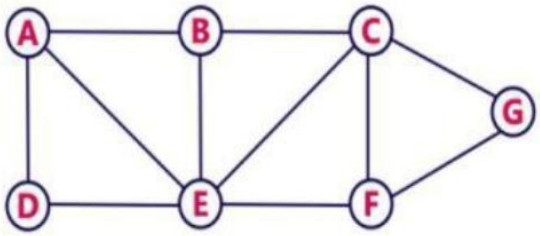
BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

We use the following steps to implement BFS traversal...

- Step 1 - Define a Queue of size total number of vertices in the graph.
- Step 2 - Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- Step 3 - Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.
- Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
- Step 5 - Repeat steps 3 and 4 until queue becomes empty.
- Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

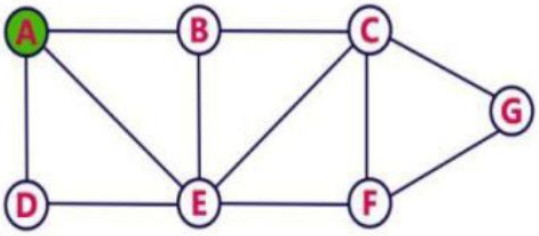
Example

Consider the following example graph to perform BFS traversal



Step 1:

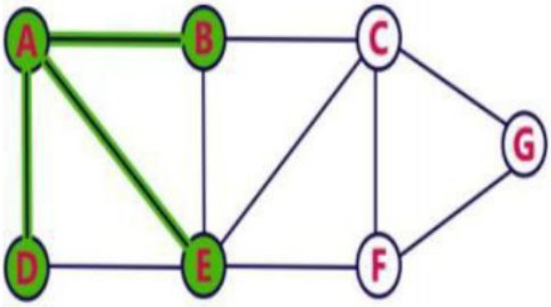
- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



Queue

Step 2:

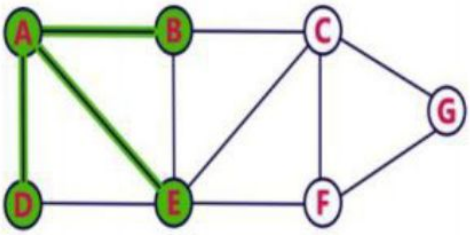
- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



Queue

Step 3:

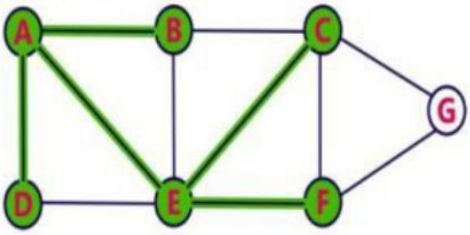
- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



Queue

Step 4:

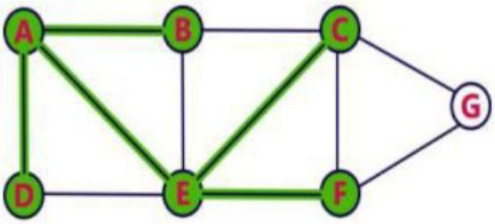
- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.



Queue

Step 5:

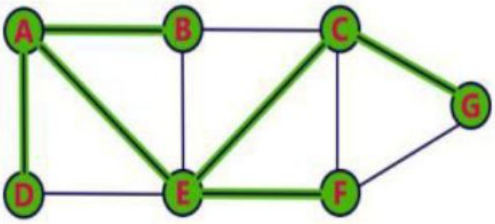
- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



Queue

Step 6:

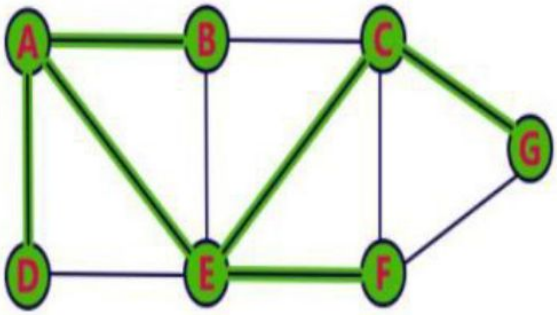
- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.



Queue

Step 7:

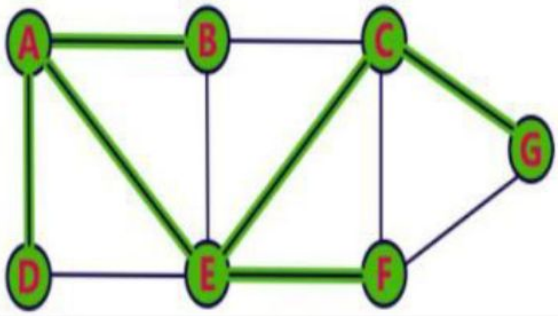
- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.



Queue

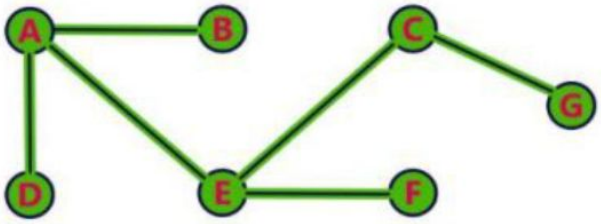
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



---****---

TOPIC 11: DEPTH FIRST SEARCH

DFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal.

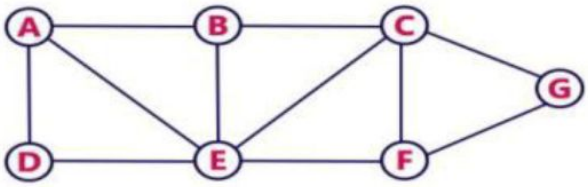
We use the following steps to implement DFS traversal...

- Step 1 - Define a Stack of size total number of vertices in the graph.
- Step 2 - Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.
- Step 3 - Visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it on to the stack.
- Step 4 - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
- Step 5 - When there is no new vertex to visit then use back tracking and pop one vertex from the stack.
- Step 6 - Repeat steps 3, 4 and 5 until stack becomes Empty.
- Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

Back tracking is coming back to the vertex from which we reached the current vertex.

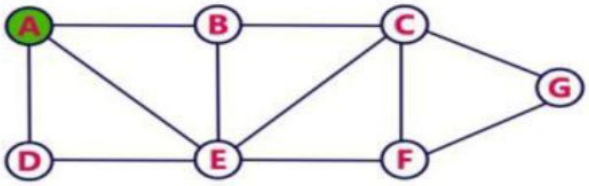
Example

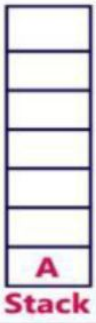
Consider the following example graph to perform DFS traversal



Step 1:

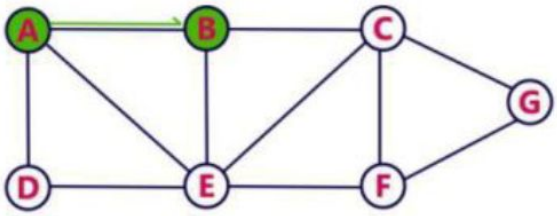
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.





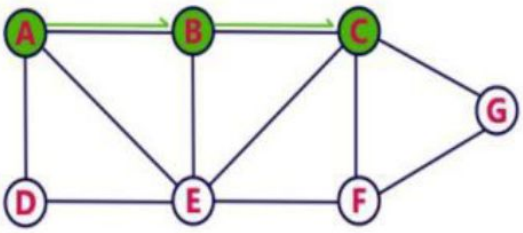
Step 2:

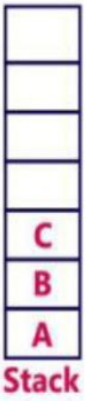
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



Step 3:

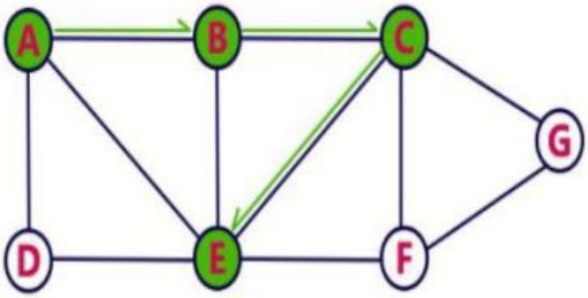
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.





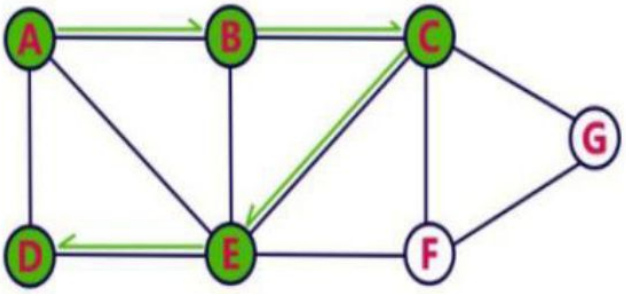
Step 4:

- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack



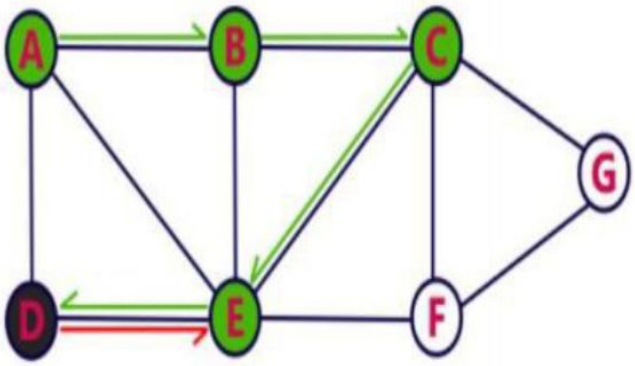
Step 5:

- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack



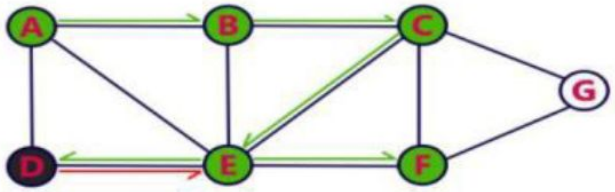
Step 6:

- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.



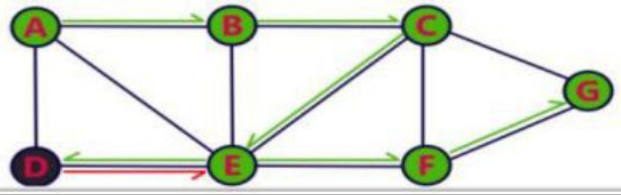
Step 7:

- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.



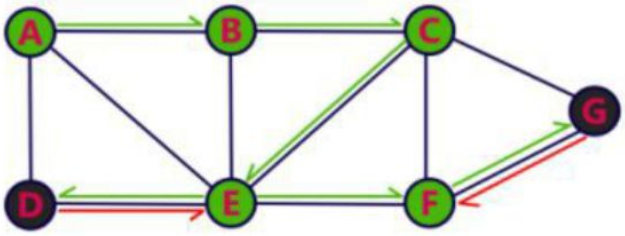
Step 8:

- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.



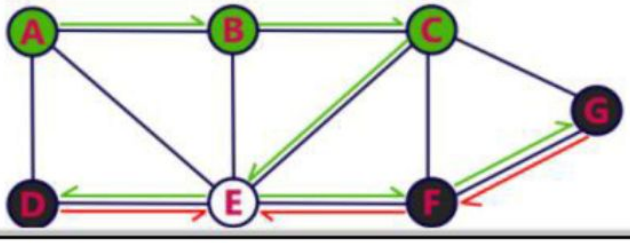
Step 9:

- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.



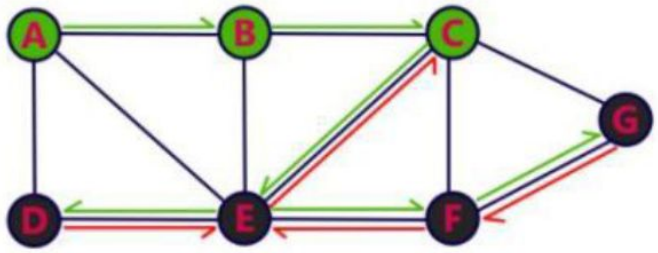
Step 10:

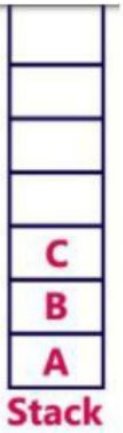
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.



Step 11:

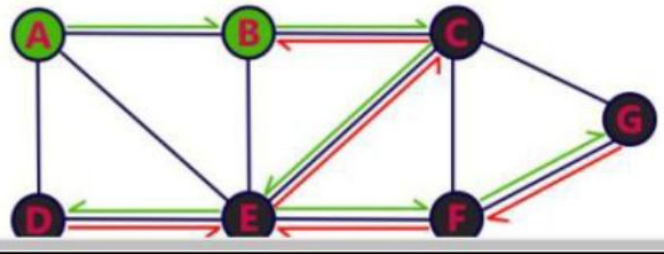
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.





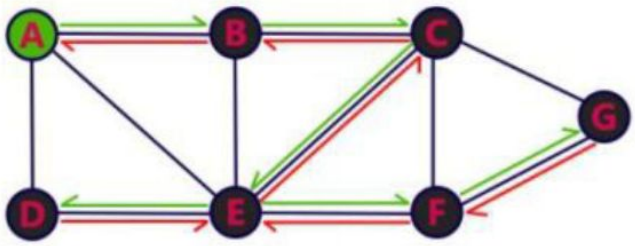
Step 12:

- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.



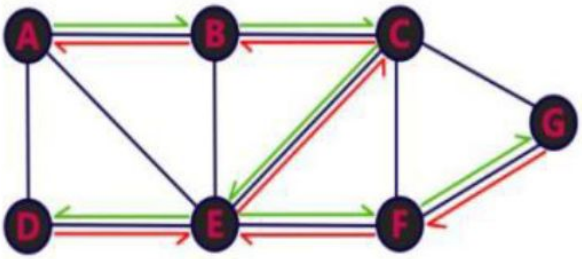
Step 13:

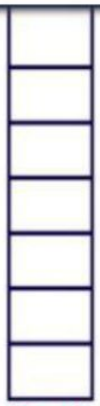
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.



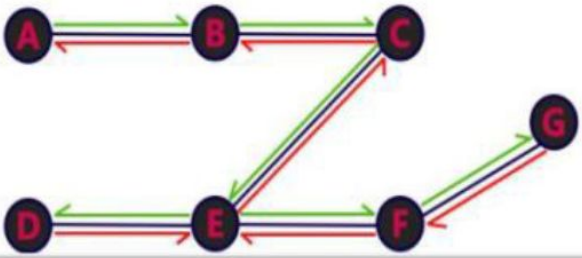
Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.





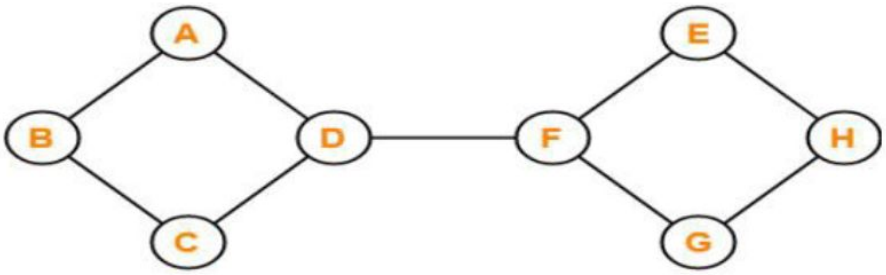
- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



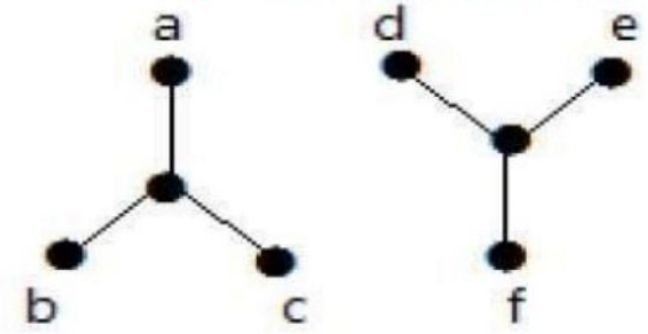
---****---

TOPIC 12 : CONNECTED COMPONENTS AND SPANNING TREES

1) Connected Graph:- A **graph** is said to be **connected** if every pair of vertices in the **graph** is **connected**. This means that there is a path between every pair of vertices.



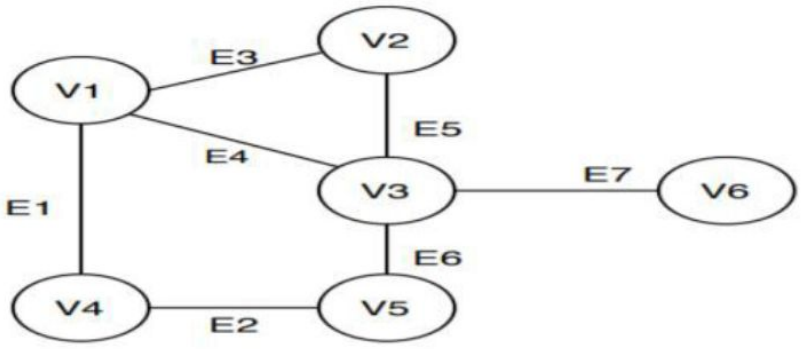
Example of Connected Graph

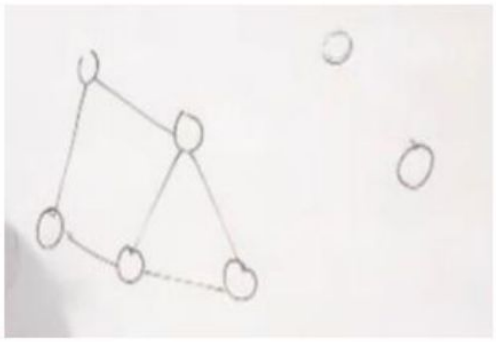


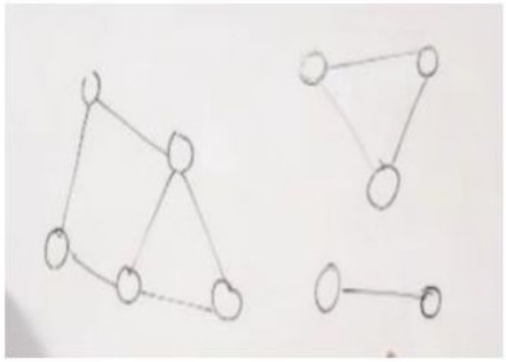
2) Connected Component is a sub graph in which each pair of nodes is connected with each other via a path.

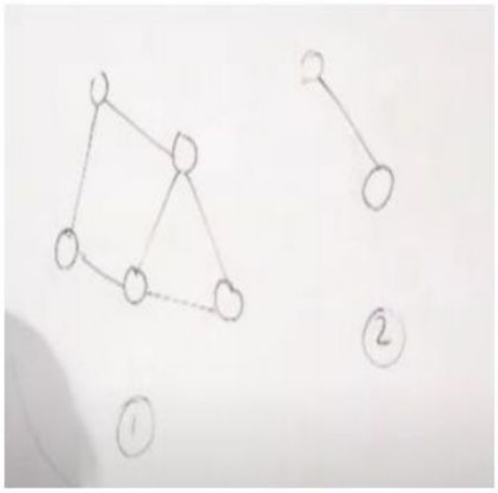
(or)

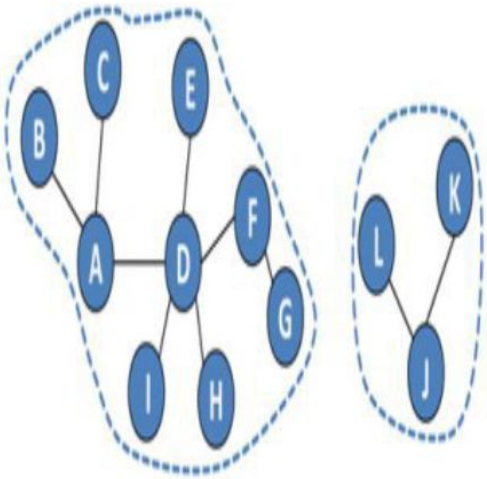
Connected Component is a maximal set of connected vertices.

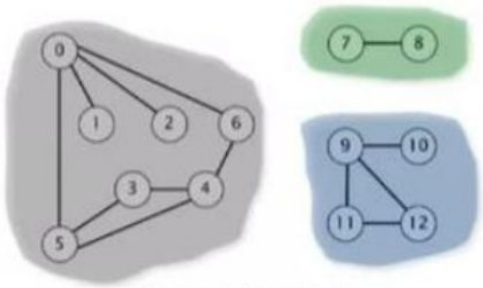










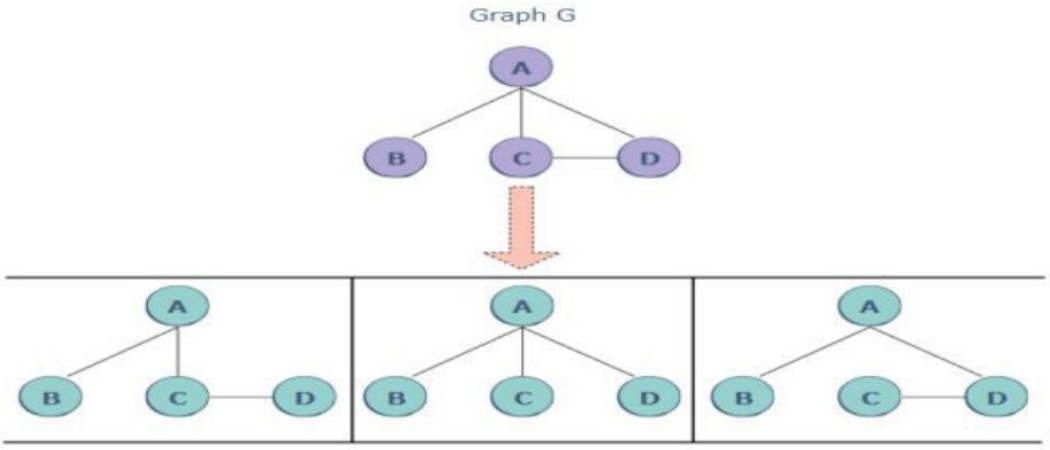


3 connected components

TOPIC 13 : SPANNING TREES

A spanning tree in G is a subgraph of G that includes all the vertices of G and is also a tree.

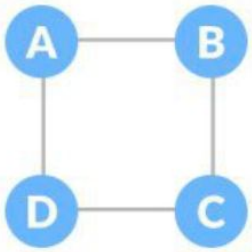
Example 1:-



Different possible spanning trees

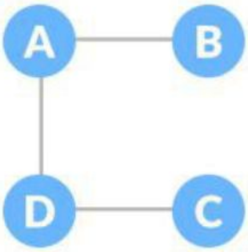
Example 2:-

Let the original graph be:

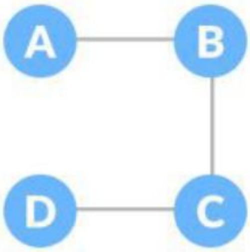


Normal graph

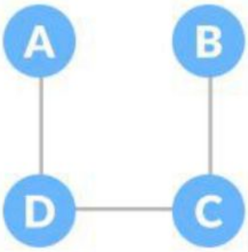
Some of the possible spanning trees that can be created from the above graph are:



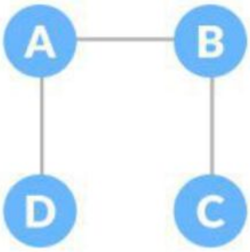
A spanning tree



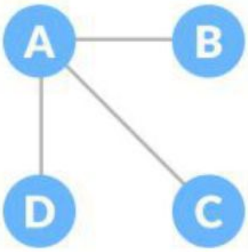
A spanning tree



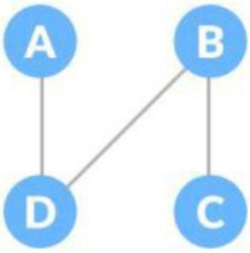
A spanning tree



A spanning tree



A spanning tree



A spanning tree

---*** THE END***---

UNIT-V

PART-2

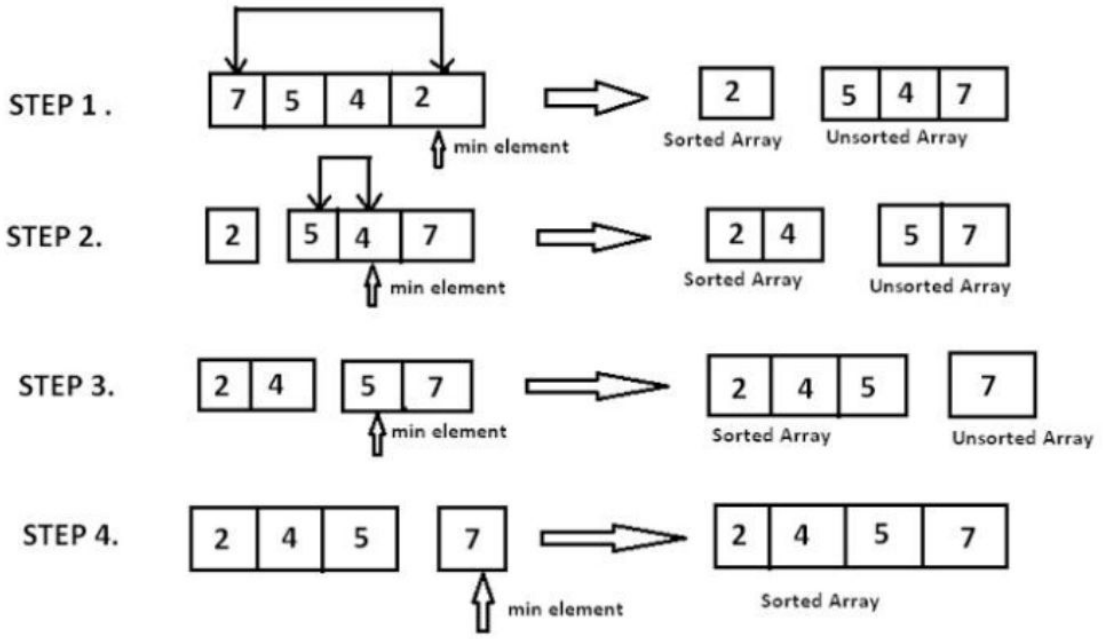
Sorting and Searching: Sorting by selection, sorting by exchange, sorting by insertion, sorting by partitioning, binary search.

TOPIC : SORTING BY SELECTION

Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

Procedure:-

1. Set the first element as minimum.
2. Compare minimum with the second element. If the second element is smaller than minimum, assign second element as minimum.
   Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.
3. After each iteration, minimum is placed in the front of the unsorted list.
4. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.



Program for selection sort

```
void main( )
{
   int a[50], n, i, j, temp, min;
   printf("\nEnter size of an array: ");
   scanf("%d", &n);
   printf("\nEnter elements of an array:\n");
   for(i=0; i<n; i++)
      scanf("%d", &a[i]);
         for (i=0; i<n; i++)
            {
               min = i;
               for (j=i+1; j<n; j++)
               {
                  if (a[j] < a[min])
                     min = j;
               }
               temp = a[i];
               a[i] = a[min];
               a[min] = temp;
            }
         printf("\nAfter sorting:\n");
            for(i=0; i<n; i++)
               printf("%d\t", a[i]);
}
```

Input and Output:-

```
Enter size of an array: 8
Enter elements of an array:
68 45 78 14 25 65 55 44
After sorting:
14 25 44 45 55 65 68 78
```

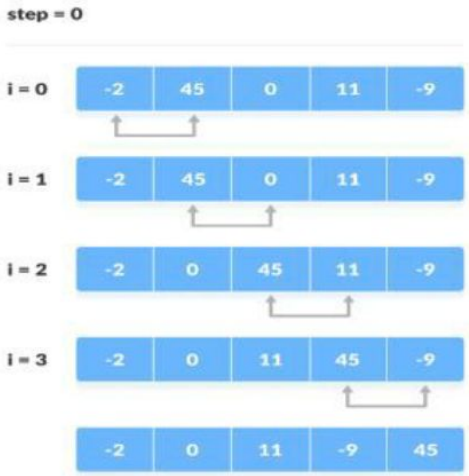
TOPIC : SORTING BY EXCHANGE

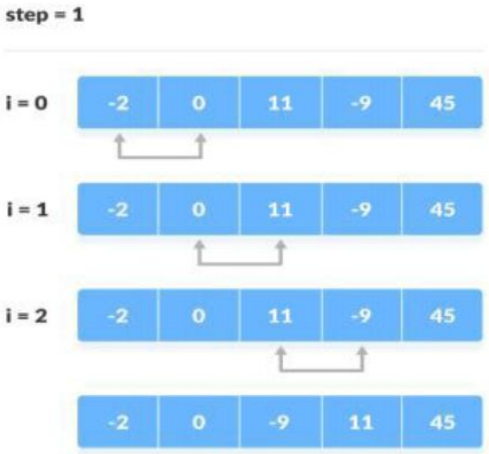
Bubble sort is an algorithm that compares the adjacent elements and swaps their positions if they are not in the intended order.

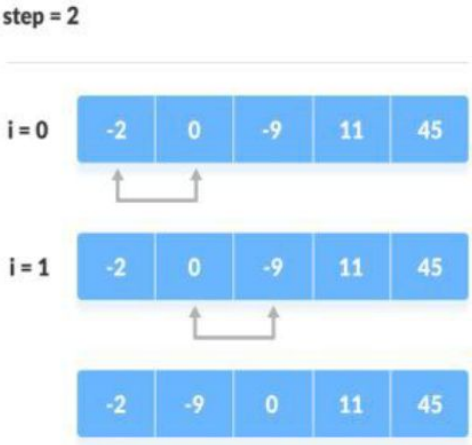
Procedure:-

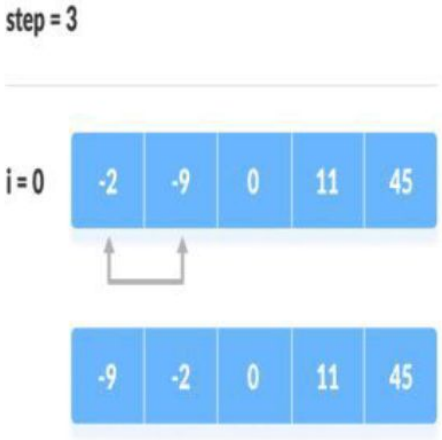
1. Starting from the first index, compare the first and the second elements. If the first element is greater than the second element, they are swapped.
   Now, compare the second and the third elements. Swap them if they are not in order.
   The above process goes on until the last element.
2. The same process goes on for the remaining iterations.
   After each iteration, the largest element among the unsorted elements is placed at the end. In each iteration, the comparison takes place up to the last unsorted element.
   The array is sorted when all the unsorted elements are placed at their correct positions.

Example:- Bubble sort









Program for bubble sort:-

```
void main( )
{
   int a[50], n, I ,j, temp, min;
   printf("\nEnter size of an array: ");
   scanf("%d", &n);
   printf("\nEnter elements of an array:\n");
   for(i=0; i<n; i++)
      scanf("%d", &a[i]);
         for(i=0; i<(n-1); i++)
            {
               for(j=0; j<(n-i-1); j++)
               {
                  if(arr[j]>arr[j+1])
                  {
                     temp = arr[j];
                     arr[j] = arr[j+1];
                     arr[j+1] = temp;
                  }
               }
            }
         printf("\nAfter sorting:\n");
            for(i=0; i<n; i++)
               printf("%d\t", a[i]);
}
```

Input and Output:-

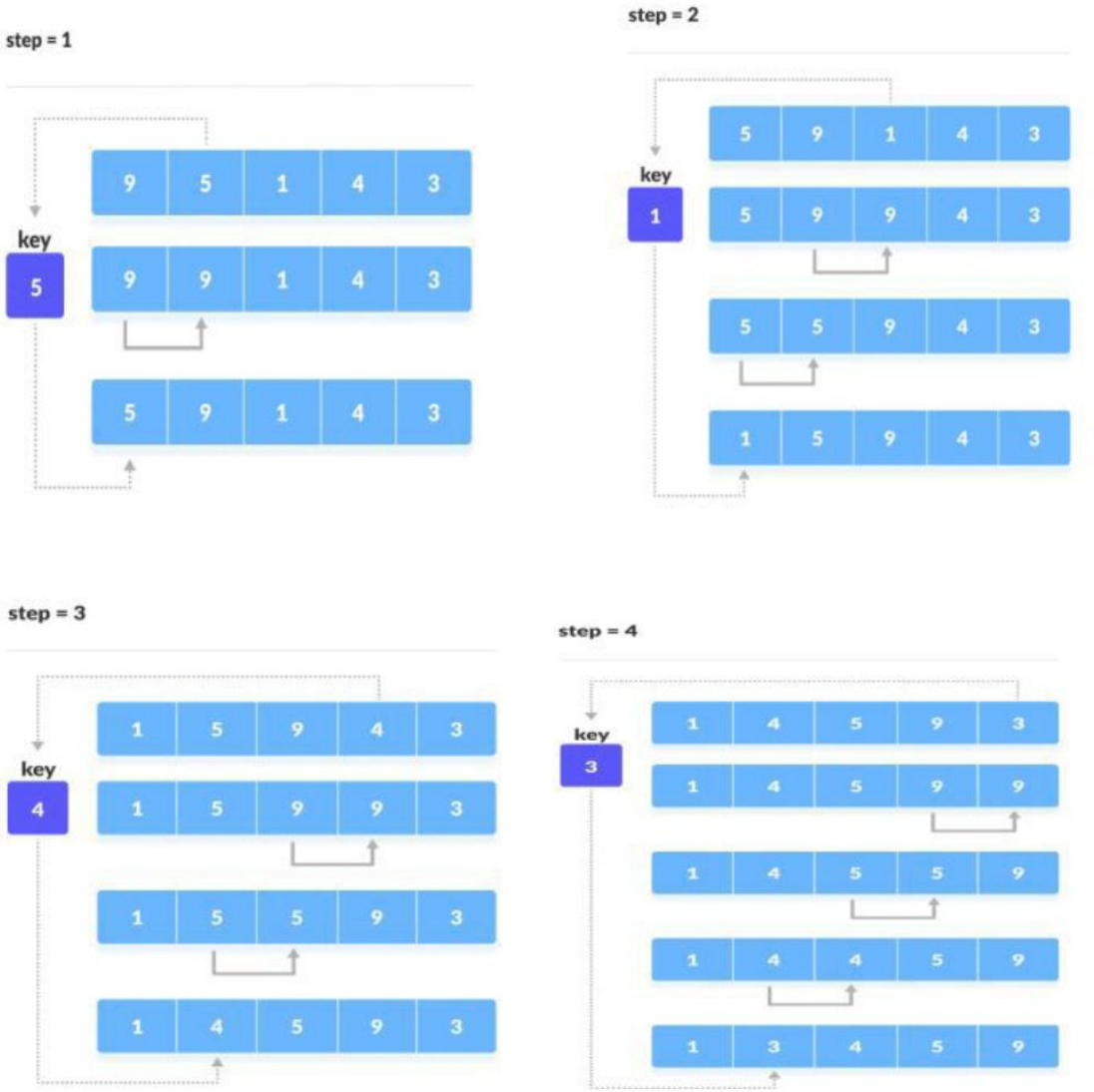
```
Enter size of an array: 8
Enter elements of an array:
68 45 78 14 25 65 55 44
After sorting:
14 25 44 45 55 65 68 78
```

TOPIC : SORTING BY INSERTION

Insertion sort works in the similar way as we sort cards in our hand in a card game.
We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put at their right place.
A similar approach is used by insertion sort.
Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.
Procedure:-

1. The first element in the array is assumed to be sorted. Take the second element and store it separately in key.
   Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.
2. Now, the first two elements are sorted.
   Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the begining of the array.
3. In a similar way, place every unsorted element at its correct position.

Example:- Insertion sort



Program for insertion sort:-

```
void main( )
{
   int a[50],n,i,j,key;
   printf("\nEnter size of an array: ");
   scanf("%d", &n);
   printf("\nEnter elements of an array:\n");
   for(i=0; i<n; i++)
      scanf("%d", &a[i]);
         for (i = 1; i < n; i++)
            {
               key = a[i];
               j = i - 1;
               while (j >= 0 && a[j] > key)
               {
                  a[j + 1] = a[j];
                  j = j - 1;
               }
               a[j + 1] = key;
            }
         printf("\nAfter sorting:\n");
            for(i=0; i<n; i++)
               printf("%d\t", a[i]);
}
```

Input and Output:-

```
Enter size of an array: 8
Enter elements of an array:
68 45 78 14 25 65 55 44
After sorting:
14 25 44 45 55 65 68 78
```

TOPIC : LINEAR SEARCH

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful and the process returns the location of that element, otherwise the search is called unsuccessful.

There are two popular search methods that are widely used in order to search some item into the list.

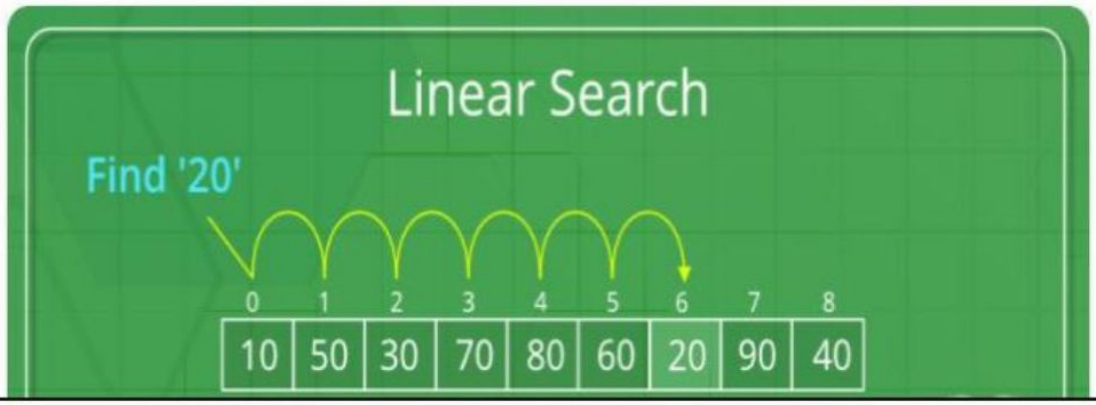
- Linear Search
- Binary Search

Linear Search:- Linear search is the simplest search algorithm and often called sequential search.

Procedure for Linear search:

- Start from the leftmost element of array and one by one compare x with each element of array.
- If x matches with an element, return the index.
- If x doesn't match with any of elements, return -1.

Example:-



Program for Linear Search

```
#include<stdio.h>
main( )
{
        int a[20],n,i,key,x;
        clrscr( );
        printf("How many elements ");
        scanf("%d",&n);
        printf("Enter %d elements\n",n);
        for(i=0;i<n;i++)
        scanf("%d",&a[i]);
        printf("Enter the element to search ");
        scanf("%d",&key);
        x=ls(a,n,key);
        if(x==-1)
        printf("Element is not found");
        else
        printf("Element %d is found at %d location",key,x);
}
ls(int a[ ],int  n, int key)
{
        int i;
        for(i=0;i<n;i++)
        {
                if(a[i]==key)
                return i;
        }
        return -1;
}
```

Input and Output:-

How many elements 10
Enter 10 elements
5 7 2 8 9 20 10 6 1 3
Enter the element to search 20
Element 20 is found at 5 location

TOPIC : BINARY SEARCH

Binary search is the most popular Search algorithm. Binary Search is applied on the sorted array or list of large size.

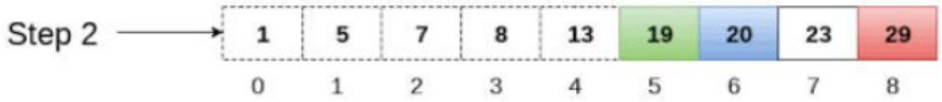
Procedure:-

1. Start with the middle element:
    - If the **target** value is equal to the middle element of the array, then return the index of the middle element.
    - If not, then compare the middle element with the target value,
        - If the target value is greater than the number in the middle index, then pick the elements to the right of the middle index, and start with Step 1.
        - If the target value is less than the number in the middle index, then pick the elements to the left of the middle index, and start with Step 1.
2. When a match is found, return the index of the element matched.
3. If no match is found, then return -1

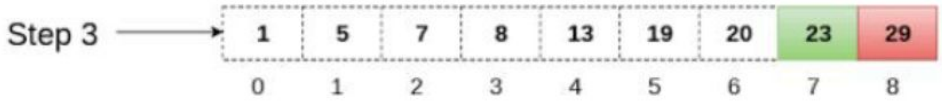
Example:- Binary Search

Item to be searched = 23

a [mid] = 13
13 < 23
beg = mid + 1 = 5
end = 8
mid = (beg + end)/2 = 13 / 2 = 6



a [mid] = 20
20 < 23
beg = mid + 1 = 7
end = 8
mid = (beg + end)/2 = 15 / 2 = 7



a [mid] = 23
23 = 23
loc = mid

Return location 7

Program for binary search

```
#include <stdio.h>
int main()
{
 int i, low, high, middle, n, search, a[100];

 printf("\nEnter size of an array: ");
 scanf("%d", &n);
 printf("\nEnter elements of an array:\n");
 for(i=0; i<n; i++)
     scanf("%d", &a[i]);
 printf("Enter value to find: ");
 scanf("%d", &search);

 low = 0;
 high = n - 1;
 mid = (low+high)/2;

 while (low <= high)

{
  if (a[mid] < search)
    low = mid + 1;
  else if (a[mid] == search)

         {
    printf("%d found at location %d.\n", search, mid+1);
    break;
   }
  else
    high = mid - 1;

  mid = (low+high)/2;
 }
 if (low > high)
  printf("Not found! %d isn't present in the list.\n", search);

 return 0;
}
```

Input and Output:-

Enter size of an array: 8
Enter elements of an array:
68 45 78 14 25 65 55 44
Enter value to find:11
11 found at location 5