1. Discuss following

1. Graph

- A graph G consist of a non-empty set V called the set of nodes (points, vertices) of the graph, a set E which is the set of edges and a mapping from the set of edges E to a set of pairs of elements of V.
- It is also convenient to write a graph as G=(V,E).
- Notice that definition of graph implies that to every edge of a graph G, we can associate a pair of nodes of the graph. If an edge X Є E is thus associated with a pair of nodes (u,v) where u, v Є V then we says that edge x connect u and v.

2. Adjacent Nodes

- Any two nodes which are connected by an edge in a graph are called adjacent node.

3. Directed & Undirected Edge

- In a graph G=(V,E) an edge which is directed from one end to another end is called a directed edge, while the edge which has no specific direction is called undirected edge.

4. Directed graph (Digraph)

- A graph in which every edge is directed is called directed graph or digraph.

5. Undirected graph

- A graph in which every edge is undirected is called undirected graph.

6. Mixed Graph

- If some of the edges are directed and some are undirected in graph then the graph is called mixed graph.

7. Loop (Sling)

- An edge of a graph which joins a node to itself is called a loop (sling).

8. Parallel Edges

- In some directed as well as undirected graphs, we may have certain pairs of nodes joined by more than one edges, such edges are called Parallel edges.

9. Multigraph

- Any graph which contains some parallel edges is called multigraph.

10. Weighted Graph

- A graph in which weights are assigned to every edge is called weighted graph.

11. Isolated Node

- In a graph a node which is not adjacent to any other node is called isolated node.

12. Null Graph

- A graph containing only isolated nodes are called null graph. In other words set of edges in null graph is empty.

13. Path of Graph

- Let G=(V, E) be a simple digraph such that the terminal node of any edge in the sequence is the initial node of the edge, if any appearing next in the sequence defined as path of the graph.

14. Length of Path

- The number of edges appearing in the sequence of the path is called length of path.

15. Degree of vertex

- The no of edges which have V as their terminal node is call as indegree of node V
- The no of edges which have V as their initial node is call as outdegree of node V
- Sum of indegree and outdegree of node V is called its Total Degree or Degree of vertex.

16. Simple Path (Edge Simple)

- A path in a diagraph in which the edges are distinct is called simple path or edge simple.

17. Elementary Path (Node Simple)

- A path in which all the nodes through which it traverses are distinct is called elementary path.

18. Cycle (Circuit)

- A path which originates and ends in the same node is called cycle (circuit).

19. Directed Tree

- A directed tree is an acyclic digraph which has one node called its root with in degree 0, while all other nodes have in degree 1.
- Every directed tree must have at least one node.
- An isolated node is also a directed tree.

20. Terminal Node (Leaf Node)

- In a directed tree, any node which has out degree 0 is called terminal node or leaf node.

21. Level of Node

- The level of any node is the length of its path from the root.

22. Ordered Tree

- In a directed tree an ordering of the nodes at each level is prescribed then such a tree is called ordered tree.

23. Forest

- If we delete the root and its edges connecting the nodes at level 1, we obtain a set of disjoint tree. A set of disjoint tree is a forest.

24. M-ary Tree

- If in a directed tree the out degree of every node is less than or equal to m then tree is called an m-ary tree.

25. Full or Complete M-ary Tree

- If the out degree of each and every node is exactly equal to m or 0 and their number of nodes at level **i** is $\mathbf{m^{(i-1)}}$ then the tree is called a full or complete m-ary tree.

26. Positional M-ary Tree

- If we consider m-ary trees in which the m children of any node are assumed to have m distinct positions, if such positions are taken into account, then tree is called positional m-ary tree.

27. Height of the tree

- The height of a tree is the length of the path from the root to the deepest node in the tree.

28. Binary tree

- If in a directed tree the out degree of every node is less than or equal to 2 then tree is called binary tree.

29. Strictly binary tree

- A strictly binary tree (sometimes proper binary tree or 2-tree or full binary tree) is a tree in which every node other than the leaves has two children.

30. Complete binary tree

- If the out degree of each and every node is exactly equal to 2 or 0 and their number of nodes at level **i** is $\mathbf{2^{(i-1)}}$ then the tree is called a full or complete binary tree.

31. Sibling

- Siblings are nodes that share the same parent node.

32. Binary search tree

- A binary search tree is a binary tree in which each node possessed a key that satisfy the following conditions

1. All key (if any) in the left sub tree of the root precedes the key in the root.
2. The key in the root precedes all key (if any) in the right sub tree.
3. The left and right sub tree sub trees of the root are again search trees.

33. Height Balanced Binary tree (AVL Tree)

- A tree is called AVL (height balance binary tree), if each node possesses one of the following properties
    1. A node is called left heavy if the longest path in its left sub tree is one longer then the longest path of its right sub tree.
    2. A node is called right heavy if the longest path in the right sub tree is one longer than path in its left sub tree.
    3. A node is called balanced, if the longest path in both the right and left sub tree are equal.

---

2. Explain the Preorder, Inorder and Postorder traversal techniques of the binary tree with suitable example.

- The most common operations performed on tree structure is that of traversal. This is a procedure by which each node in the tree is processed exactly once in a systematic manner.
- There are three ways of traversing a binary tree.
    1. Preorder Traversal
    2. Inorder Traversal
    3. Postorder Traversal

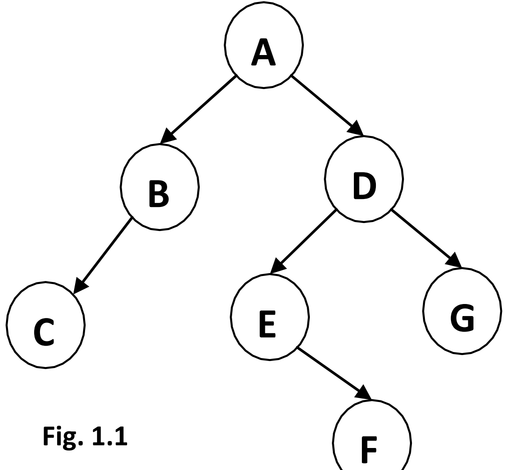


Fig. 1.1

Preorder traversal : A B C D E F G

Inorder traversal : C B A E F D G

Postorder traversal : C B F E G D A

Converse Preorder traversal : A D G E F B C

Converse Inorder traversal : G D F E A B C

Converse Postorder traversal : G F E D C B A

Preorder

- Preorder traversal of a binary tree is defined as follow
  - Process the root node
  - Traverse the left subtree in preorder
  - Traverse the right subtree in preorder
- If particular subtree is empty (i.e., node has no left or right descendant) the traversal is performed by doing nothing, In other words, a null subtree is considered to be fully traversed when it is encountered.
- The preorder traversal of a tree (Fig. 1.1) is given by A B C D E F G

Inorder

- The Inorder traversal of a binary tree is given by following steps,
  - Traverse the left subtree in Inorder
  - Process the root node
  - Traverse the right subtree in Inorder
- The Inorder traversal of a tree (Fig. 1.1) is given by C B A E F D G

Postorder

- The postorder traversal is given by
  - Traverse the left subtree in postorder
  - Traverse the right subtree in postorder
  - Process the root node
- The Postorder traversal of a tree (Fig. 1.1) is given by C B F E G D A

Converse …

- If we interchange left and right words in the preceding definitions, we obtain three new traversal orders which are called
  - Converse Preorder (A D G E F B C)
  - Converse Inorder (G D F E A B C)
  - Converse Postorder (G F E D C B A)

---

3. Write the algorithm of Preorder, Inorder and Postorder traversal techniques of the binary tree.

Procedure : RPREORDER(T)

- Given a binary tree whose root node address is given by pointer variable T and whose node structure is same as described below. This procedure traverses the tree in preorder, in a recursive manner.

1. **[Check for empty Tree]**
   If T = NULL
   then write (‘Empty Tree’)
   return
   else write (DATA(T))

2. **[Process the Left Subtree]**
   If LPTR (T) ≠ NULL
   then RPREORDER (LPTR (T))

3. **[Process the Right Subtree]**
   If RPTR (T) ≠ NULL
   then RPREORDER (RPTR (T))

4. **[Finished]**
   return

Procedure : RINORDER(T)

- Given a binary tree whose root node address is given by pointer variable T and whose node structure is same as described below. This procedure traverses the tree in inorder, in a recursive manner.

1. **[Check for empty Tree]**
   If T = NULL
   then write (‘Empty Tree’)
   return

2. **[Process the Left Subtree]**
   If LPTR (T) ≠ NULL
   then RINORDER (LPTR (T))

3. **[Process the root node]**
   write (DATA(T))

4. **[Process the Right Subtree]**
   If RPTR (T) ≠ NULL
   then RINORDER (RPTR (T))

5. **[Finished]**
   return

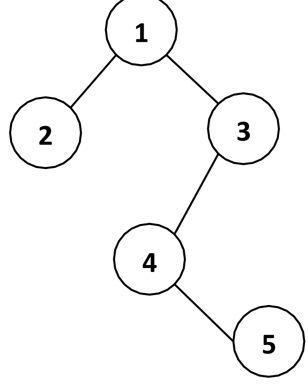
Procedure : RPOSTORDER(T)

- Given a binary tree whose root node address is given by pointer variable T and whose node structure is same as described below. This procedure traverses the tree in postorder, in a recursive manner.

1. **[Check for empty Tree]**
   If T = NULL
   then write ('Empty Tree')
   return
2. **[Process the Left Subtree]**
   If LPTR (T) ≠ NULL
   then RPOSTORDER (LPTR (T))
3. **[Process the Right Subtree]**
   If RPTR (T) ≠ NULL
   then RPOSTORDER (RPTR (T))
4. **[Process the root node]**
   write (DATA(T))
5. **[Finished]**
   return

---

4. Give traversal order of following tree into Inorder, Preorder and Postorder.



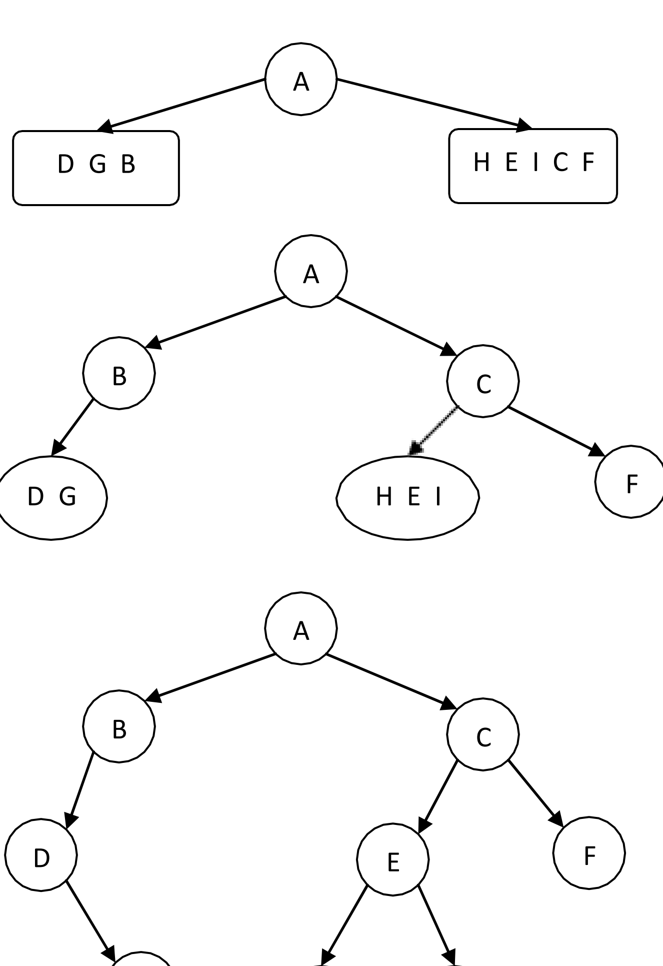
Inorder: 2 1 4 5 3
Preorder: 1 2 3 4 5
Post order: 2 5 4 3 1

---

5. Construct a tree for the given Inorder and Postorder traversals

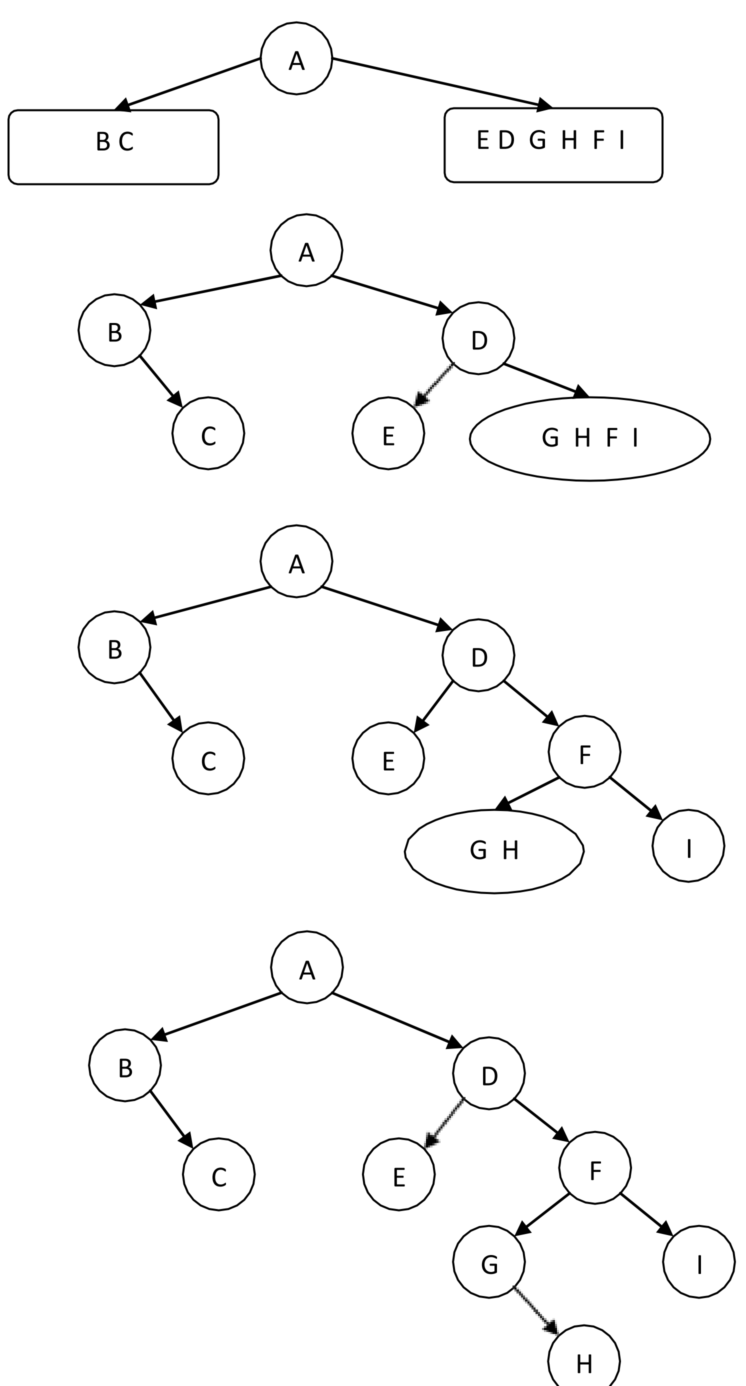
Inorder : D G B A H E I C F

Postorder : G D B H I E F C A



Postorder : C B E H G I F D A

Inorder : B C A E D G H F I



6. Construct a tree for the given Inorder and Preorder traversals

Preorder : G B Q A C K F P D E R H
Inorder : Q B K C F A G P E D H R

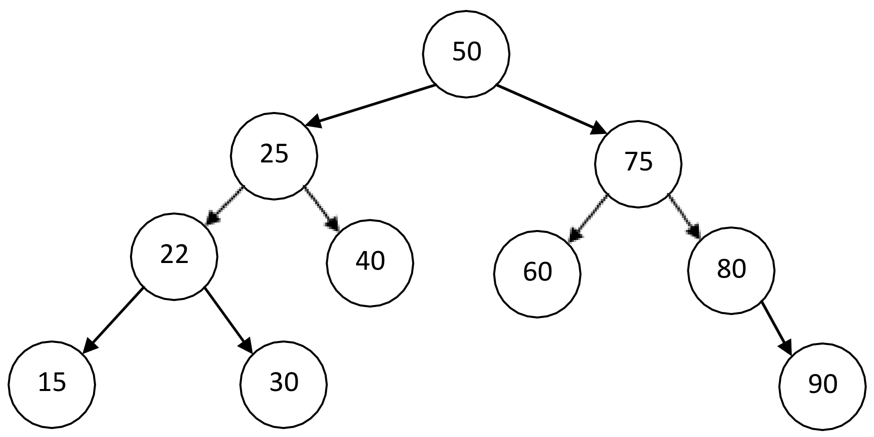
G
Q B K C F A
P E D H R

G
B
P
Q
K C F A
D E R H

G
B
P
Q
A
D
K C F
E
R H

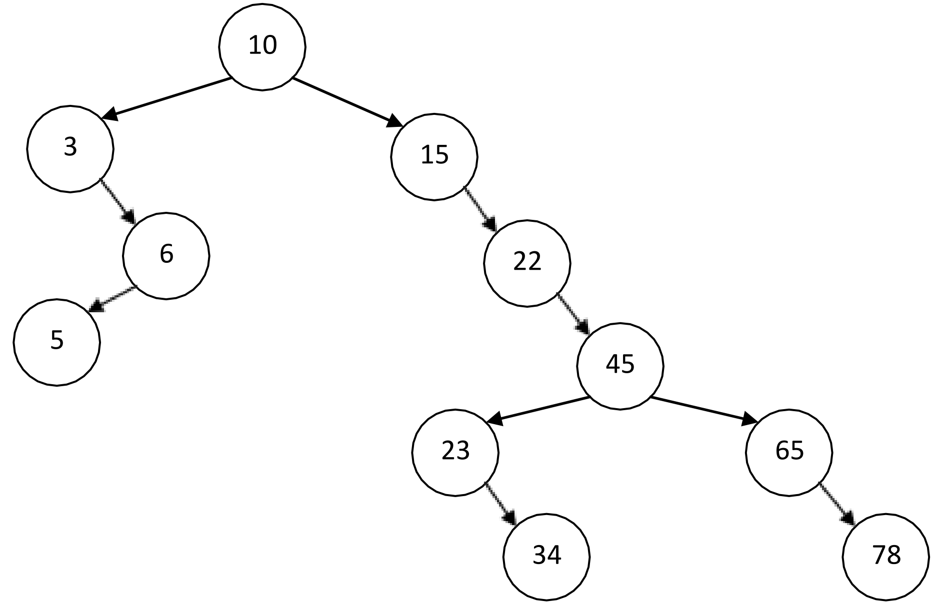
G
B
P
Q
A
D
C
E
R
K
F
H

7. Create a binary search tree for the following data :
50 ,25 ,75, 22,40,60,80,90,15,30



---

8. Construct binary search tree for the following data and find its Inorder, Preorder and Postorder traversal
10,3,15,22,6,45,65,23,78,34,5



Preorder : 10, 3, 6, 5, 15, 22, 45, 23, 34, 65, 78
Inorder : 3, 5, 6, 10, 15, 22, 23, 34, 45, 65, 78
Postorder : 5, 6, 3, 34, 23, 78, 65, 45, 22, 15, 10

9. Write a short note on threaded binary tree

- The wasted NULL links in the binary tree storage representation can be replaced by threads.
- A binary tree is threaded according to particular traversal order. e.g.: Threads for the inorder traversals of tree are pointers to its higher nodes, for this traversal order.
  - If left link of node P is null, then this link is replaced by the address of its predecessor.
  - If right link of node P is null, then it is replaced by the address of its successor
- Because the left or right link of a node can denote either structural link or a thread, we must somehow be able to distinguish them.
- Method 1:- Represent thread a –ve address.
- Method 2:- To have a separate Boolean flag for each of left and right pointers, node structure for this is given below,

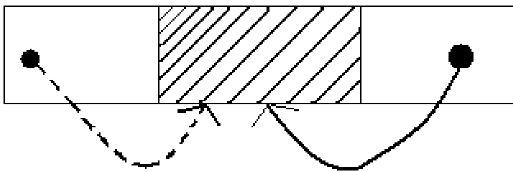
LPTR	LTHREAD	Data	RTHREAD	RPTR

Alternate node for threaded binary tree.

- LTHREAD = true = Denotes leaf thread link
- LTHREAD = false = Denotes leaf structural link
- RTHREAD = true = Denotes right threaded link
- RTHREAD = false = Denotes right structural link

- Head node is simply another node which serves as the predecessor and successor of first and last tree nodes. Tree is attached to the left branch of the head node

Head



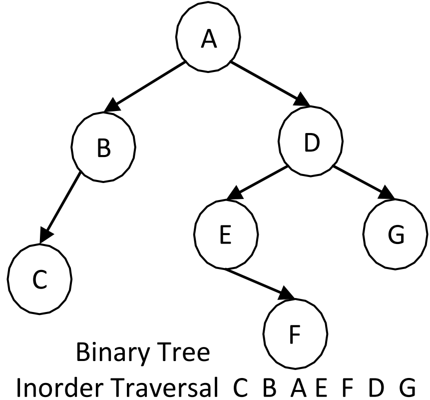
Advantages

- Inorder traversal is faster than unthreaded version as stack is not required.
- Effectively determines the predecessor and successor for inorder traversal, for unthreaded tree this task is more difficult.
- A stack is required to provide upward pointing information in tree which threading provides.
- It is possible to generate successor or predecessor of any node without having over head of stack with the help of threading.

Disadvantages

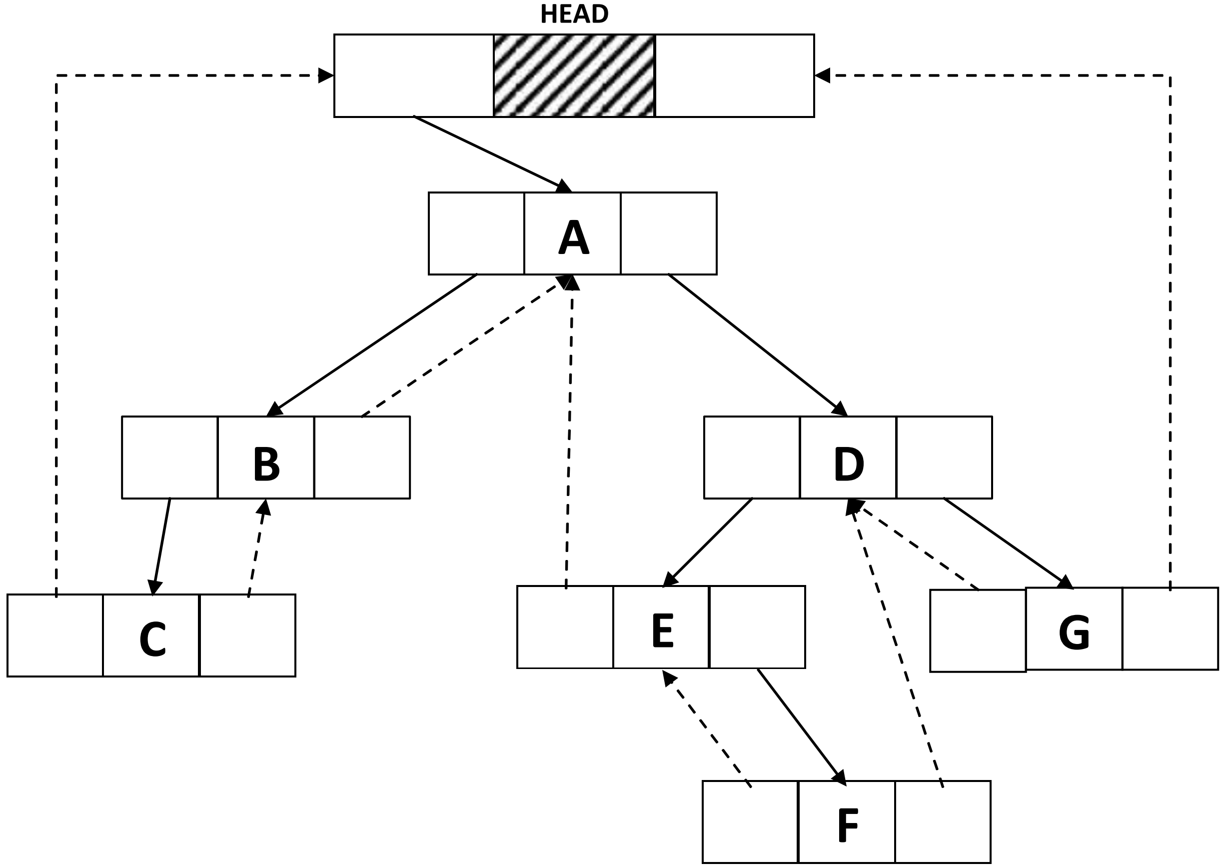
- Threaded trees are unable to share common subtrees

- If –ve addressing is not permitted in programming language, two additional fields are required.
- Insertion into and deletion from threaded binary tree are more time consuming because both thread and structural link must be maintained.



Binary Tree
Inorder Traversal C B A E F D G

Fully In-threaded binary tree of given binary tree



10. Draw a right in threaded binary tree for the given tree

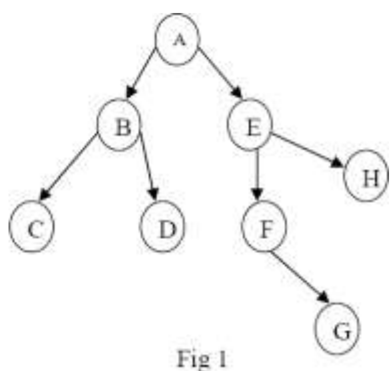
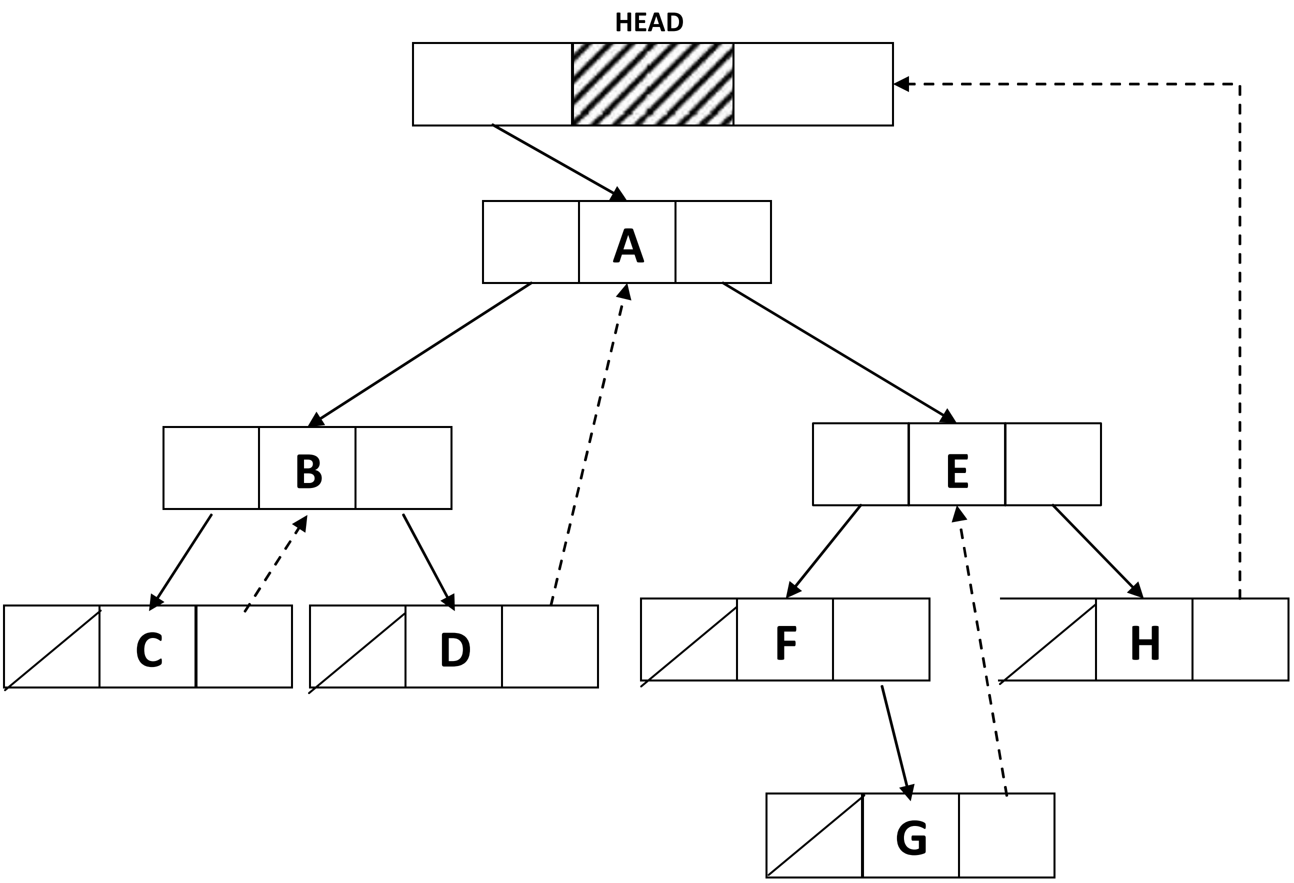


Fig 1

Right In-threaded binary tree of given binary tree



11. What is the meaning of height balanced tree? How rebalancing is done in height balanced tree.

A tree is called AVL (height balance binary tree), if each node possesses one of the following properties

1. A node is called left heavy if the longest path in its left sub tree is one longer then the longest path of its right sub tree.
2. A node is called right heavy if the longest path in the right sub tree is one longer than path in its left sub tree.
3. A node is called balanced, if the longest path in both the right and left sub tree are equal.

If tree becomes unbalanced by inserting any node, then based on position of insertion, we need to rotate the unbalanced node. Rotation is the process to make tree balanced

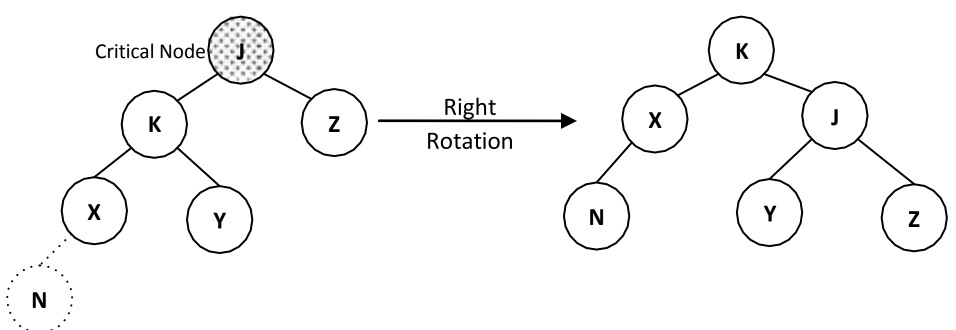
1) Insertion into Left sub-tree of nodes Left child – Single Right Rotation
2) Insertion into Right sub-tree of node's Left child – Left Right Rotation
3) Insertion into Left sub-tree of node's Right child – Right Left Rotation
4) Insertion into Right sub-tree of node's Right child – Single Left Rotation

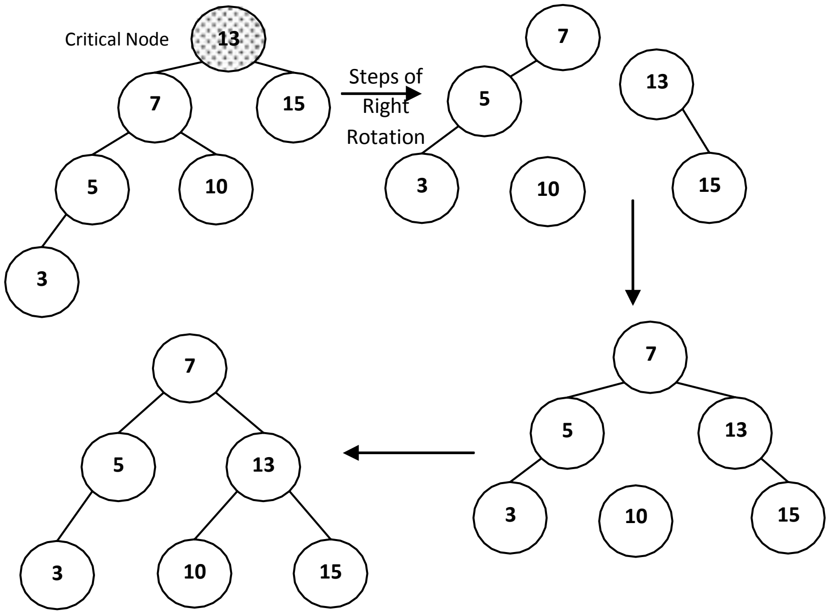
1) Insertion into Left sub-tree of nodes Left child – Single Right Rotation

If node becomes unbalanced after insertion of new node at Left sub-tree of nodes Left child, then we need to perform Single Right Rotation for unbalanced node.

Right Rotation

a. Detach leaf child's right sub-tree
b. Consider leaf child to be the new parent
c. Attach old parent onto right of new parent
d. Attach old leaf child's old right sub-tree as leaf sub-tree of new right child

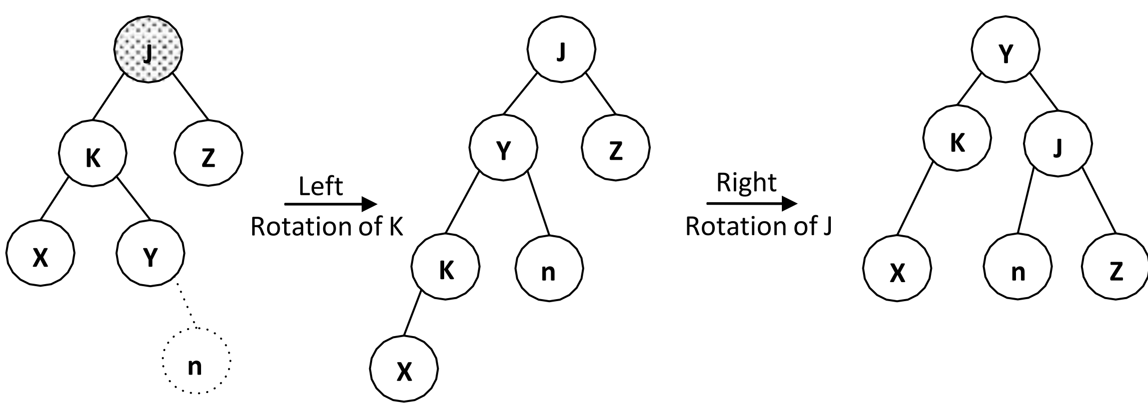




2) Insertion into Right sub-tree of node's Left child – Left Right Rotation

If node becomes unbalanced after insertion of new node at Right sub-tree of node's Left child, then we need to perform Left Right Rotation for unbalanced node.

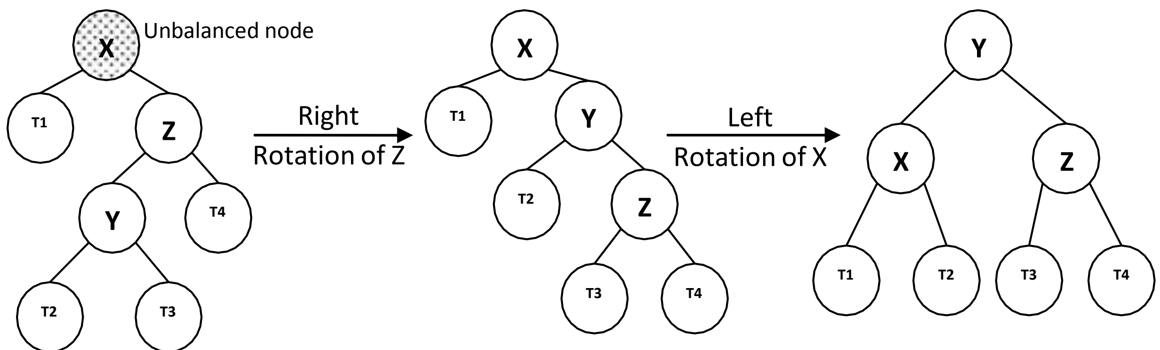
Leaf rotation of leaf child followed by right rotation of parent



3) Insertion into Left sub-tree of node's Right child – Right Left Rotation

If node becomes unbalanced after insertion of new node at Left sub-tree of node's Right child, then we need to perform Right Left Rotation for unbalanced node.

Single right rotation of right child followed by left rotation of parent

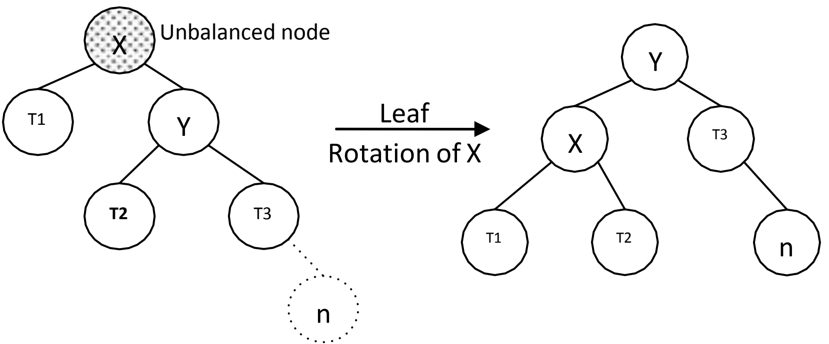


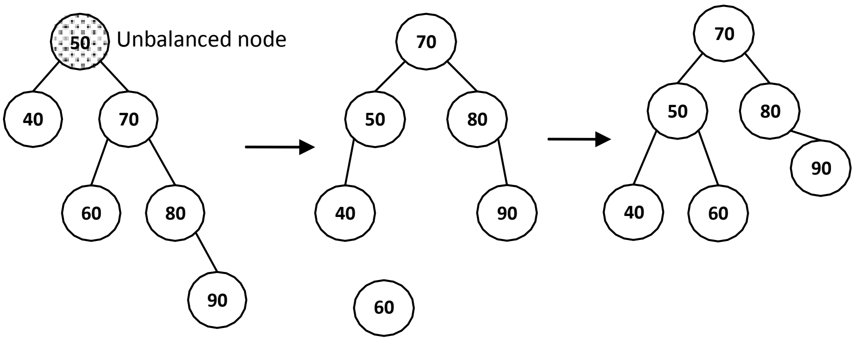
4) ***Insertion into Right sub-tree of node's Right child – Single Left Rotation***

If node becomes unbalanced after insertion of new node at Right sub-tree of nodes Right child, then we need to perform Single Left Rotation for unbalanced node.

Left Rotation

- **a.** Detach right child's leaf sub-tree
- **b.** Consider right child to be new parent
- **c.** Attach old parent onto left of new parent
- **d.** Attach old right child's old left sub-tree as right sub-tree of new left child

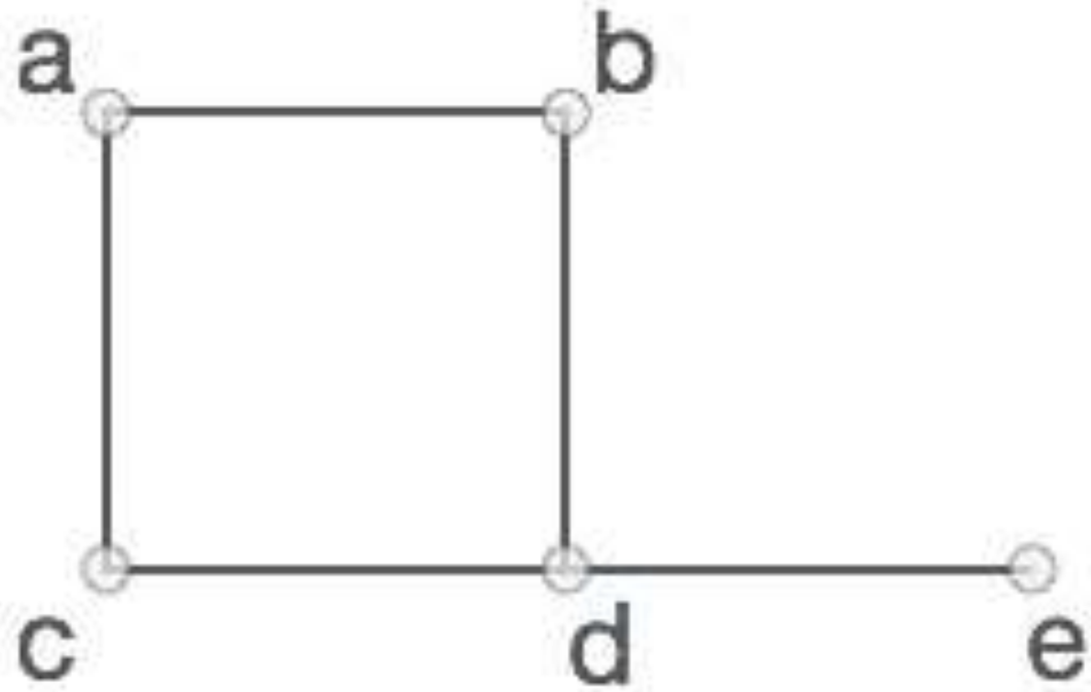




27. Graphs

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices,** and the links that connect the vertices are called **edges**.

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}

Graph Data Structure

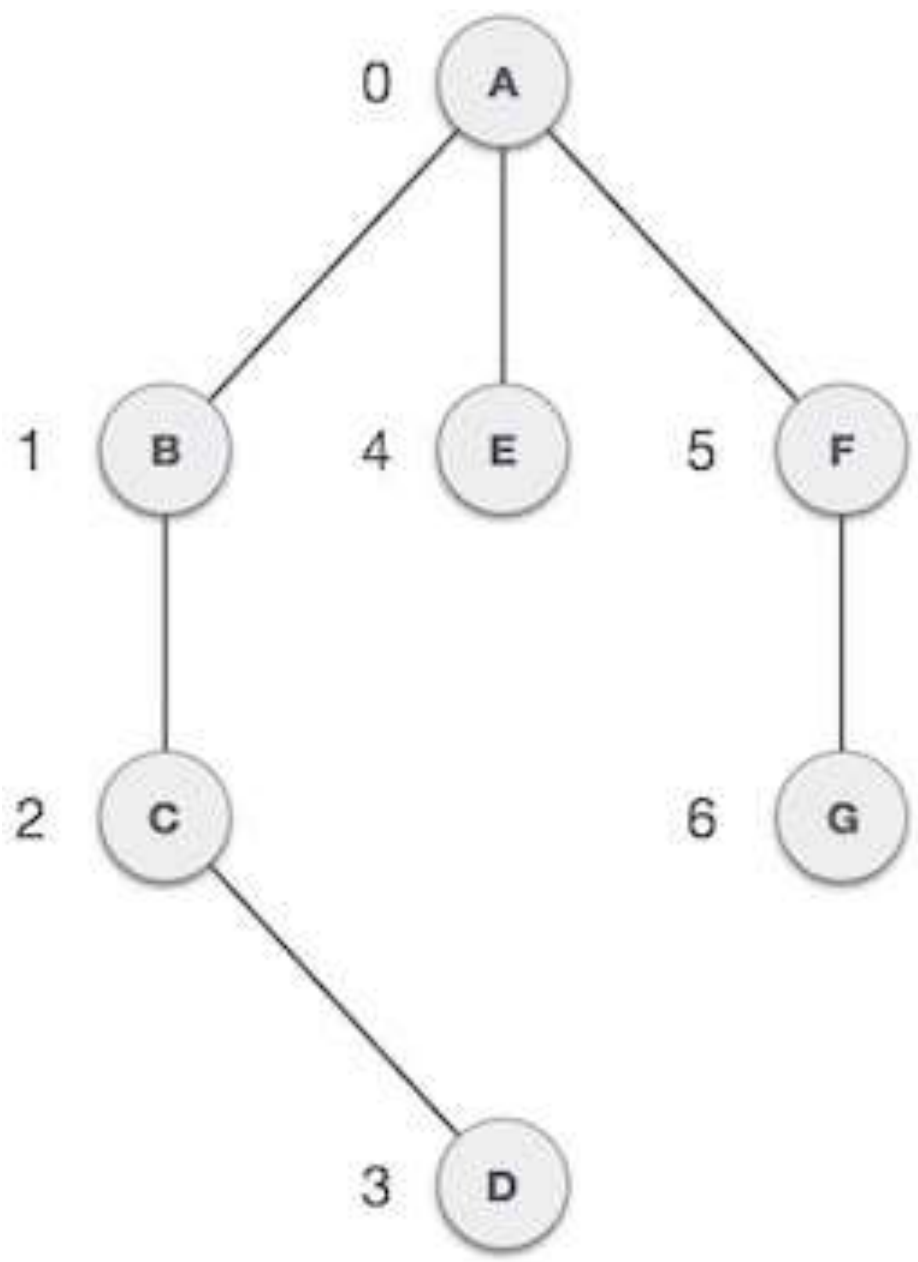
Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms −

- **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 represents 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

Following are the basic primary operations that can be performed on a Graph:

- **Add Vertex** – Adds a vertex to the graph.

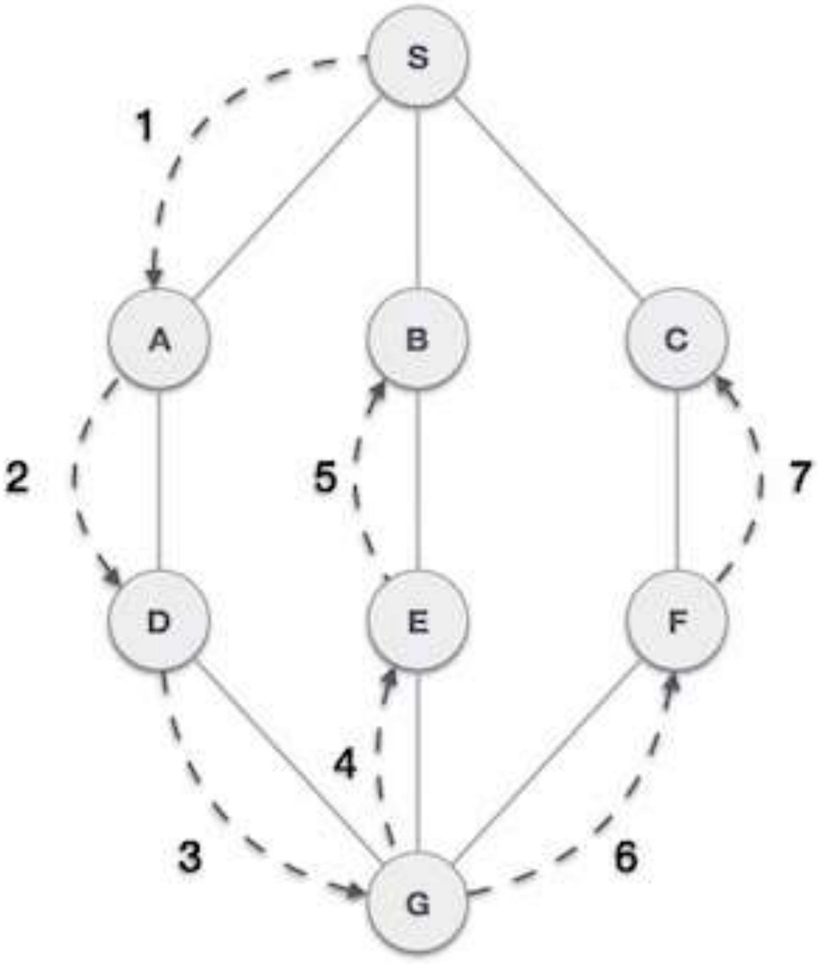
- **Add Edge** – Adds an edge between the two vertices of the graph.

- **Display Vertex** – Displays a vertex of the graph.

To know more about Graph, please read Graph Theory Tutorial. We shall learn about traversing a graph in the coming chapters.

28. Depth First Traversal

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack 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, DFS algorithm traverses from A to B to C to D first then to E, then to F and lastly to G. 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.

Steps	Traversal	Description
1.		Initialize the stack.
2.		Mark **S** as 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.		Mark **A** as 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.

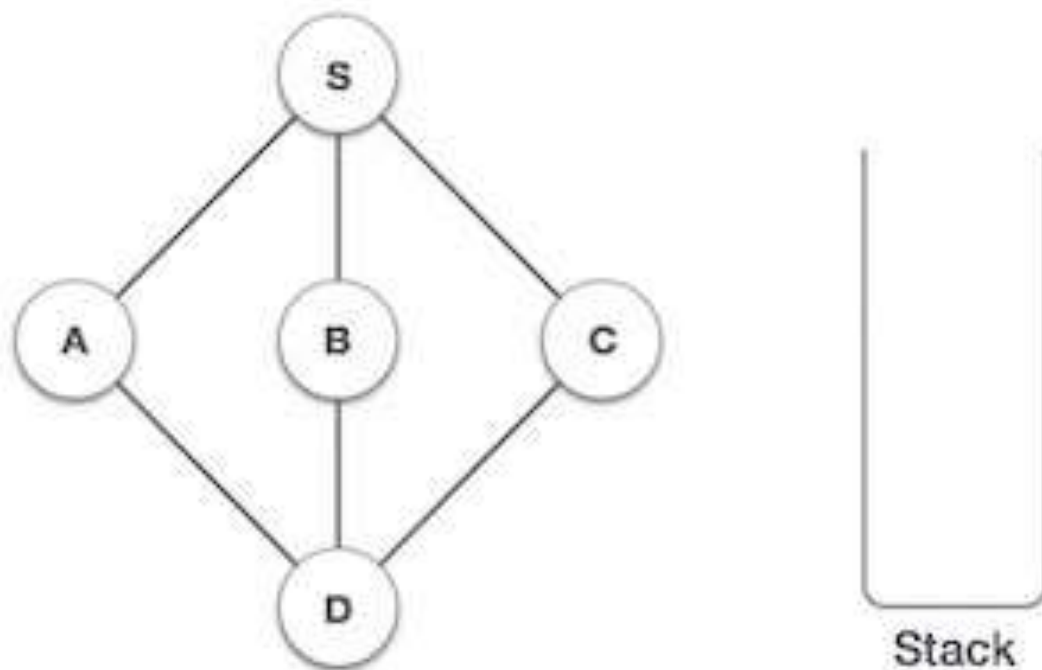
4.		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.
5.		We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.
6.		We check the stack top for 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.
7.		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so 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.

To know about the implementation of this algorithm in C programming language, click here.

Depth First Traversal in C

We shall not see the implementation of Depth First Traversal (or Depth First Search) in C programming language. For our reference purpose, we shall follow our example and take this as our graph model –



Implementation in C

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAX 5

struct Vertex {
   char label;
   bool visited;
};
```

```
//stack variables

int stack[MAX];
int top = -1;



//graph variables

//array of vertices
struct Vertex* lstVertices[MAX];

//adjacency matrix
int adjMatrix[MAX][MAX];

//vertex count
int vertexCount = 0;

//stack functions

void push(int item) {
   stack[++top] = item;
}

int pop() {
   return stack[top--];
}

int peek() {
   return stack[top];
}

bool isStackEmpty() {
   return top == -1;
}
```

```
//graph functions

//add vertex to the vertex list
void addVertex(char label) {
   struct Vertex* vertex = (struct Vertex*) malloc(sizeof(struct Vertex));
   vertex->label = label;
   vertex->visited = false;
   lstVertices[vertexCount++] = vertex;
}

//add edge to edge array
void addEdge(int start,int end) {
   adjMatrix[start][end] = 1;
   adjMatrix[end][start] = 1;
}

//display the vertex
void displayVertex(int vertexIndex) {
   printf("%c ",lstVertices[vertexIndex]->label);
}

//get the adjacent unvisited vertex
int getAdjUnvisitedVertex(int vertexIndex) {
   int i;

   for(i = 0; i<vertexCount; i++) {
      if(adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited == false) {
         return i;
      }
   }

   return -1;
}
```

```
void depthFirstSearch() {
   int i;

   //mark first node as visited
   lstVertices[0]->visited = true;

   //display the vertex
   displayVertex(0);

   //push vertex index in stack
   push(0);

   while(!isStackEmpty()) {
      //get the unvisited vertex of vertex which is at top of the stack
      int unvisitedVertex = getAdjUnvisitedVertex(peek());

      //no adjacent vertex found
      if(unvisitedVertex == -1) {
         pop();
      }else {
         lstVertices[unvisitedVertex]->visited = true;
         displayVertex(unvisitedVertex);
         push(unvisitedVertex);
      }
   }

   //stack is empty, search is complete, reset the visited flag
   for(i = 0;i < vertexCount;i++) {
      lstVertices[i]->visited = false;
   }
}
```

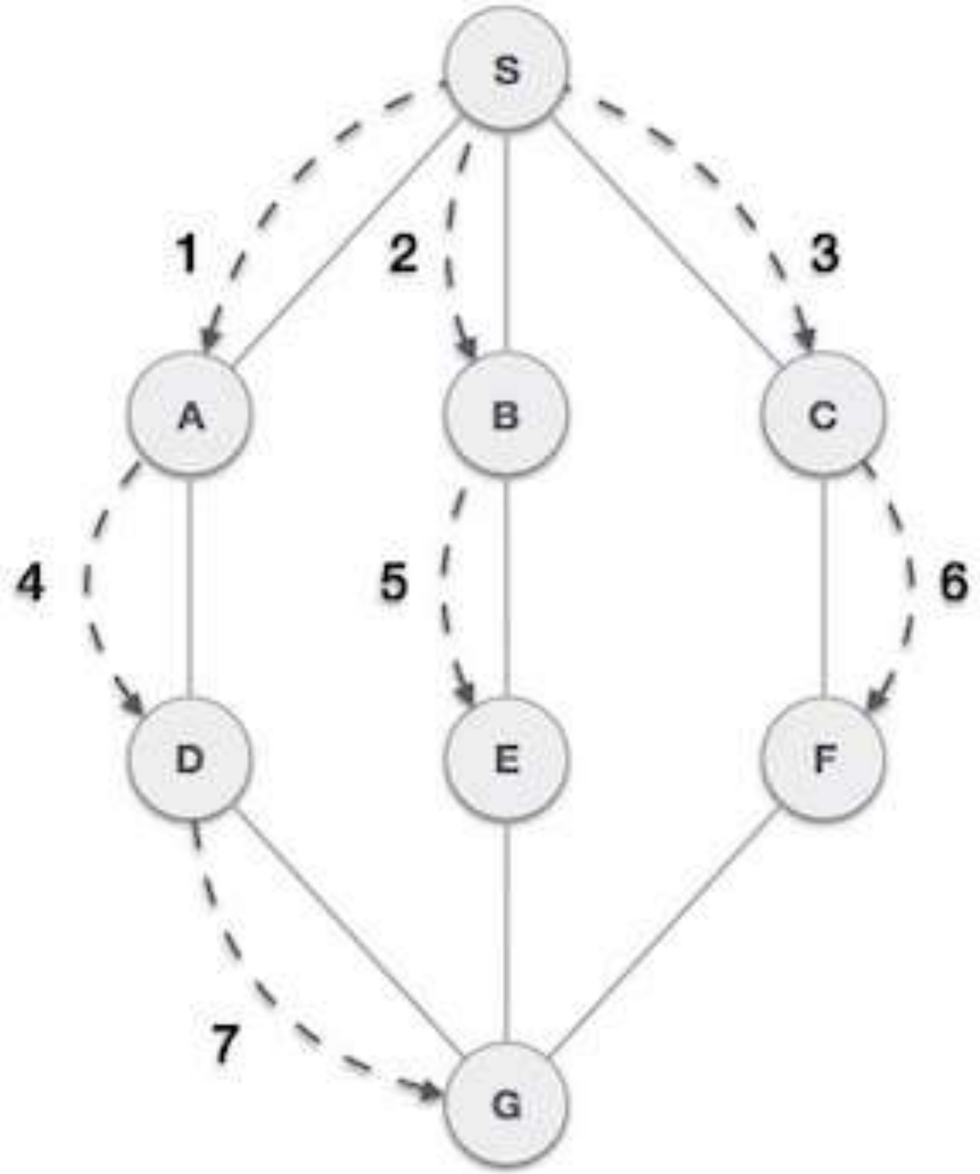
```
int main() {
   int i, j;

   for(i = 0; i<MAX; i++) // set adjacency {
      for(j = 0; j<MAX; j++) // matrix to 0
         adjMatrix[i][j] = 0;
   }

   addVertex('S');   // 0
   addVertex('A');   // 1
   addVertex('B');   // 2
   addVertex('C');   // 3
   addVertex('D');   // 4

   addEdge(0, 1);    // S - A
   addEdge(0, 2);    // S - B
   addEdge(0, 3);    // S - C
   addEdge(1, 4);    // A - D
   addEdge(2, 4);    // B - D
   addEdge(3, 4);    // C - D

   printf("Depth First Search: ");

   depthFirstSearch();

   return 0;
}
```

If we compile and run the above program, it will produce the following result –

```
Depth First Search: S A D B C
```

29. Breadth 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.

Steps	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.

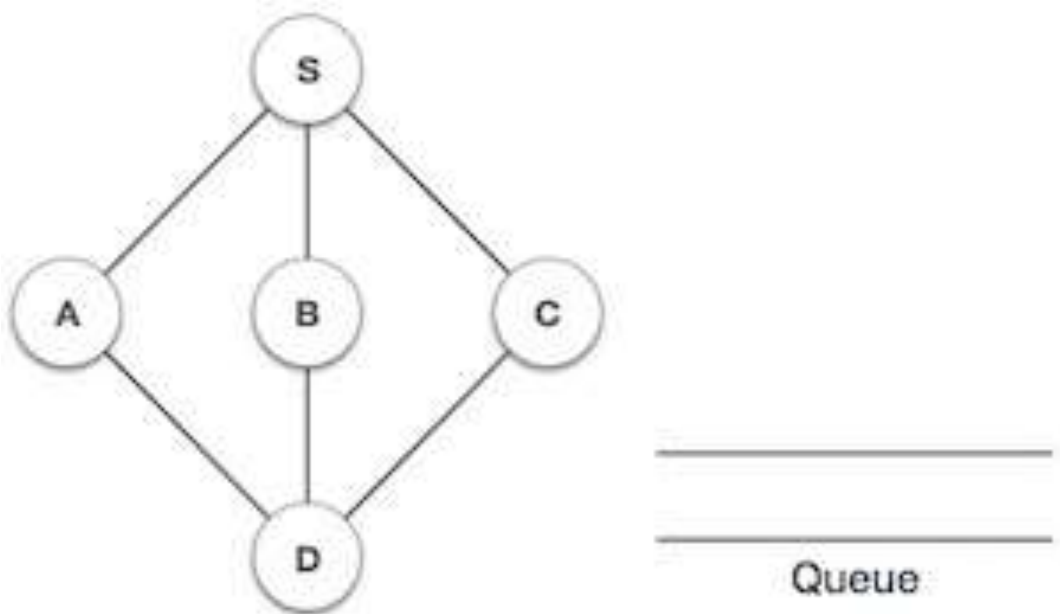
5.	S, A, B, C, D — Queue: C B A	Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.
6.	S, A, B, C, D — Queue: C B	Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.
7.	S, A, B, C, D — Queue: D C B	From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

The implementation of this algorithm in C programming language can be seen here.

Breadth First Traversal in C

We shall not see the implementation of Breadth First Traversal (or Breadth First Search) in C programming language. For our reference purpose, we shall follow our example and take this as our graph model –



Implementation in C

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAX 5

struct Vertex {
   char label;
   bool visited;
};

//queue variables

int queue[MAX];
int rear = -1;
int front = 0;
int queueItemCount = 0;

//graph variables
```

```
//array of vertices
struct Vertex* lstVertices[MAX];

//adjacency matrix
int adjMatrix[MAX][MAX];

//vertex count
int vertexCount = 0;



//queue functions

void insert(int data) {
   queue[++rear] = data;
   queueItemCount++;
}

int removeData() {
   queueItemCount--;
   return queue[front++];
}

bool isQueueEmpty() {
   return queueItemCount == 0;
}

//graph functions

//add vertex to the vertex list
void addVertex(char label) {
   struct Vertex* vertex = (struct Vertex*) malloc(sizeof(struct Vertex));
   vertex->label = label;
   vertex->visited = false;
   lstVertices[vertexCount++] = vertex;
}
```

```
//add edge to edge array
void addEdge(int start,int end) {
   adjMatrix[start][end] = 1;
   adjMatrix[end][start] = 1;
}

//display the vertex
void displayVertex(int vertexIndex) {
   printf("%c ",lstVertices[vertexIndex]->label);
}

//get the adjacent unvisited vertex
int getAdjUnvisitedVertex(int vertexIndex) {
   int i;

   for(i = 0; i<vertexCount; i++) {
      if(adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited == false)
         return i;
   }

   return -1;
}

void breadthFirstSearch() {
   int i;

   //mark first node as visited
   lstVertices[0]->visited = true;

   //display the vertex
   displayVertex(0);
```

```
   //insert vertex index in queue
   insert(0);
   int unvisitedVertex;

   while(!isQueueEmpty()) {
      //get the unvisited vertex of vertex which is at front of the queue
      int tempVertex = removeData();

      //no adjacent vertex found
      while((unvisitedVertex = getAdjUnvisitedVertex(tempVertex)) != -1) {
         lstVertices[unvisitedVertex]->visited = true;
         displayVertex(unvisitedVertex);
         insert(unvisitedVertex);
      }

   }

   //queue is empty, search is complete, reset the visited flag
   for(i = 0;i<vertexCount;i++) {
      lstVertices[i]->visited = false;
   }
}

int main() {
   int i, j;

   for(i = 0; i<MAX; i++) // set adjacency {
      for(j = 0; j<MAX; j++) // matrix to 0
         adjMatrix[i][j] = 0;
   }

   addVertex('S');   // 0
   addVertex('A');   // 1
   addVertex('B');   // 2
   addVertex('C');   // 3
   addVertex('D');   // 4
```

```
   addEdge(0, 1);    // S - A
   addEdge(0, 2);    // S - B
   addEdge(0, 3);    // S - C
   addEdge(1, 4);    // A - D
   addEdge(2, 4);    // B - D
   addEdge(3, 4);    // C - D


   printf("\nBreadth First Search: ");

   breadthFirstSearch();

   return 0;
}
```

If we compile and run the above program, it will produce the following result –

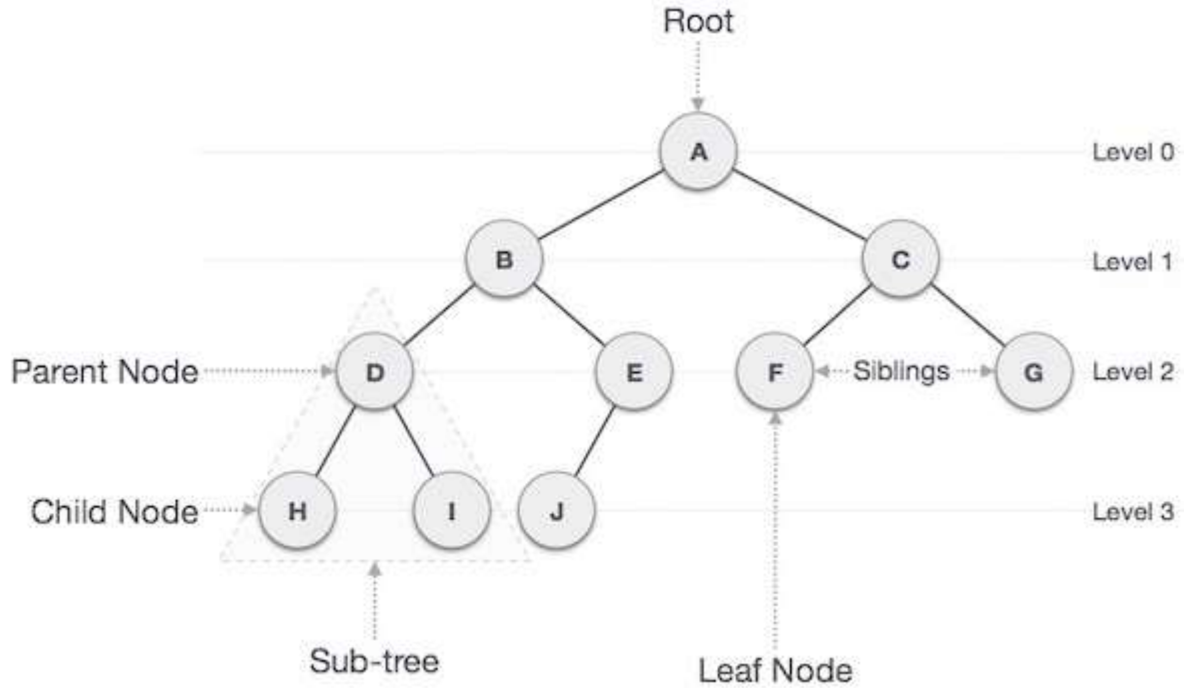
```
Breadth First Search: S A B C D
```

Tree Data Structure

30. Tree

Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically.

Binary Tree is a special datastructure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.



Important Terms

Following are the important terms with respect to tree.

- **Path** – Path refers to the sequence of nodes along the edges of a tree.
- **Root** – The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** – Any node except the root node has one edge upward to a node called parent.
- **Child** – The node below a given node connected by its edge downward is called its child node.
- **Leaf** – The node which does not have any child node is called the leaf node.
- **Subtree** – Subtree represents the descendants of a node.

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

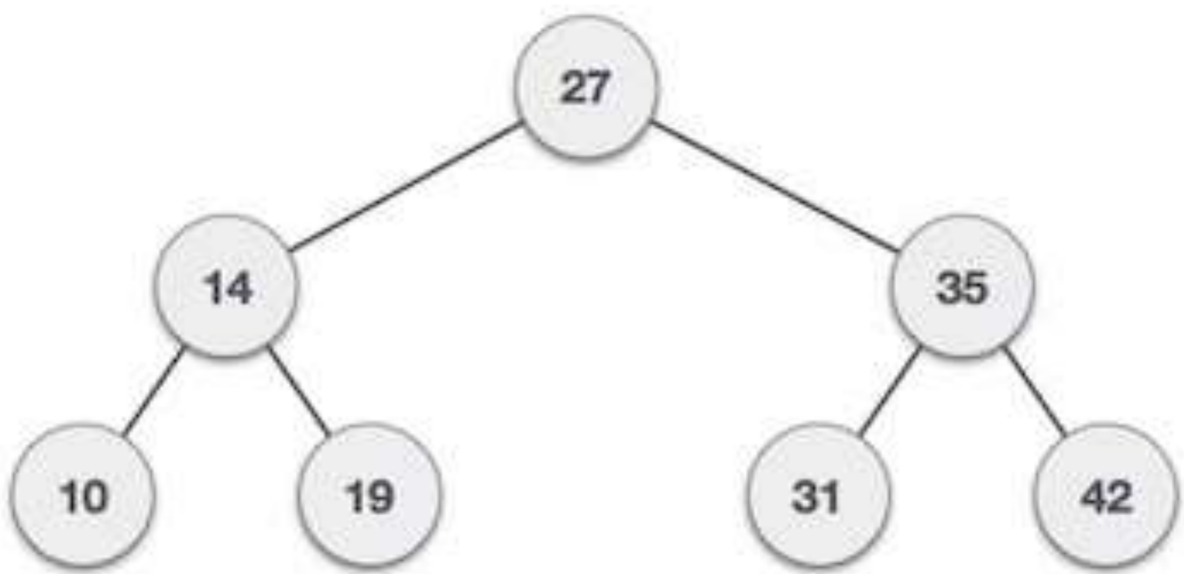
- **Traversing** – Traversing means passing through nodes in a specific order.

- **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.

- **Keys** – Key represents a value of a node based on which a search operation is to be carried out for a node.

Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.



We're going to implement tree using node object and connecting them through references.

Tree Node

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

In a tree, all nodes share common construct.

BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following –

- **Insert** – Inserts an element in a tree/create a tree.
- **Search** – Searches an element in a tree.
- **Pre-order Traversal** – Traverses a tree in a pre-order manner.
- **In-order Traversal** – Traverses a tree in an in-order manner.
- **Post-order Traversal** – Traverses a tree in a post-order manner.

We shall learn creating (inserting into) a tree structure and searching a data item in a tree in this chapter. We shall learn about tree traversing methods in the coming chapter.

Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm

```
If root is NULL
   then create root node
return

If root exists then
   compare the data with node.data

   while until insertion position is located

      If data is greater than node.data
         goto right subtree
      else
         goto left subtree

   endwhile
```

```
   insert data

end If
```

Implementation

The implementation of insert function should look like this –

```
void insert(int data) {
   struct node *tempNode = (struct node*) malloc(sizeof(struct node));
   struct node *current;
   struct node *parent;

   tempNode->data = data;
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;

   //if tree is empty, create root node
   if(root == NULL) {
      root = tempNode;
   }else {
      current = root;
      parent  = NULL;

      while(1) {
         parent = current;

         //go to left of the tree
         if(data < parent->data) {
            current = current->leftChild;

            //insert to the left
            if(current == NULL) {
               parent->leftChild = tempNode;
               return;
            }
         }
```

```
         //go to right of the tree
         else {
            current = current->rightChild;

            //insert to the right
            if(current == NULL) {
               parent->rightChild = tempNode;
               return;
            }
         }
      }
   }
}
```

Search Operation

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```
If root.data is equal to search.data
   return root
else
   while data not found

      If data is greater than node.data
         goto right subtree
      else
         goto left subtree

      If data found
         return node

   endwhile

   return data not found
```

```
end if
```

The implementation of this algorithm should look like this.

```
struct node* search(int data) {
   struct node *current = root;
   printf("Visiting elements: ");

   while(current->data != data) {
      if(current != NULL)
      printf("%d ",current->data);
     //go to left tree

      if(current->data > data) {
         current = current->leftChild;
      }
      //else go to right tree
      else {
         current = current->rightChild;
      }

      //not found
      if(current == NULL) {
         return NULL;
      }

      return current;
   }
}
```

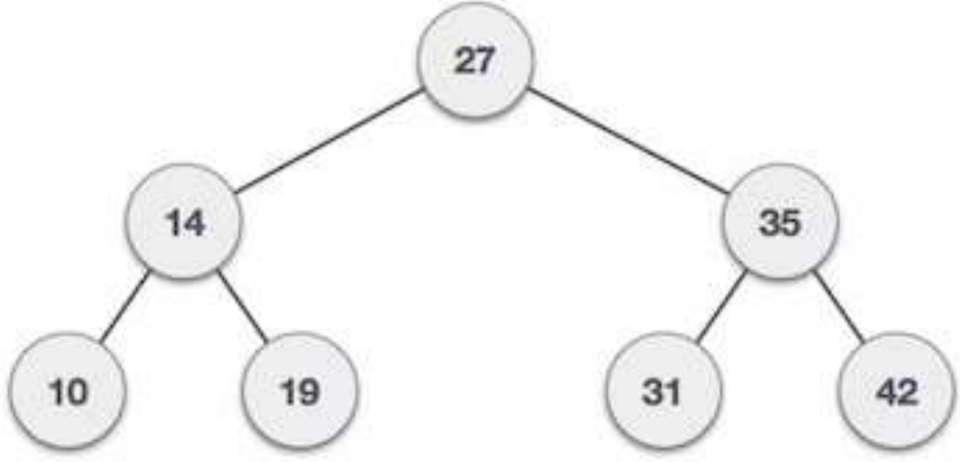
To know about the implementation of binary search tree data structure, please click here.

Tree Traversal in C

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot random access a node in a tree. There are three ways which we use to traverse a tree –

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

We shall now look at the implementation of tree traversal in C programming language here using the following binary tree –



Implementation in C

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;

   struct node *leftChild;
   struct node *rightChild;
};

struct node *root = NULL;

void insert(int data) {
   struct node *tempNode = (struct node*) malloc(sizeof(struct node));
   struct node *current;
   struct node *parent;
   tempNode->data = data;
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;
```

```
    //if tree is empty
    if(root == NULL) {
       root = tempNode;
    }else {
       current = root;
       parent = NULL;

       while(1) {
          parent = current;

          //go to left of the tree
          if(data < parent->data) {
             current = current->leftChild;

             //insert to the left
             if(current == NULL) {
                parent->leftChild = tempNode;
                return;
             }
          }//go to right of the tree
          else {
             current = current->rightChild;

             //insert to the right
             if(current == NULL) {
                parent->rightChild = tempNode;
                return;
             }
          }
       }
    }
 }
 struct node* search(int data) {
    struct node *current = root;
    printf("Visiting elements: ");
```

```
   while(current->data != data) {
      if(current != NULL)
         printf("%d ",current->data);

      //go to left tree
      if(current->data > data) {
         current = current->leftChild;
      }
      //else go to right tree
      else {
         current = current->rightChild;
      }

      //not found
      if(current == NULL) {
         return NULL;
      }
   }

   return current;
}

void pre_order_traversal(struct node* root) {
   if(root != NULL) {
      printf("%d ",root->data);
      pre_order_traversal(root->leftChild);
      pre_order_traversal(root->rightChild);
   }
}

void inorder_traversal(struct node* root) {
   if(root != NULL) {
      inorder_traversal(root->leftChild);
      printf("%d ",root->data);
      inorder_traversal(root->rightChild);
   }
}
```

```
void post_order_traversal(struct node* root) {
   if(root != NULL) {
      post_order_traversal(root->leftChild);
      post_order_traversal(root->rightChild);
      printf("%d ", root->data);
   }
}

int main() {
   int i;
   int array[7] = { 27, 14, 35, 10, 19, 31, 42 };

   for(i = 0; i < 7; i++)
      insert(array[i]);
   i = 31;
   struct node * temp = search(i);

   if(temp != NULL) {
      printf("[%d] Element found.", temp->data);
      printf("\n");
   }else {
      printf("[ x ] Element not found (%d).\n", i);
   }

   i = 15;
   temp = search(i);


   if(temp != NULL) {
      printf("[%d] Element found.", temp->data);
      printf("\n");
   }else {
      printf("[ x ] Element not found (%d).\n", i);
   }

   printf("\nPreorder traversal: ");
```

```
   pre_order_traversal(root);

   printf("\nInorder traversal: ");
   inorder_traversal(root);

   printf("\nPost order traversal: ");
   post_order_traversal(root);

   return 0;
}
```

If we compile and run the above program, it will produce the following result –

```
Visiting elements: 27 35 [31] Element found.
Visiting elements: 27 14 19 [ x ] Element not found (15).

Preorder traversal: 27 14 10 19 35 31 42
Inorder traversal: 10 14 19 27 31 35 42
Post order traversal: 10 19 14 31 42 35 27
```

31. Tree Traversal

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree –

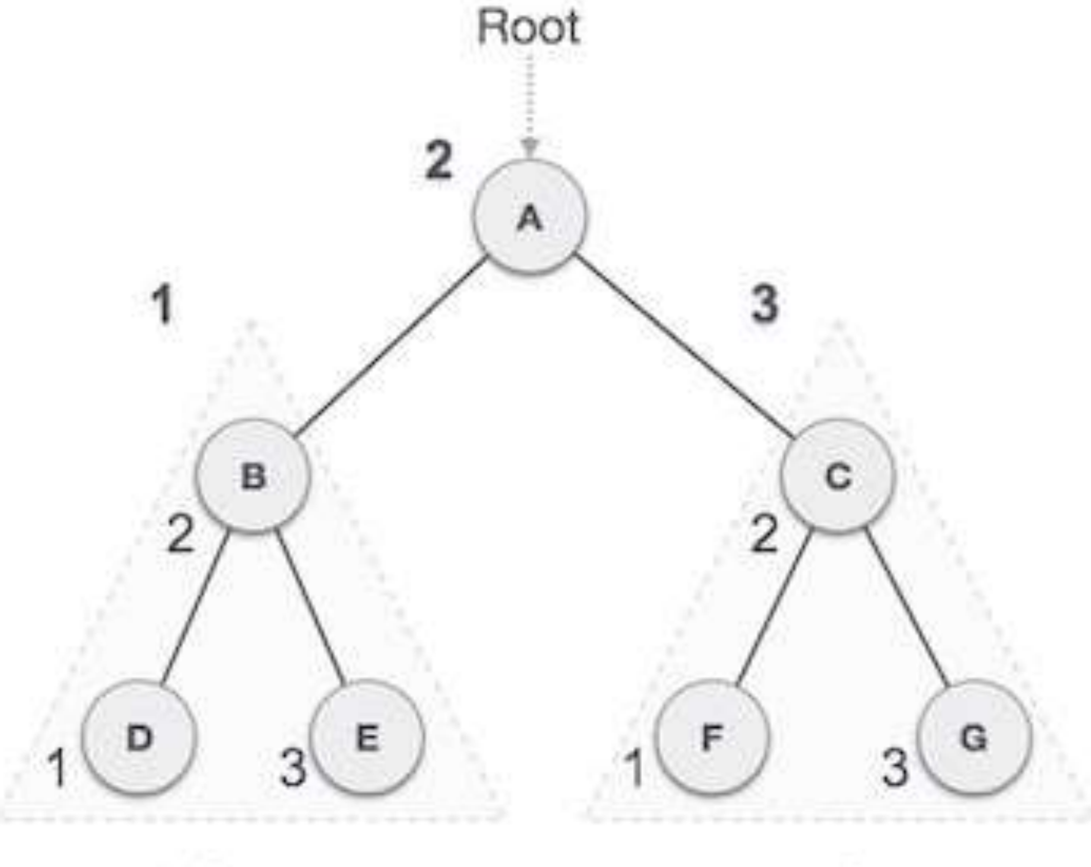
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of in-order traversal of this tree will be –

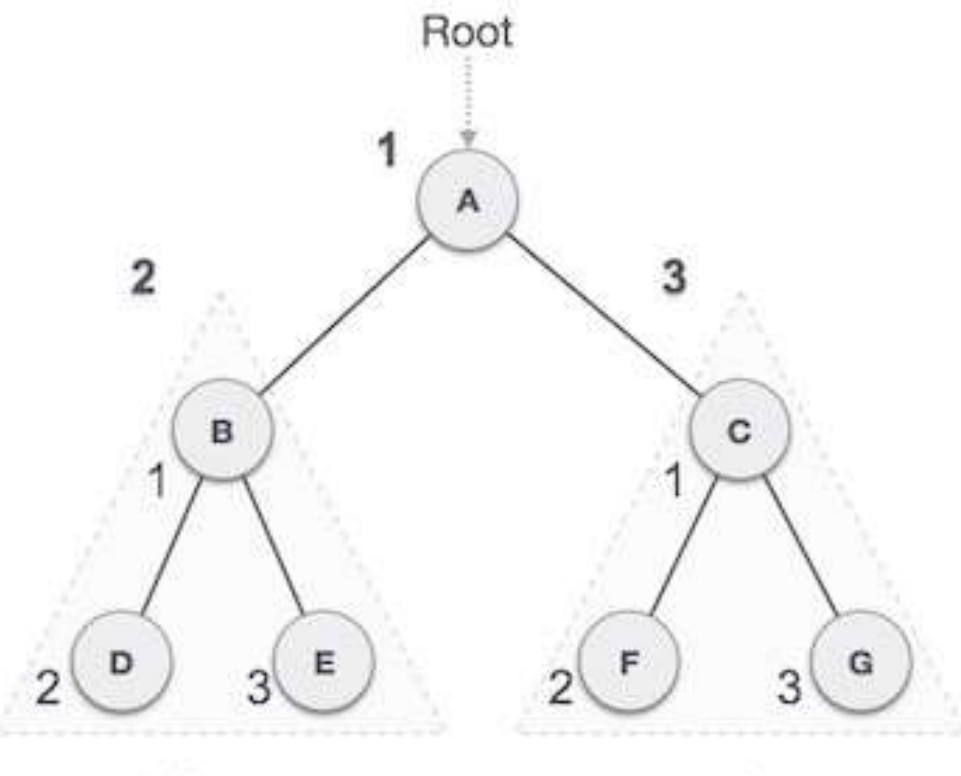
D → B → E → A → F → C → G

Algorithm

```
Until all nodes are traversed –
Step 1 – Recursively traverse left subtree.
Step 2 – Visit root node.
Step 3 – Recursively traverse right subtree.
```

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be –

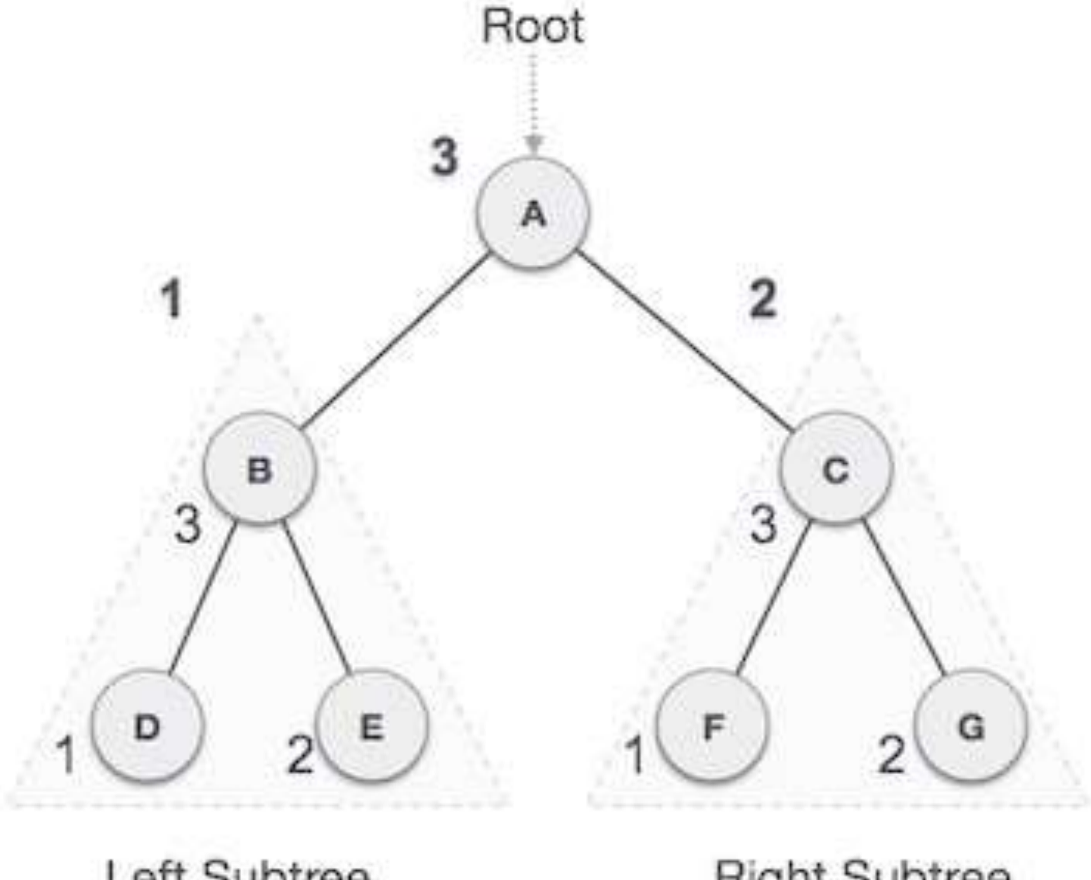
A → B → D → E → C → F → G

Algorithm

```
Until all nodes are traversed –
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.
```

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following pre-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be –

D → E → B → F → G → C → A

Algorithm

```
Until all nodes are traversed –
Step 1 – Recursively traverse left subtree.
Step 2 – Recursively traverse right subtree.
Step 3 – Visit root node.
```

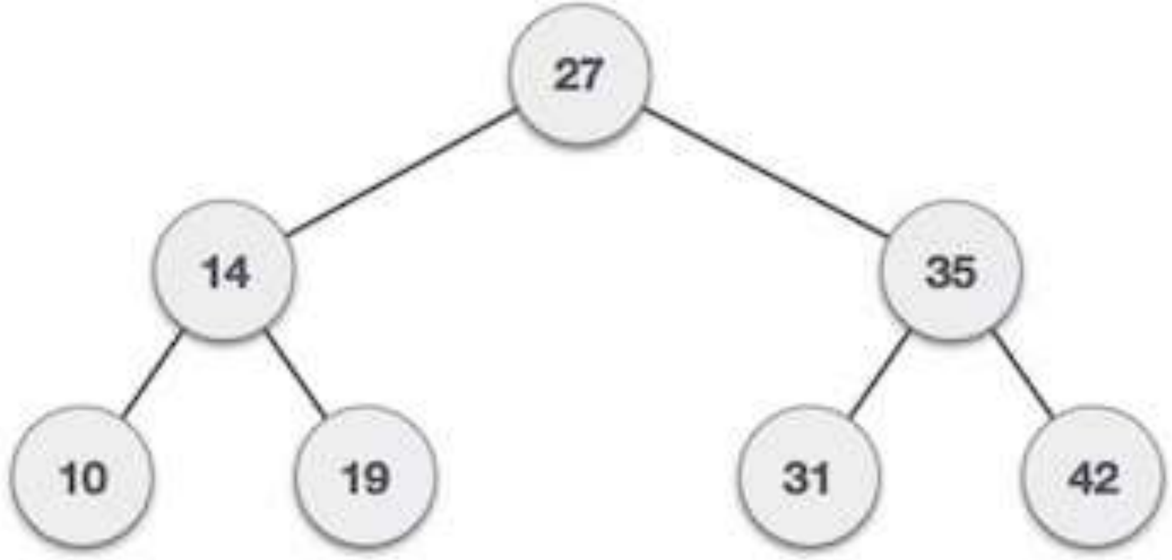
To check the C implementation of tree traversing, please click here

Tree Traversal in C

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree –

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

We shall now see the implementation of tree traversal in C programming language here using the following binary tree –



Implementation in C

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;

   struct node *leftChild;
   struct node *rightChild;
};

struct node *root = NULL;

void insert(int data) {
   struct node *tempNode = (struct node*) malloc(sizeof(struct node));
   struct node *current;
   struct node *parent;

   tempNode->data = data;
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;

   //if tree is empty
   if(root == NULL) {
      root = tempNode;
   }else {
      current = root;
      parent = NULL;

      while(1) {
         parent = current;

         //go to left of the tree
         if(data < parent->data) {
            current = current->leftChild;

            //insert to the left
```

```
            if(current == NULL) {
               parent->leftChild = tempNode;
               return;
            }
         }//go to right of the tree
         else {
            current = current->rightChild;

            //insert to the right
            if(current == NULL) {
               parent->rightChild = tempNode;
               return;
            }
         }
      }
   }
}

struct node* search(int data) {
   struct node *current = root;
   printf("Visiting elements: ");

   while(current->data != data) {
      if(current != NULL)
         printf("%d ",current->data);

      //go to left tree
      if(current->data > data) {
         current = current->leftChild;
      }
      //else go to right tree
      else {
         current = current->rightChild;
      }

      //not found
      if(current == NULL) {
```

```
         return NULL;
      }
   }
   return current;
}

void pre_order_traversal(struct node* root) {
   if(root != NULL) {
      printf("%d ",root->data);
      pre_order_traversal(root->leftChild);
      pre_order_traversal(root->rightChild);
   }
}

void inorder_traversal(struct node* root) {
   if(root != NULL) {
      inorder_traversal(root->leftChild);
      printf("%d ",root->data);
      inorder_traversal(root->rightChild);
   }
}

void post_order_traversal(struct node* root) {
   if(root != NULL) {
      post_order_traversal(root->leftChild);
      post_order_traversal(root->rightChild);
      printf("%d ", root->data);
   }
}







int main() {
   int i;
```

```
   int array[7] = { 27, 14, 35, 10, 19, 31, 42 };

   for(i = 0; i < 7; i++)
      insert(array[i]);
   i = 31;
   struct node * temp = search(i);

   if(temp != NULL) {
      printf("[%d] Element found.", temp->data);
      printf("\n");
   }else {
      printf("[ x ] Element not found (%d).\n", i);
   }

   i = 15;
   temp = search(i);

   if(temp != NULL) {
      printf("[%d] Element found.", temp->data);
      printf("\n");
   }else {
      printf("[ x ] Element not found (%d).\n", i);
   }

   printf("\nPreorder traversal: ");
   pre_order_traversal(root);

   printf("\nInorder traversal: ");
   inorder_traversal(root);

   printf("\nPost order traversal: ");
   post_order_traversal(root);
   return 0;
}
```

If we compile and run the above program, it will produce the following result –

```
Visiting elements: 27 35 [31] Element found.
Visiting elements: 27 14 19 [ x ] Element not found (15).

Preorder traversal: 27 14 10 19 35 31 42
Inorder traversal: 10 14 19 27 31 35 42
Post order traversal: 10 19 14 31 42 35 27
```

32. Binary Search Tree

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties –

- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than or equal to its parent node's key.

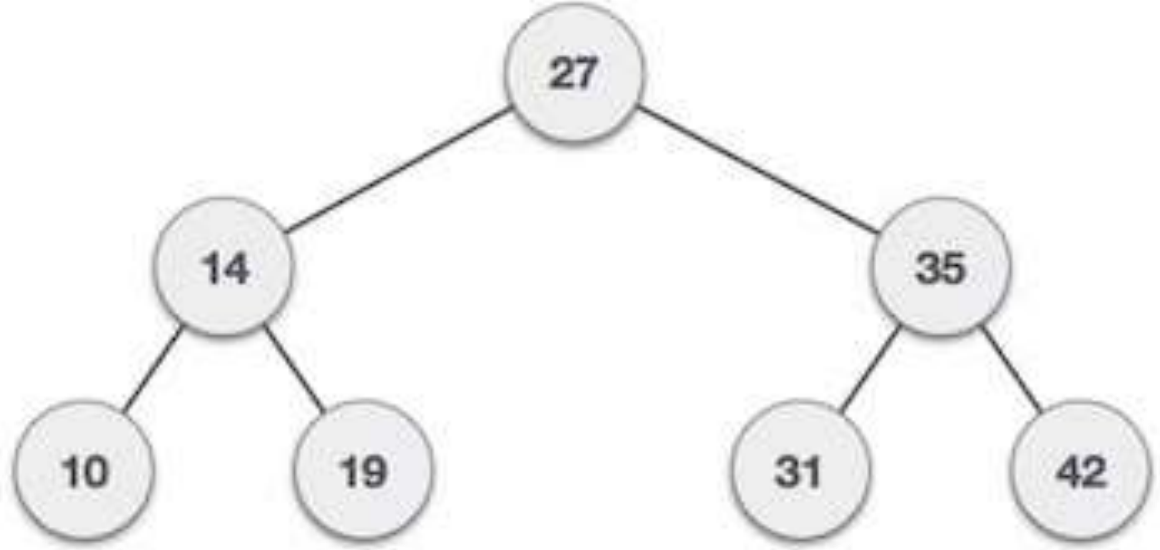
Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as –

```
left_subtree (keys)  ≤  node (key)  ≤  right_subtree (keys)
```

Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST –



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

Basic Operations

Following are the basic operations of a tree -

- **Search** – Searches an element in a tree.
- **Insert** – Inserts an element in a tree.
- **Pre-order Traversal** – Traverses a tree in a pre-order manner.
- **In-order Traversal** – Traverses a tree in an in-order manner.
- **Post-order Traversal** – Traverses a tree in a post-order manner.

Node

Define a node having some data, references to its left and right child nodes.

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

Search Operation

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

```
struct node* search(int data){
   struct node *current = root;
   printf("Visiting elements: ");

   while(current->data != data){

      if(current != NULL) {
         printf("%d ",current->data);
         //go to left tree
         if(current->data > data){
            current = current->leftChild;
```

```
         }//else go to right tree
         else {
            current = current->rightChild;
         }

         //not found
         if(current == NULL){
            return NULL;
         }
      }
   }
   return current;
}
```

Insert Operation

Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

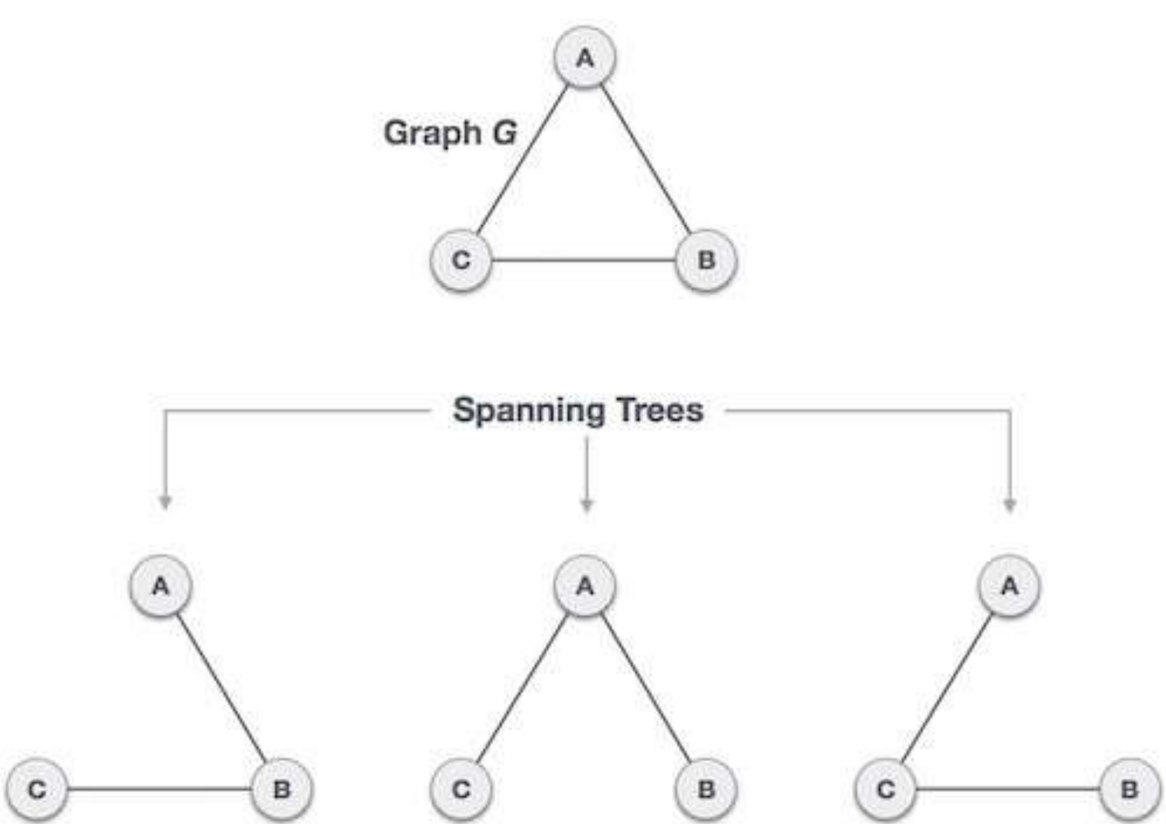
```
void insert(int data){
   struct node *tempNode = (struct node*) malloc(sizeof(struct node));
   struct node *current;
   struct node *parent;

   tempNode->data = data;
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;

   //if tree is empty
   if(root == NULL){
      root = tempNode;
   }else {
      current = root;
      parent = NULL;

      while(1){
         parent = current;
```

```
            //go to left of the tree
            if(data < parent->data){
               current = current->leftChild;
               //insert to the left

               if(current == NULL){
                  parent->leftChild = tempNode;
                  return;
               }
            }//go to right of the tree
            else{
               current = current->rightChild;
               //insert to the right
               if(current == NULL){
                  parent->rightChild = tempNode;
                  return;
               }
            }
         }
      }
   }
```

34. Spanning Tree

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees, where **n** is the number of nodes. In the above addressed example, **n is 3**, hence $\mathbf{3^{3-2} = 3}$ spanning trees are possible.

General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G -

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).

- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

Mathematical Properties of Spanning Tree

- Spanning tree has **n-1** edges, where **n** is the number of nodes (vertices).
- From a complete graph, by removing maximum **e-n+1** edges, we can construct a spanning tree.
- A complete graph can have maximum n^{n-2} number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are –

- **Civil Network Planning**
- **Computer Network Routing Protocol**
- **Cluster Analysis**

Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

Minimum Spanning-Tree Algorithm

We shall learn about two most important spanning tree algorithms here –

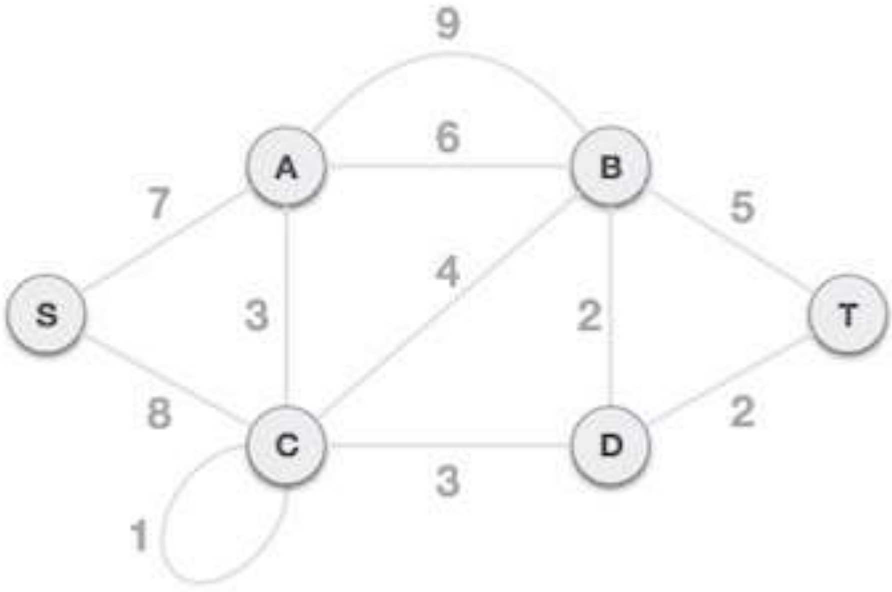
- Kruskal's Algorithm
- Prim's Algorithm

Both are greedy algorithms.

Kruskal's Spanning Tree Algorithm

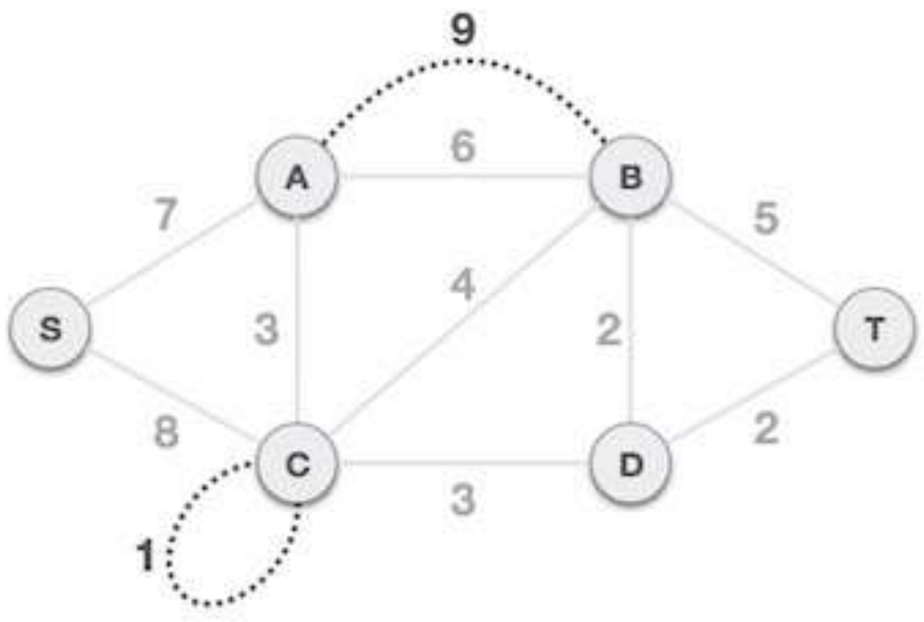
Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

To understand Kruskal's algorithm let us consider the following example –

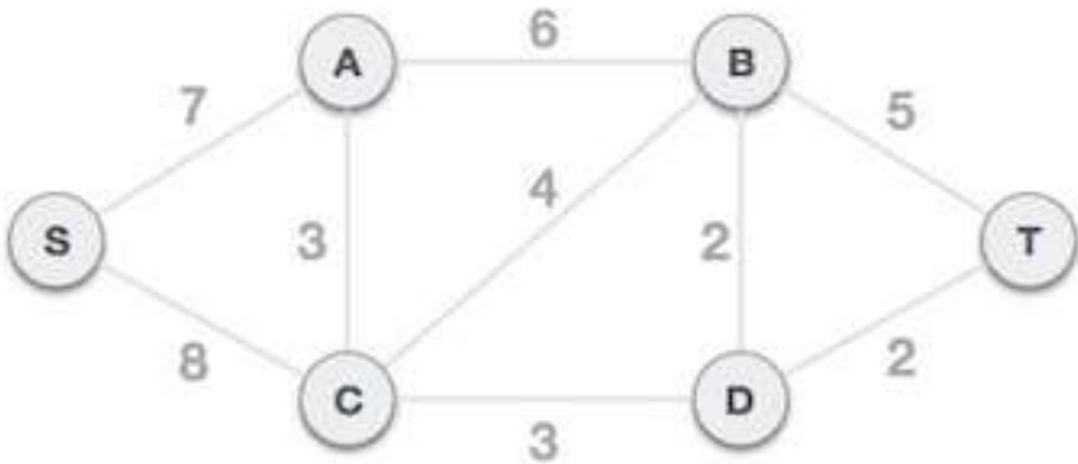


Step 1 - Remove all loops and parallel edges

Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.



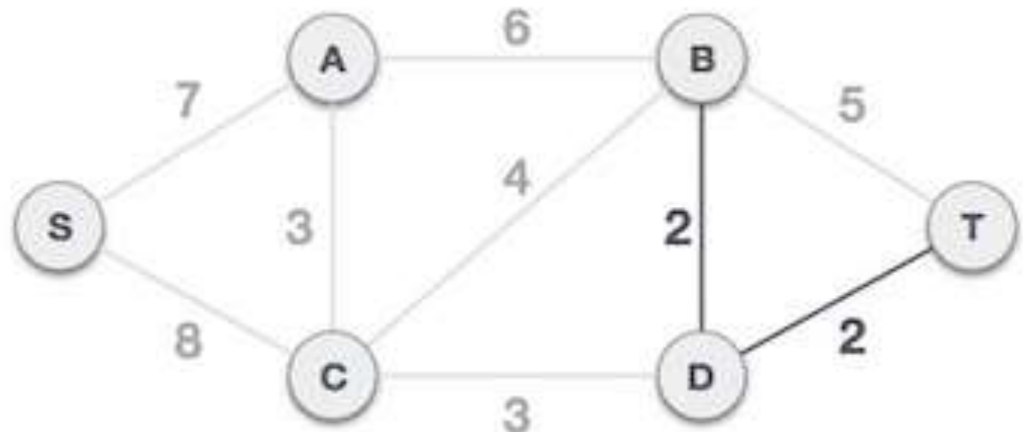
Step 2 - Arrange all edges in their increasing order of weight

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

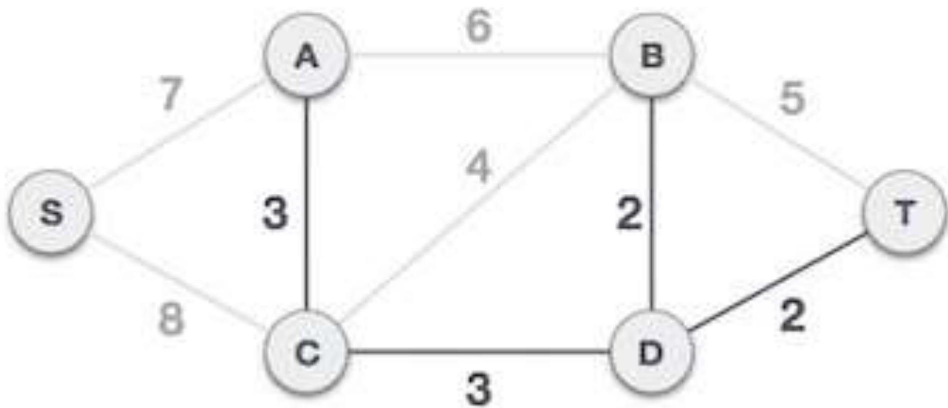
Step 3 - Add the edge which has the least weightage

Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

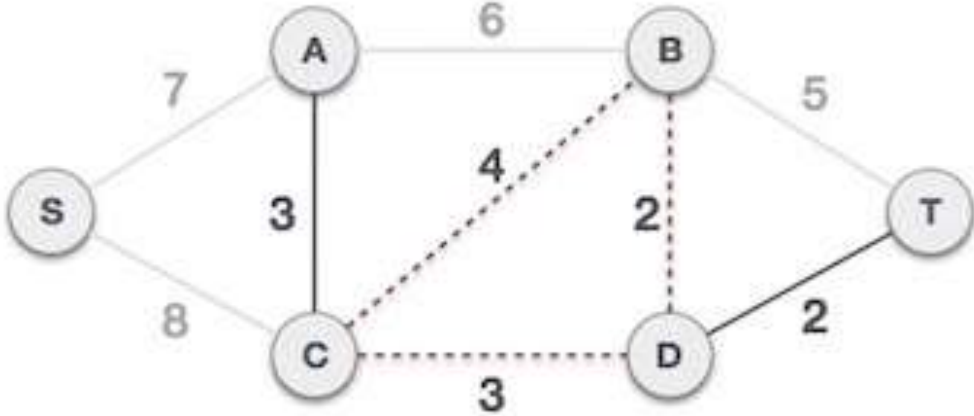


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

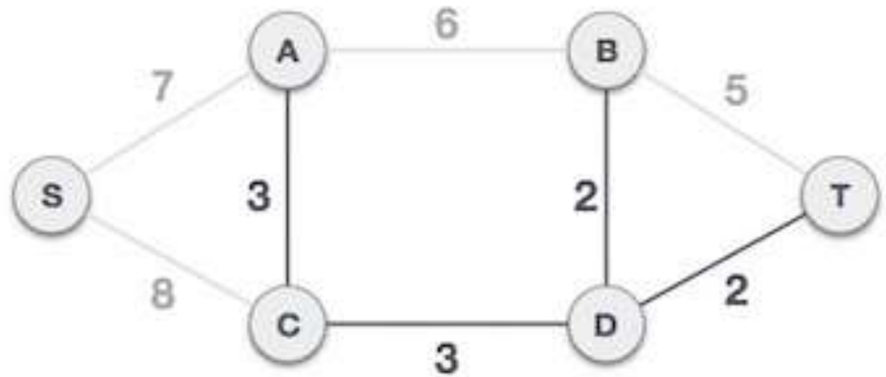
Next cost is 3, and associated edges are A,C and C,D. We add them again −



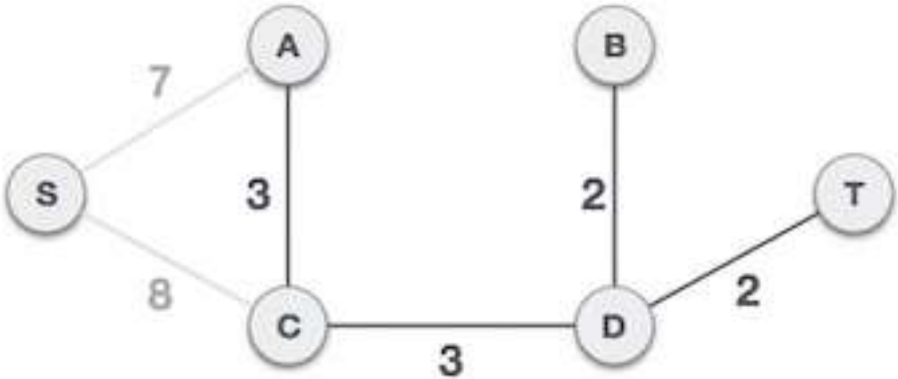
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph.



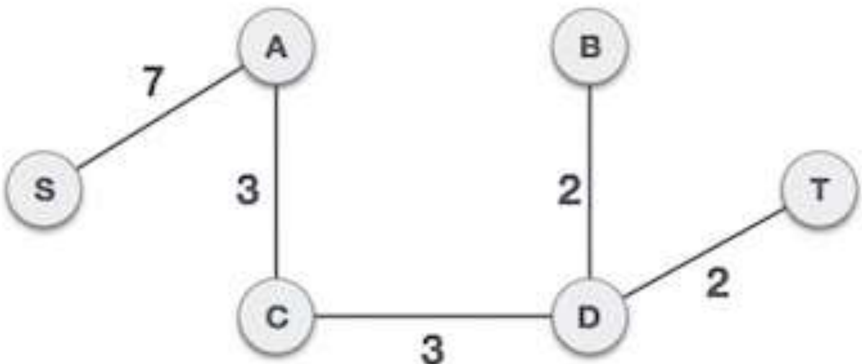
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



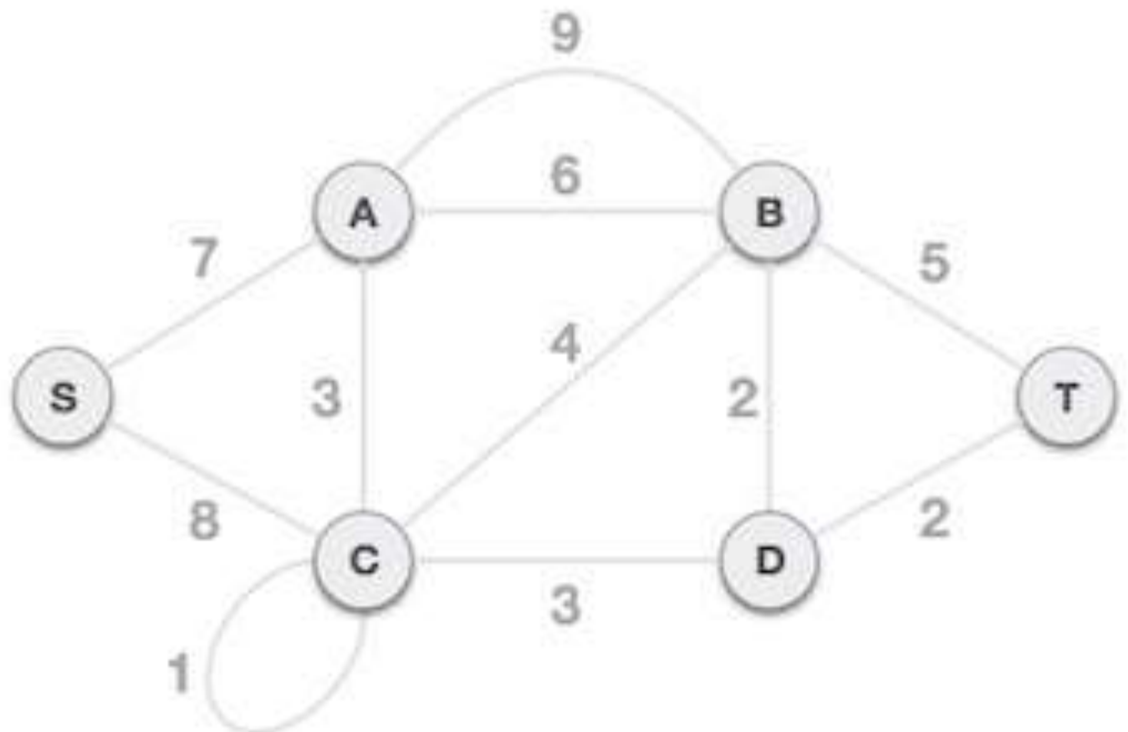
By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

Prim's Spanning Tree Algorithm

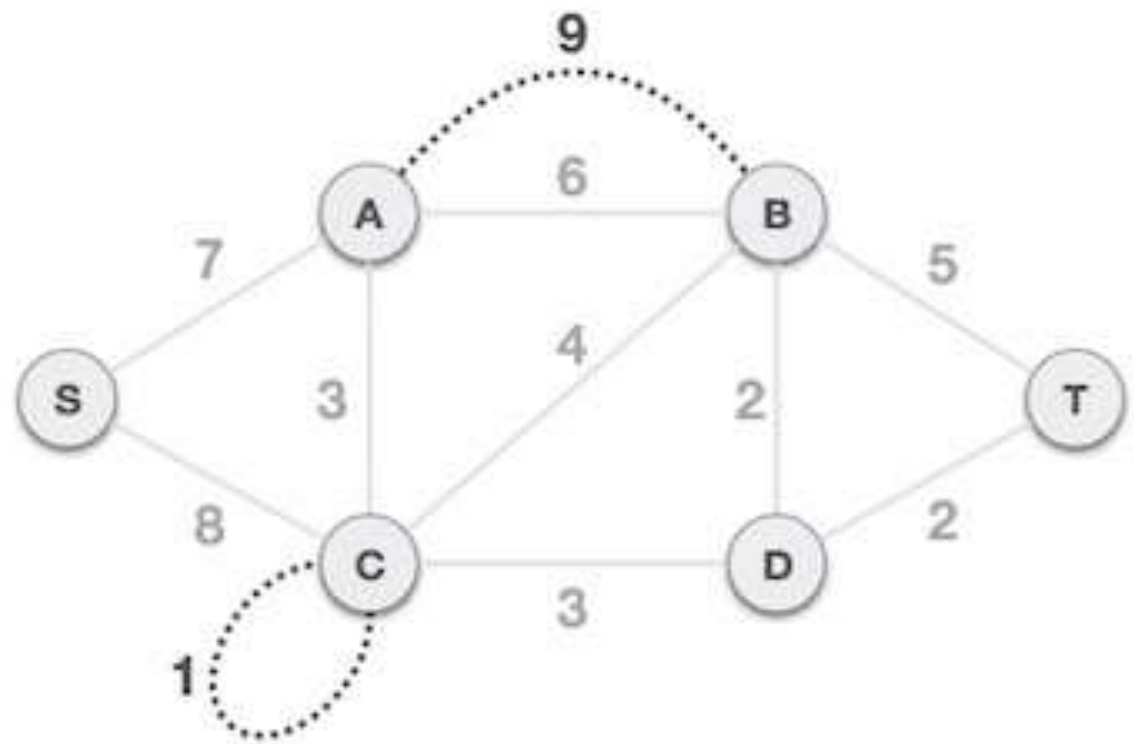
Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

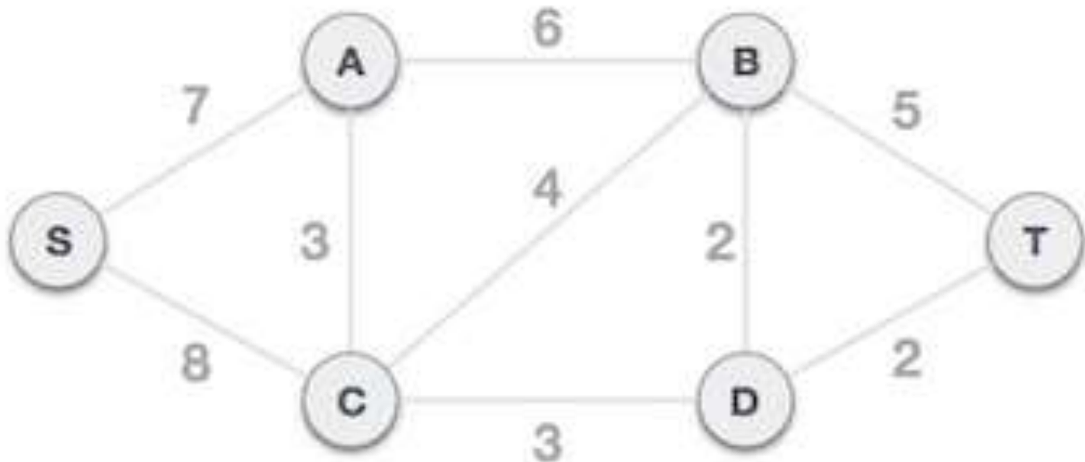
To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example –



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

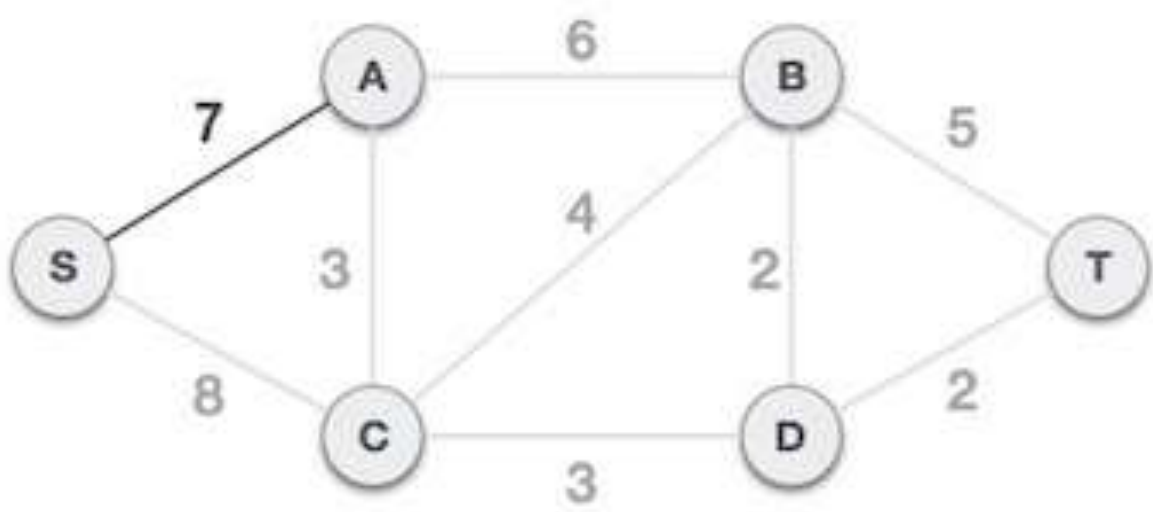


Step 2 - Choose any arbitrary node as root node

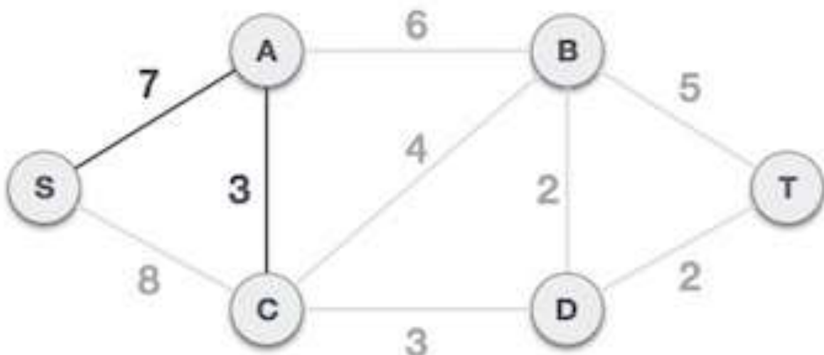
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

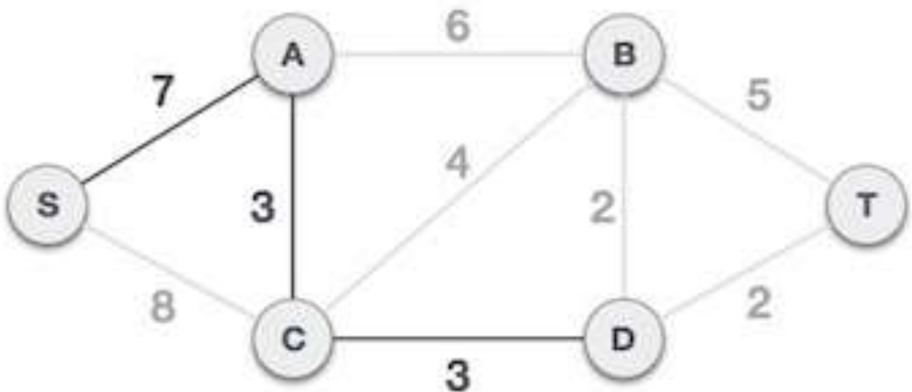
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



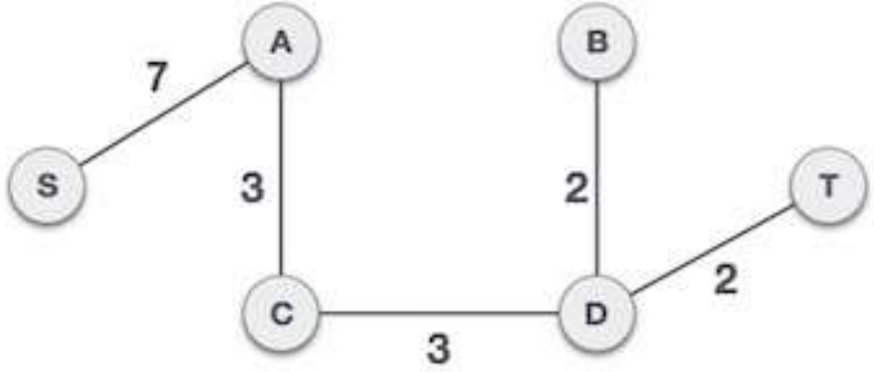
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.