

## **Unit:- 5 Tree (Part:-1)**

- Objectives
- Properties of a tree
- Binary trees
  - o Properties of binary trees
  - o Implementation
  - o Traversals of a binary tree
    - In order traversal
    - Post order traversal
    - Preorder traversal
- Binary search trees (bst)
  - o Insertion in bst
  - o Deletion of a node
  - o Search for a key in bst
- Height balanced tree
- B-tree Algorithm
  - o Insertion, Deletion

## **Unit :- 5 Graph (Part:-2)**

- Adjacency matrix and adjacency lists
- Graph traversal
  - o Depth First Search (DFS)
  - o Implementation
  - o Breadth First Search (BFS)
  - o Implementation
- Shortest path problem
- Minimal spanning tree



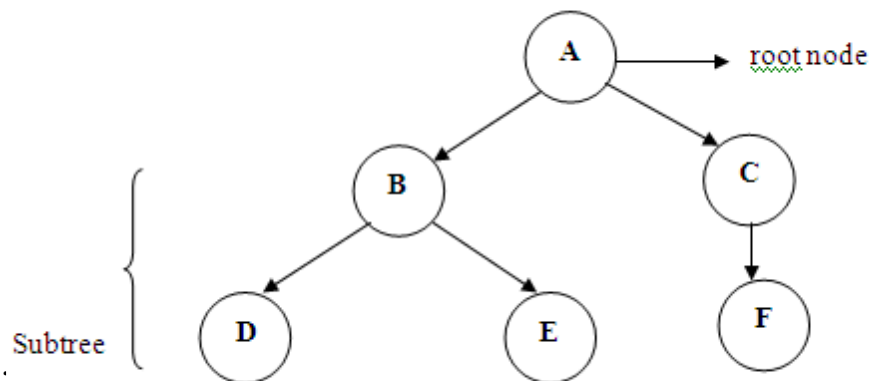
## Objectives

1) What is tree? What is root of the node?

Ans:

Tree is a non-linear data structure which is a set of one or more nodes such that:

- 1) **There is specially designated node known as root of the tree.**
- 2) The remaining nodes are divided into  $n$  disjoint set of nodes  $T_1, T_2, \dots, T_n$ , each of which is a tree



▪ The above figure shows tree

which is divided into two disjoint sets.

{B,D,E} and {C,F}

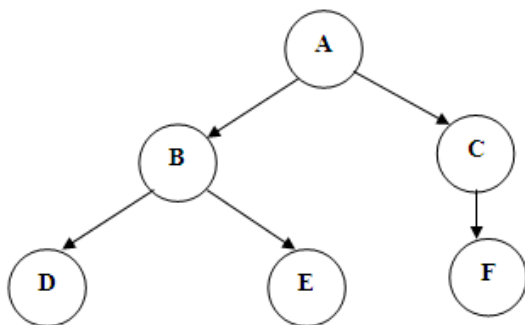
2) What are leaf and non-leaf nodes?

Ans:

♦ **Leaf nodes are the nodes that do not have any children.**

♦ **Non-leaf nodes are the nodes that have children.**

Example:



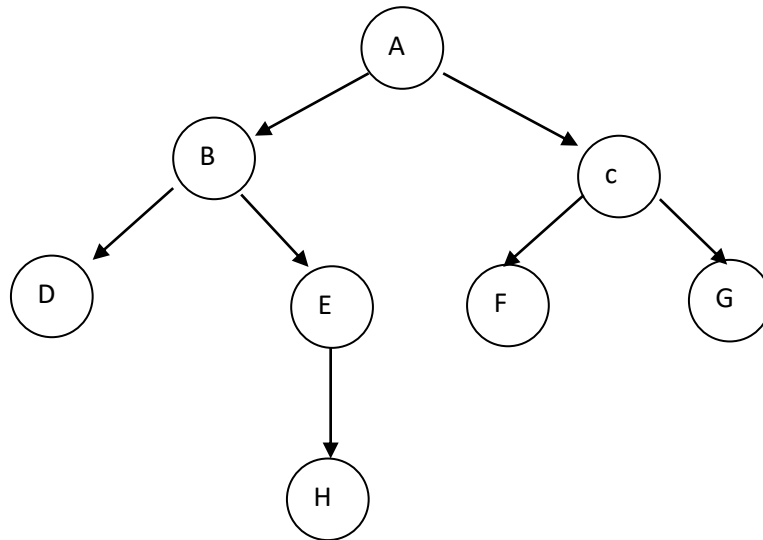
In the above example, nodes D, E and F are leaf nodes and nodes A, B and c are non-leaf nodes.

3) What are ancestor and descendant?

Ans:

♦ Every node which is **parent to leaf and non-leaf nodes is known as ancestor.**

Example:



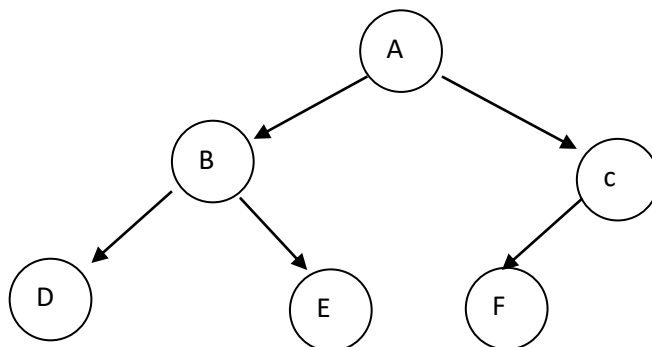
- In the above example, A is known as ancestor of E.
- All the nodes you can reach from the given node is known as descendant.
- In the above example, node H is known as descendant of node C.

4) What is the meaning of Siblings?

Ans:-

♦ Siblings are the **nodes that share the same parent node.**

Example:



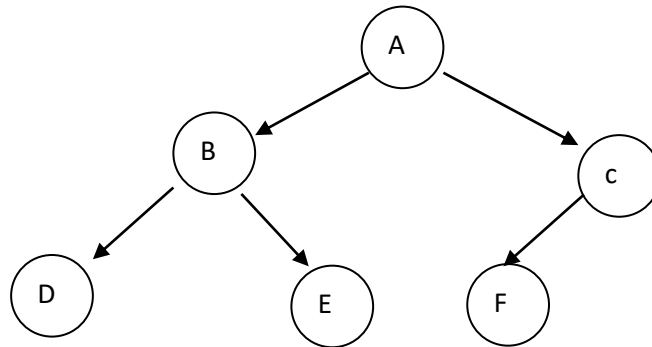
In the above example, nodes B and C are known as siblings.

5) What is the meaning of in-degree of vertex?

Ans:

◆ In-degree of vertex is defined as number of edges (lines) arriving at node.

Example:

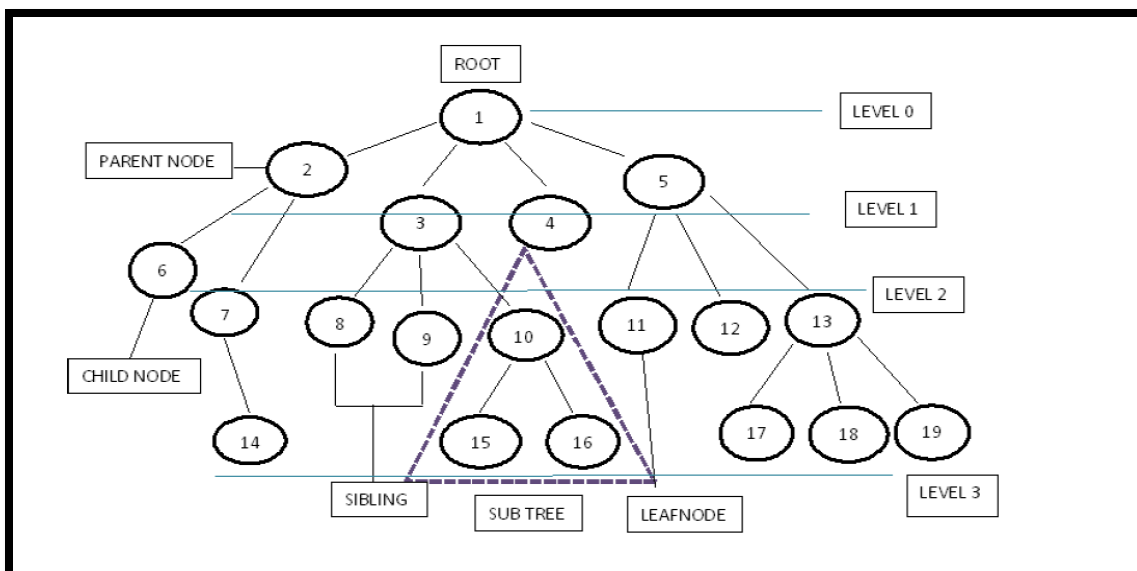


- In the above example, all the nodes except A have in-degree 1.

Note: Root is the only tree which have in-degree 0.

### Properties of a tree

- A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.
- Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially.
- As the size of data increases, the time to access each data element increases linearly.
- The tree data structure allows quicker and easier access to the data as it is a non-linear datastructure.



## Basic Terminology In Tree Data Structure:

**Path** — Path refers to the sequence of nodes along the edges of a tree.

**Parent Node:** The node which is a predecessor of a node is called the parent node of that node. {2} is the parent node of {6, 7}.

**Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: {6, 7} are the child nodes of {2}.

**Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {1} is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.

**Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. {6, 14, 8, 9, 15, 16, 4, 11, 12, 17, 18, 19} are the leaf nodes of the tree.

**Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {1, 2} are the parent nodes of the node {7}

**Descendant:** Any successor node on the path from the leaf node to that node. {7, 14} are the descendants of the node. {2}.

**Sibling:** Children of the same parent node are called siblings. {8, 9, 10} are called siblings.

**Depth of a node:** The count of edges from the root to the node. Depth of node {14} is 3.

**Height of a node:** The number of edges on the longest path from that node to a leaf. Height of node {3} is 2.

**Height of a tree:** The height of a tree is the height of the root node i.e the count of edges from the root to the deepest node. The height of the above tree is 3.

**Level of a node:** The count of edges on the path from the root node to that node. The root node has level 0.

**Internal node:** A node with at least one child is called Internal Node.

**Neighbor of a Node:** Parent or child nodes of that node are called neighbors of that node.  
**Subtree:** Subtree represents the descendants of a node.

**Visiting node** — Visiting refers to checking the value of a node when control is on the node.

**node.keys** — Key represents a value of a node based on which a search operation is to be carried out for a node.

**Traversing** — Traversing means passing through nodes in a specific order.

**Levels** — Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.



Binary tree can be implemented by two methods:-

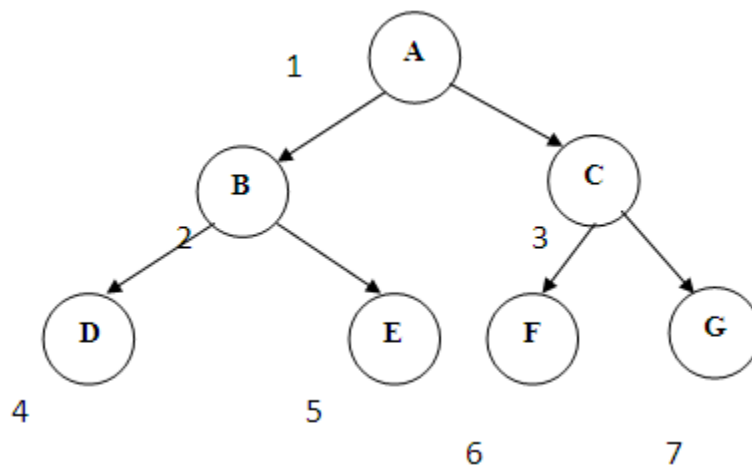
1) Sequential representation (Linear)

## 2) Linked List representation (Non-linear)

### 1) Sequential Representation:-

- ◆ In sequential representation, **nodes are stored in one dimension array**. Also assign level number to every node in tree.
- ◆ We can assign node numbers in such a way that root is assigned the number 1. Then, left sub tree must be assigned  $2p+1$  and right sub tree must be assigned  $2p+2$ .

Example:



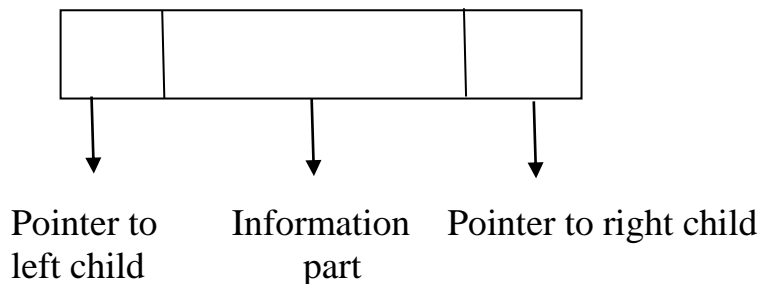
The above tree is represented in the array as:

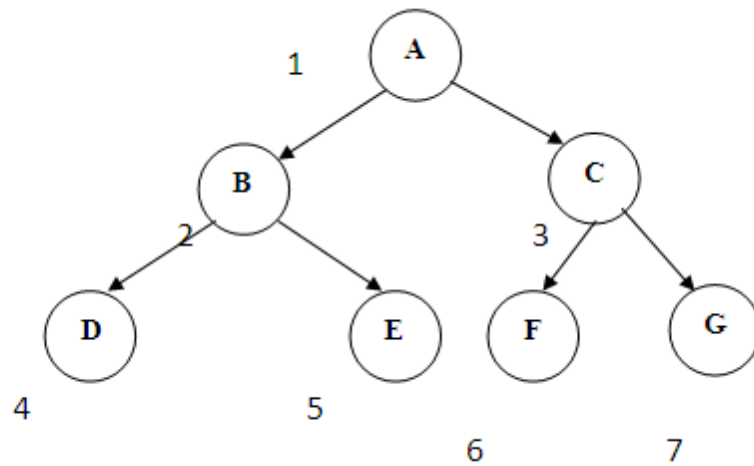
0 1 2 3 4 5 6

A B C D E F G

### 2) Linked List Representation:-

- ◆ In linked list representation, **every node is stored separately and keeps the address of other nodes.**





```

struct node
{
int data;
struct node *lefttree;
struct node *righttree;
};
  
```

### ✚ Traversals of a binary tree

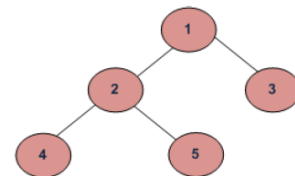
- Tree traversal is the method or technique to display data in a binary tree.
- Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways.
- In the traversing method, the tree is processed in such a way that each node is visited only once.

Following are the generally used ways for traversing trees.

#### 1) Depth First Traversals:

- o Inorder (Left, Root, Right) : 4 2 5 1 3
- o Preorder (Root, Left, Right) : 1 2 4 5 3
- o Postorder (Left, Right, Root) : 4 5 2 3 1

#### 2) Breadth First or Level Order Traversal : 1 2 3 4 5



#### **Inorder Traversal(left subtree, root, right subtree):**

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

#### **Preorder Traversal (root, left subtree, right subtree):**

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Example: Preorder traversal for the above given figure is 1 2 4 5 3.

### **Postorder Traversal (left subtree, right subtree, root):**

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

Example: Postorder traversal for the above given figure is 4 5 2 3 1.

### **One more example :**

One more example :

InOrder(root) visits nodes in the following order:

4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



## **Binary search trees (bst)**

### **Binary Search Tree**

- Binary Search tree can be defined as a class of binary trees, in which the nodes are arranged in a specific order. This is also called an ordered binary tree.
- In a binary search tree, the value of all the nodes in the left sub-tree is less than the value of the root.
- Similarly, the value of all the nodes in the right subtree is greater than or equal to the value of the root.
- This rule will be recursively applied to all the left and right subtrees of the root.

### **Advantages of using binary search tree**

1. Searching becomes very efficient in a binary search tree since we get a hint at each step, about which sub-tree contains the desired element.
2. The binary search tree is considered as an efficient data structure in comparison to arrays and linked lists. In the searching process, it removes half a sub-tree a every step.
3. It also speeds up the insertion and deletion operations as compared to that in array and linked list.

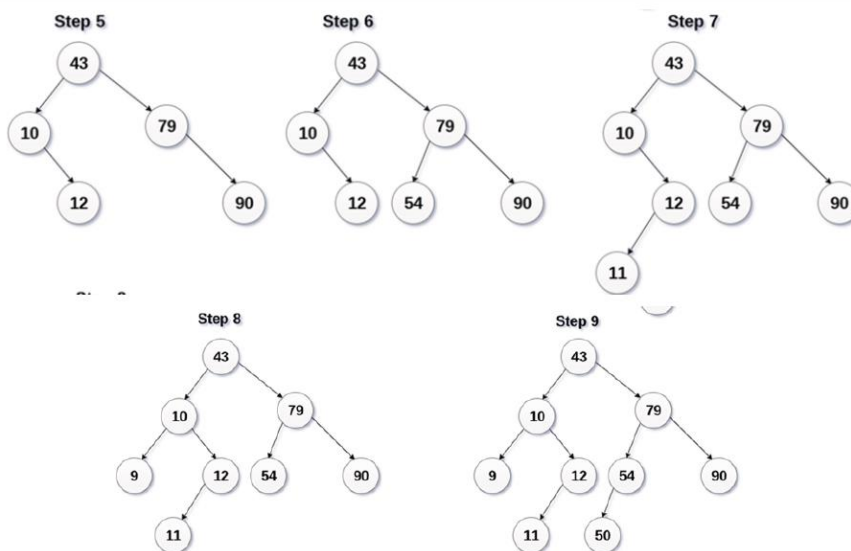
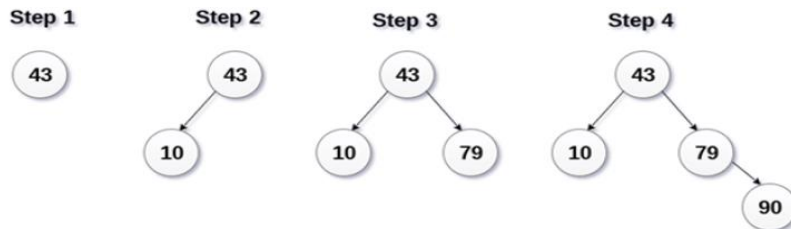


Create the binary search tree using the following data elements.

43, 10, 79, 90, 12, 54, 11, 9, 50

1. Insert 43 into the tree as the root of the tree.
2. Read the next element, if it is less than the root node element, insert it as the root of the left sub-tree.
3. Otherwise, insert it as the root of the right of the right subtree.

The process of creating BST by using the given elements, is shown in the image below.



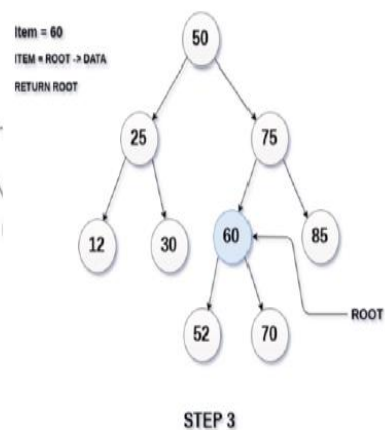
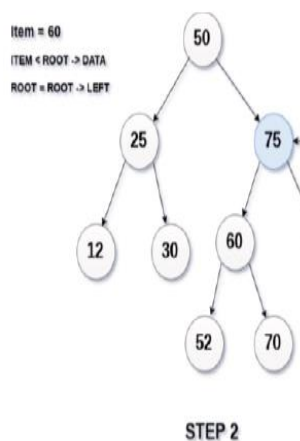
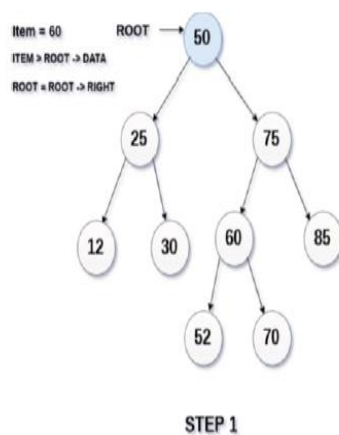
- Binary search trees (bst)
  - o Insertion in bst
  - o Deletion of a node
  - o Search for a key in bst

**Search for a key in bst :** Searching means finding or locating some specific element or node within a data structure.

However, searching for some specific node in a binary search tree is pretty easy due to

the fact that elements in BST are stored in a particular order.

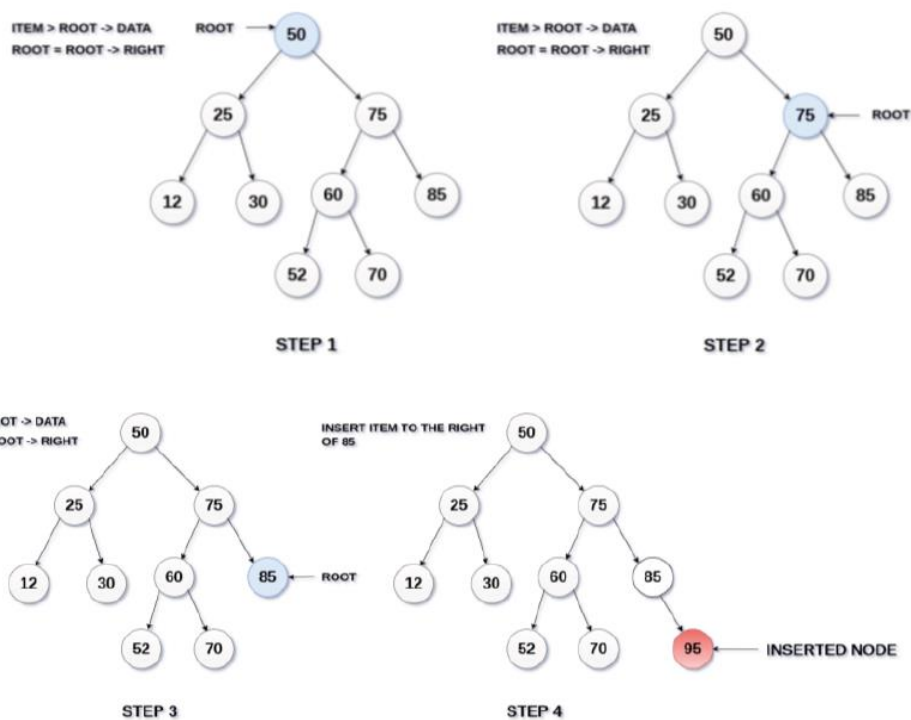
1. Compare the element with the root of the tree.
2. If the item is matched then return the location of the node.
3. Otherwise check if the item is less than the element present on root, if so then move to the left sub-tree.
4. If not, then move to the right subtree.
5. Repeat this procedure recursively until a match is found.
6. If an element is not found then return NULL.



## Insertion in bst:

- Insert function is used to add a new element in a binary search tree at appropriate location. Insert function is to be designed in such a way that it must violate the property of binary search tree at each value.

1. Allocate the memory for trees.
2. Set the data part to the value and set the left and right pointer of the tree, pointing to NULL.
3. If the item to be inserted, will be the first element of the tree, then the left and right of this node will point to NULL.
4. Else, check if the item is less than the root element of the tree, if this is true, then recursively perform this operation with the left of the root.
5. If this is false, then perform this operation recursively with the right subtree of the root.



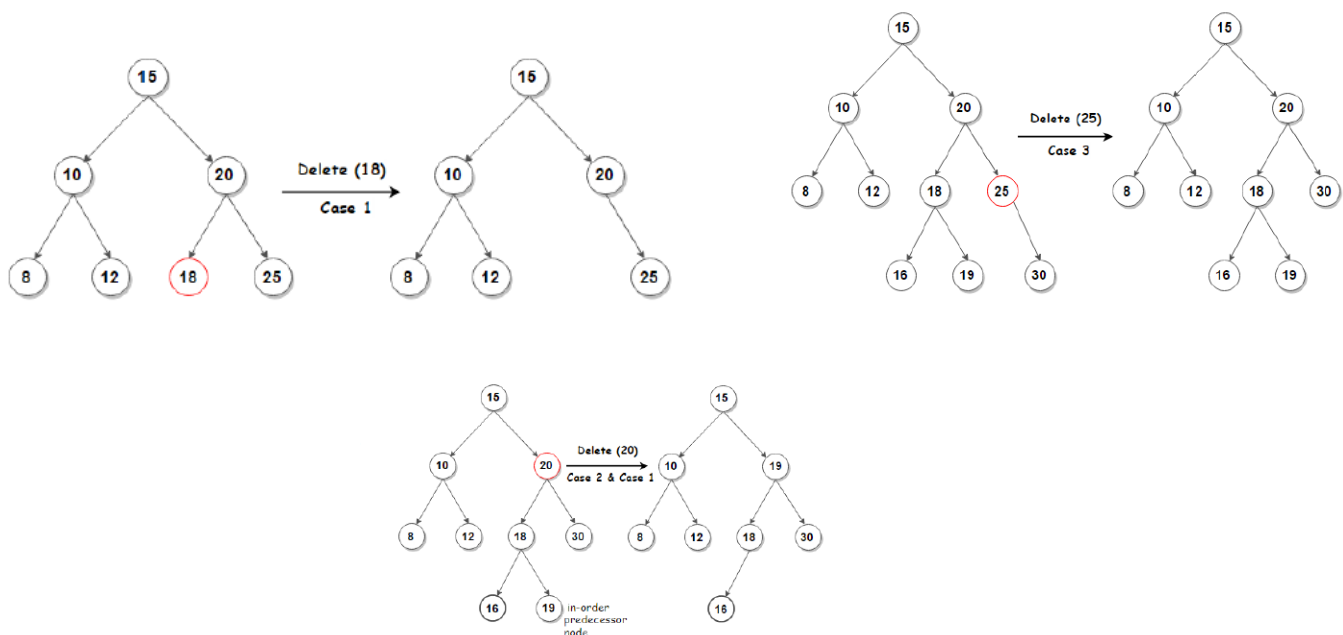
**Deletion :** Delete function is used to delete the specified node from a binary search tree. However, we must delete a node from a binary search tree in such a way that the property of the binary search tree doesn't violate. There are three situations of deleting a node from a binary search tree. The node to be deleted is a leaf node It is the simplest case, in this case, to replace the leaf node with the NULL and simply free the allocated space.

In the following image, we are deleting the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.

**Case 1:** Deleting a node with no children: remove the child node from the tree. It is the simplest case, in this case, to replace the leaf node with the NULL and simply free the allocated space.

**Case 2:** Deleting a node with two children: call the node to be deleted N. Do not delete N. Instead, choose either its inorder successor node or its inorder predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.

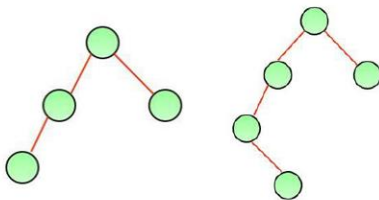
**Case 3 :** Deleting a node with one child: remove the node and replace it with its child.



## Height Balanced Tree :

- AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree.
- AVL Tree was invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honor of its inventors.
- AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.
- Tree is said to be balanced if the balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.
- If the balance factor of any node is 1, it means that the left sub-tree is one level higher than the right subtree. If balance factor of any node is 0, it means that the left subtree and right subtree contain equal height.
- If the balance factor of any node is -1, it means that the left sub-tree is one level lower than the right subtree.
- In an AVL tree, every node maintains an extra information known as **balance factor**.
- A balanced binary tree, also referred to as a height-balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of any node differ by not more than 1.
- An empty tree is height-balanced.
- Non-empty binary tree T is balanced if: 1) Left subtree of T is balanced 2) Right subtree of T is balanced and 3) The difference between heights of left subtree and right subtree is not more than 1.
- The balance factor of a node is calculated either height of left subtree - height of right subtree (OR) height of right subtree - height of left subtree. In the following explanation, we calculate as follows...

**Balance factor = heightOfLeftSubtree - heightOfRightSubtree**



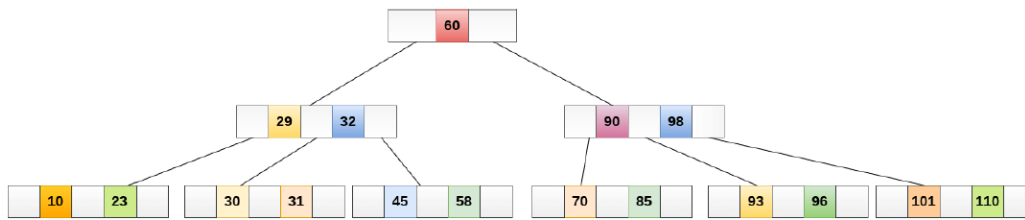
- The above height-balancing scheme is used in AVL trees. The diagram below shows two trees, one of them is height-balanced and other is not. The second tree is not height-balanced because the height of the left subtree is 2 more than the height of the right subtree. To check if a tree is height-balanced, get the height of left and right subtrees. Return true if the difference between heights is not more than 1 and left and right subtrees are balanced, otherwise return false.

● **Every AVL Tree is a binary search tree but every Binary Search Tree need not be an AVL tree.**

- B-tree Algorithm
  - o Insertion, Deletion

### **B - Tree :**

- A B-tree is a tree data structure that keeps data sorted and allows searches, insertions, and deletions.
- Unlike self-balancing binary search trees, it is optimized for systems that read and write large blocks of data. It is most commonly used in database and file systems.
- One of the main reasons for using B tree is its capability to store a large number of keys in a single node and large key values by keeping the height of the tree relatively small.
- In the B-tree data is sorted in a specific order, with the lowest value on the left and the highest value on the right.
- Unlike a binary tree, in B-tree, a node can have more than two children.
- B-tree has a height of  $\log_M N$  (Where 'M' is the order of tree and N is the number of nodes). And the height is adjusted automatically at each update. To insert the data or key in B-tree is more complicated than binary tree.
- In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to the main memory access time.
- The main idea of using B-Trees is to reduce the number of disk accesses.
- Most of the tree operations (search, insert, delete, max, min, ..etc ) require  $O(h)$  disk accesses where h is the height of the tree.
- The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, the B-Tree node size is kept equal to the disk block size. Since the height of the B-tree is low, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, ..etc.
- While performing some operations on B Tree, any property of B Tree may violate such as the number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.



A B tree of order  $m$  contains all the properties of an  $M$  way tree. In addition, it contains the following properties.

- The root nodes must have at least 2 nodes.
- All leaf nodes must be at the same level.
- Above the leaf nodes of the B-tree, there should be no empty sub-trees.
- B- tree's height should lie as low as possible.

#### **Traversal in B-Tree:**

Traversal is also similar to Inorder traversal of Binary Trees. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for remaining children and keys. In the end, recursively print the rightmost child.

#### **Inserting in B-Tree :**

Insertions are done at the leaf node level. The following algorithm needs to be followed

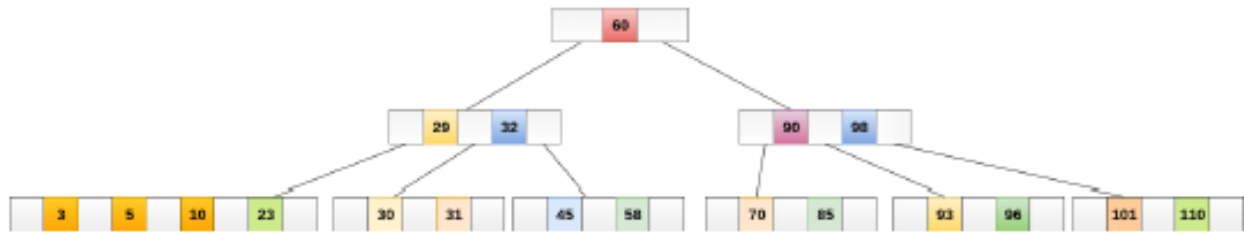
in order to insert an item into B Tree.

1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contains less than  $m-1$  keys then insert the element in the increasing order.
3. Else, if the leaf node contains  $m-1$  keys, then follow the following steps.

Insert the new element in the increasing order of elements.

- Split the node into the two nodes at the median.
- Push the median element up to its parent node.
- If the parent node also contains  $m-1$  number of keys, then split it too by following the same steps.

**Example:** Insert the node 8 into the B Tree of order 5 shown in the following image.



8 will be inserted to the right of 5, therefore insert 8.



The node now contains 5 keys which is greater than  $(5 - 1 = 4)$  keys. Therefore split the node from the median i.e. 8 and push it up to its parent node shown as follows.



## Deletion in B-Tree :

Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node.

Following algorithm needs to be followed in order to delete a node from a B tree.

1. Locate the leaf node.
2. If there are more than  $m/2$  keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain  $m/2$  keys then complete the keys by taking the element from the right or left sibling.
  - If the left sibling contains more than  $m/2$  elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
  - If the right sibling contains more than  $m/2$  elements then push its smallest element up to the parent and move the intervening element down to the node where the key is deleted.



4. If neither of the siblings contain more than  $m/2$  elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.

5. If the parent is left with less than  $m/2$  nodes then, apply the above process on the parent too.

If the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, the successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

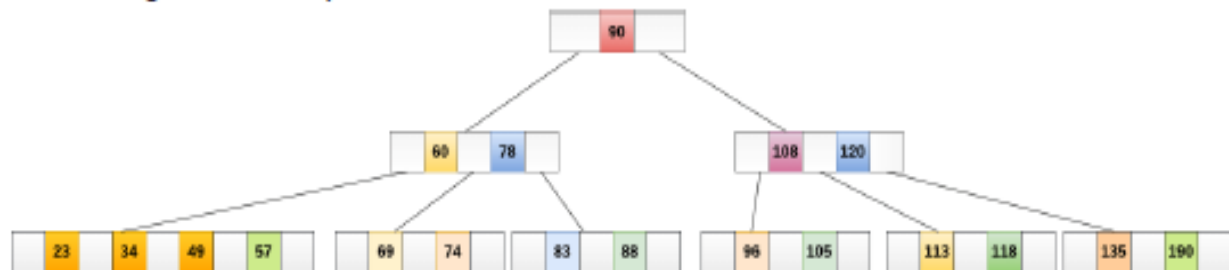
**Example 1:** Delete the node 53 from the B Tree of order 5 shown in the following figure.



53 is present in the right child of element 49. Delete it.



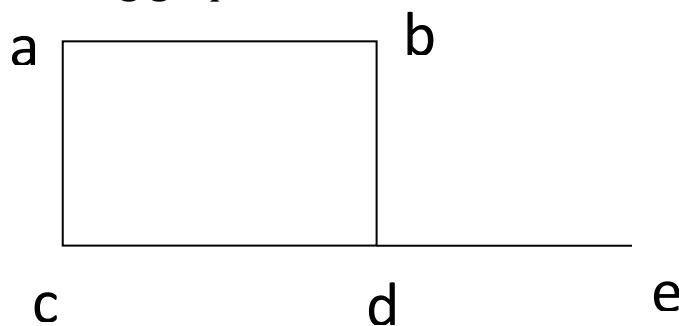
Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right subtree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49. The final B tree is shown as follows.



- A Graph is a non-linear data structure consisting of nodes and edges.
- The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.

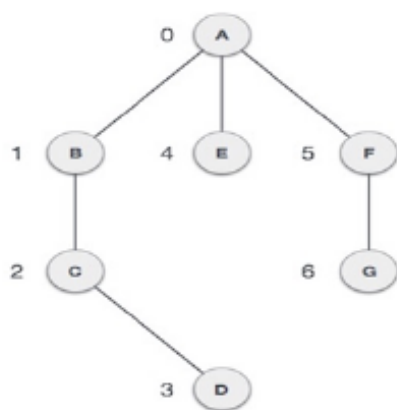
- **A Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes**

Formally, a graph is a pair of sets  $(V, E)$ , where  $V$  is the set of vertices and  $E$  is the set of edges, connecting the pairs of vertices. Take a look at the following graph



In the above graph,  
 $V = \{a, b, c, d, e\}$   
 $E = \{ab, ac, bd, cd, de\}$

- Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.
- We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms



**Vertex** – Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.

**Edge** – Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represent edges. We can use a two-dimensional array to represent an array as shown in the following

image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

**Adjacency** (□□□□□,□□ □□□□□)□□ – Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.

**Path** – Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.

#### **Basic Operations on Graph :**

- **Add Vertex :** Adds a vertex to the graph.
- **Remove Vertex :** Removes a vertex to the graph.
- **Add Edge:** Adds an edge between the two vertices of the graph.
- **Remove Edge :** Removes an edge between the two vertices of the graph.
- **Display Vertex :** Displays a vertex of the graph.
- **Breadth-First Search (BFS)**
- **Depth First Search (DFS)**

#### **Adjacency Matrix and Adjacent Lists :**

- A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.
- Graph is a mathematical structure and finds its application in many areas, where the problem is to be solved by computers.
- The problem related to graph G must be represented in computer memory using any suitable data structure to solve the same.
- There are two standard ways of maintaining a graph G in the memory of a computer.

#### **Adjacency Matrix:**

- Adjacency Matrix is a 2D array of size  $V \times V$  where  $V$  is the number of vertices in a graph. Let the 2D array be  $adj[][]$ , a slot  $adj[i][j] = 1$  indicates that there is an edge

**Path** – Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.

#### **Basic Operations on Graph :**

- **Add Vertex :** Adds a vertex to the graph.
- **Remove Vertex :** Removes a vertex to the graph.
- **Add Edge:** Adds an edge between the two vertices of the graph.
- **Remove Edge :** Removes an edge between the two vertices of the graph.

- **Display Vertex :** Displays a vertex of the graph.
- **Breadth-First Search (BFS)**
- **Depth First Search (DFS)**

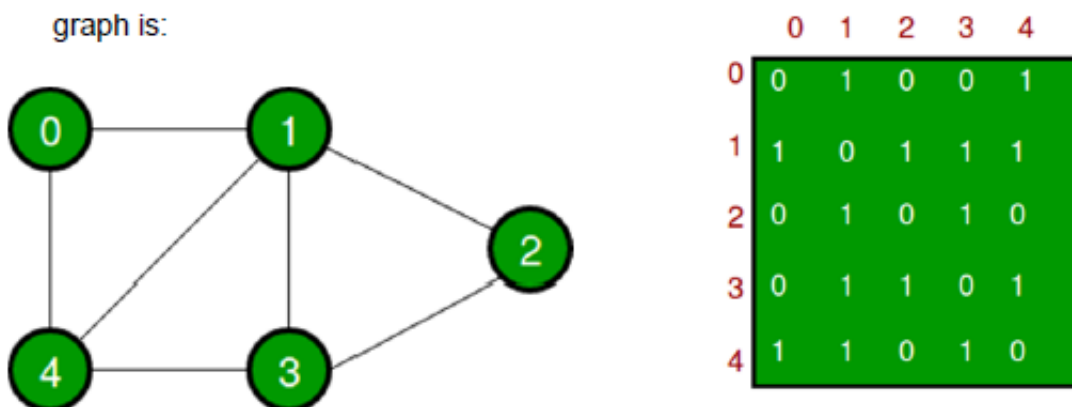
#### **Adjacency Matrix and Adjacent Lists :**

- A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.
- Graph is a mathematical structure and finds its application in many areas, where the problem is to be solved by computers.
- The problem related to graph G must be represented in computer memory using any suitable data structure to solve the same.
- There are two standard ways of maintaining a graph G in the memory of a computer.

#### **Adjacency Matrix:**

- Adjacency Matrix is a 2D array of size  $V \times V$  where  $V$  is the number of vertices in a
- graph. Let the 2D array be  $adj[][]$ , a slot  $adj[i][j] = 1$  indicates that there is an edge
- from vertex  $i$  to vertex  $j$ .
- Adjacency matrix for an undirected graph is always symmetric. (□□□□□□)
- The adjacency matrix for the above example

graph is:



#### **Linked representation(i.e. Adjacency list representation) :**

- An Adjacency list is an array consisting of the address of all the linked lists.
- The first node of the linked list represents the vertex and the remaining lists connected to this node represent the vertices to which this node is connected.
- This representation can also be used to represent a weighted graph. The linked list can slightly be changed to even store the weight of the edge.

- An array of lists is used. The size of the array is equal to the number of vertices.
- Let the array be an array[].
- The index of the array represents a vertex and each element in its linked list represents the vertices that form an edge with the vertex.
- An entry array[i] represents the list of vertices adjacent to the ith vertex.
- The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above graph.



#### Graph Traversal :

- Graph traversal is a technique used for a searching vertex in a graph.
- The graph traversal is also used to decide the order of vertices visited in the search process.
- A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into a looping path.

There are two graph traversal techniques and they are as follows...

1. DFS (Depth First Search)
2. BFS (Breadth First Search/level order)

#### 1. Depth First 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 onto the Stack.

**Step 3** - Visit any one of the non-visited adjacent vertices of a vertex which is at the top of the stack and push it onto 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 backtracking and pop one vertex from the stack.

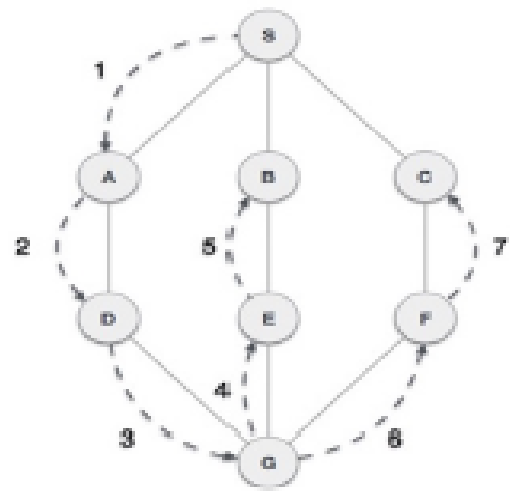
**Step 6** - Repeat steps 3, 4 and 5 until the stack becomes Empty.

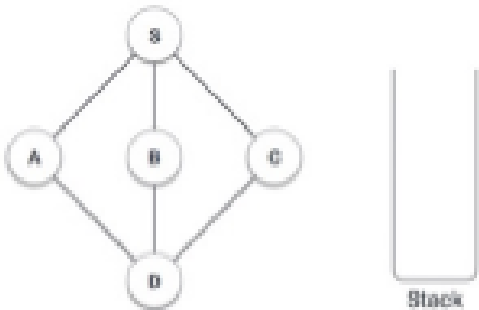
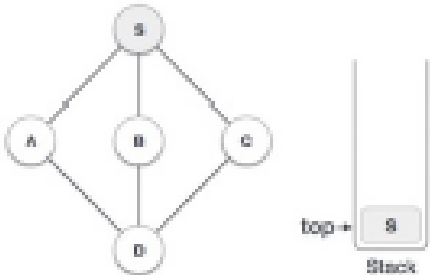
**Step 7** - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

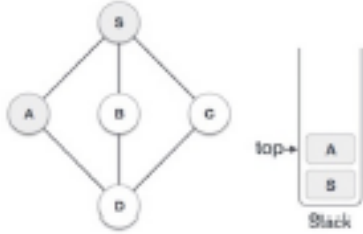
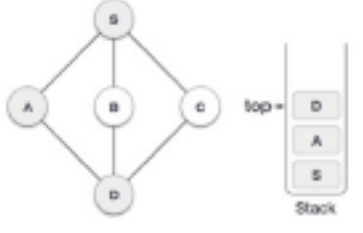
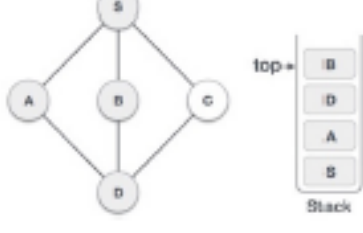
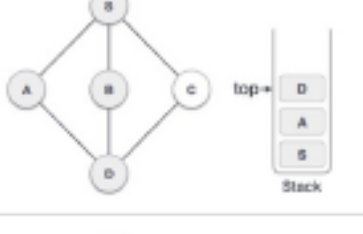
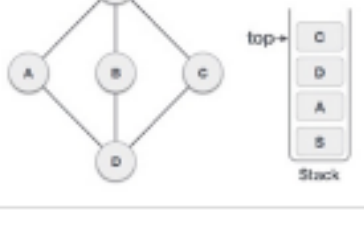
**Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

**Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

**Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

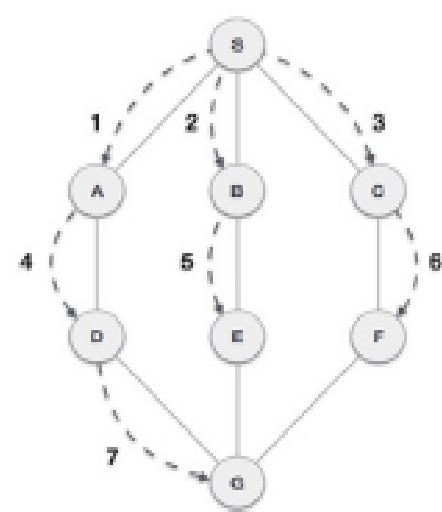


Step	Traversal	Description
1		Initialize the stack.
2		Mark S visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.

3		<p>Mark A visited and put it onto the stack. Explore any unvisited adjacent node from A. Both S and D are adjacent to A but we are concerned for unvisited nodes only.</p>
4		<p>Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.</p>
5		<p>We choose B, mark it as visited and put onto the stack. Here B does not have any unvisited adjacent nodes. So, we pop B from the stack.</p>
6		<p>We check the stack top to return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.</p>
7		<p>Only unvisited adjacent node from D is C now. So we visit C, mark it as visited and put it onto the stack.</p>

As C does not have any unvisited adjacent nodes, we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

**Breadth(DFS) First Traversal :**



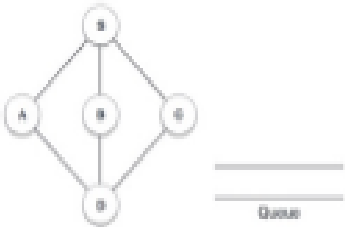
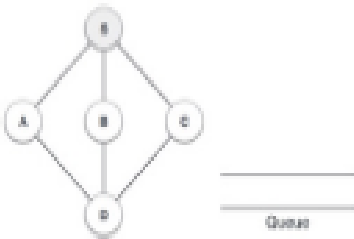
Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

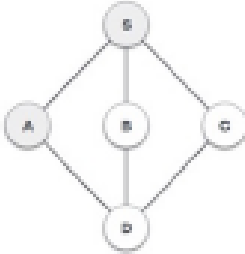

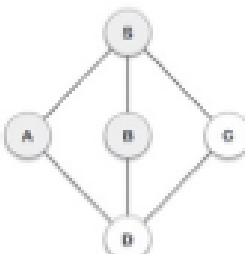
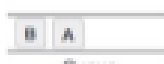
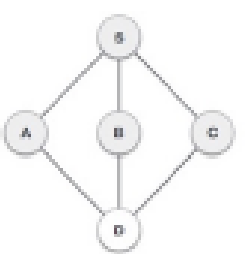

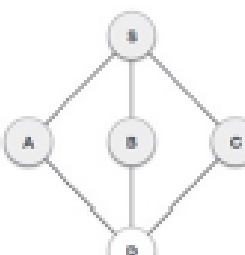

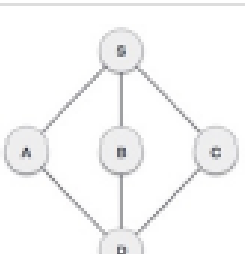

**Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

**Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.

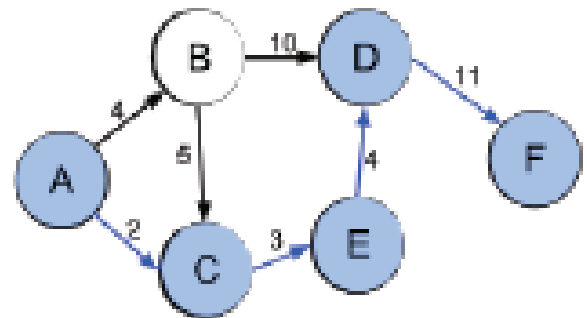
**Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting S (starting node), and mark it as visited.



3	 	We then see an unvisited adjacent node from S. In this example, we have three nodes but alphabetically we choose A, mark it as visited and enqueue it.
4	 	Next, the unvisited adjacent node from S is B. We mark it as visited and enqueue it.
5	 	Next, the unvisited adjacent node from S is C. We mark it as visited and enqueue it.
6	 	Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A.
7	 	From A we have D as unvisited adjacent node. We mark it a

- As we always do when we want to reach from one station to another, at that time the driver takes the shortest possible route to reach the destination.
- There are many instances to find the shortest path for traveling from one place to another.
- That is to find which route can reach as quick as possible of a route for which the traveling cost is minimum.
- In a graph, finding the shortest path is the most important problem.
- In graph theory, the **shortest path problem** is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.



• Shortest path algorithms are applied to automatically find directions between physical locations, such as driving directions on web mapping websites like MapQuest or Google Maps. For this application fast specialized algorithms are available.

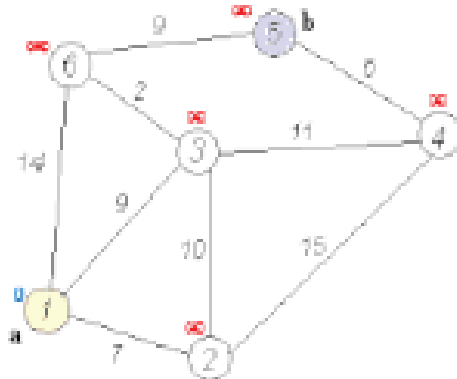
[ Shortest path (A, C, E, D, F) between vertices A and F in the weighted directed graph  
The problem is also sometimes called the **single-pair shortest path problem**, to distinguish it from the following variations: ]

- The **single-source shortest path problem**, in which we have to find shortest paths from a source vertex  $v$  to all other vertices in the graph.
  - The **single-destination shortest path problem**, in which we have to find shortest paths from all vertices in the directed graph to a single destination vertex  $v$ . This can be reduced to the single-source shortest path problem by reversing the arcs in the directed graph.
  - The **all-pairs shortest path problem**, in which we have to find shortest paths between every pair of vertices  $v, v'$  in the graph.
- The most important algorithms for solving this problem are:
- **Dijkstra's algorithm** solves the single-source shortest path problem with non-negative edge weight.
  - **Bellman-Ford algorithm** solves the single-source problem if edge weights may be negative.
  - **A\* search algorithm** solves for single-pair shortest path using heuristics to try to speed up the search.
  - **Floyd-Warshall algorithm** solves all pairs shortest paths.
  - **Johnson's algorithm** solves all pairs' shortest paths, and may be faster than Floyd-Warshall on sparse graphs.

- The Viterbi algorithm solves the shortest stochastic path problem with an additional probabilistic weight on each node.

### DIJKSTRA'S ALGORITHM

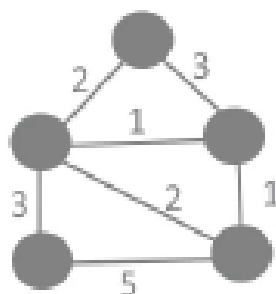
Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.



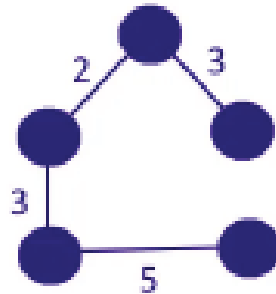
[ Dijkstra's algorithm to find the shortest path between a and b. It picks the unvisited vertex with the lowest distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller. Mark visited (set to red) when done with neighbors. ]

### Minimal Spanning Tree :

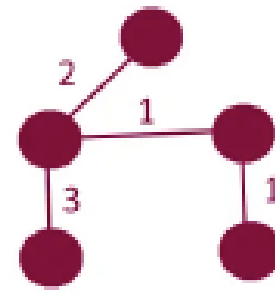
- A spanning tree of a graph is a collection of connected edges that include every vertex in the graph, but that do not form a cycle.
- A spanning tree of a graph is just a sub graph that contains all the vertices and is a tree(with no cycle).
- A graph may have many spanning trees.
- A minimum spanning tree(MST) for a graph is a subgraph of a graph that contains all the vertices of G(graph).
- If a graph G is not a connected graph, then it cannot have any spanning tree.



Graph



Spanning Tree  
Cost = 13



Minimum Spanning  
Tree, Cost = 7