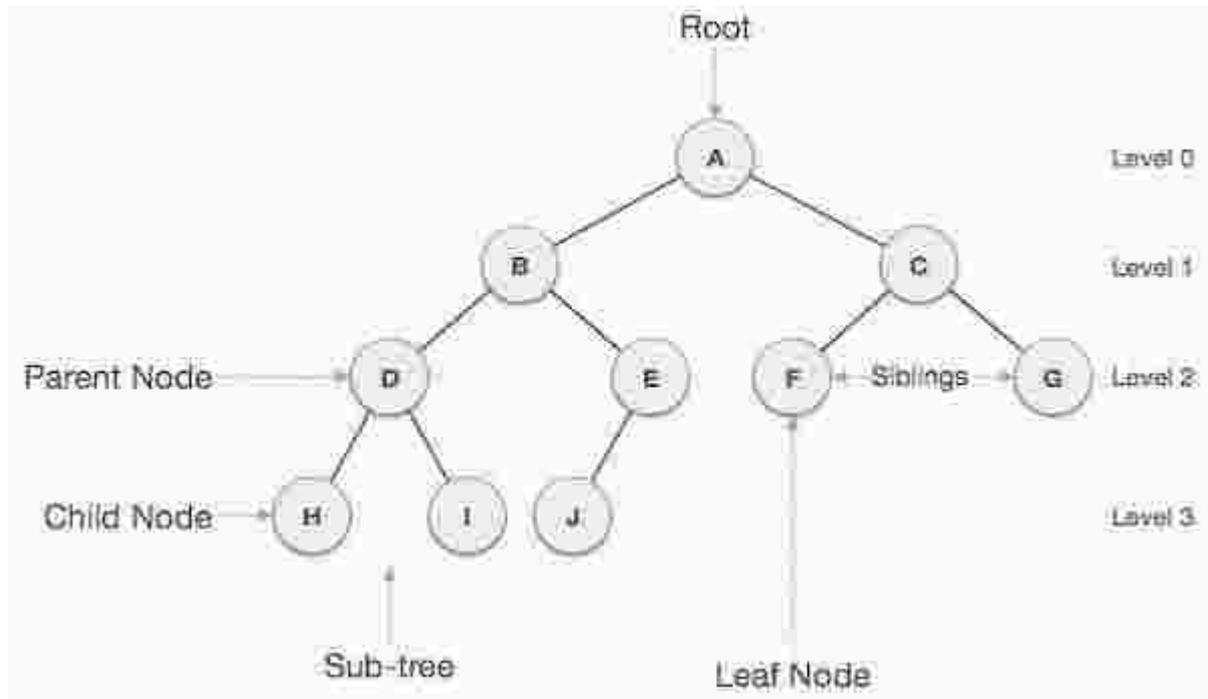


# Trees:

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:

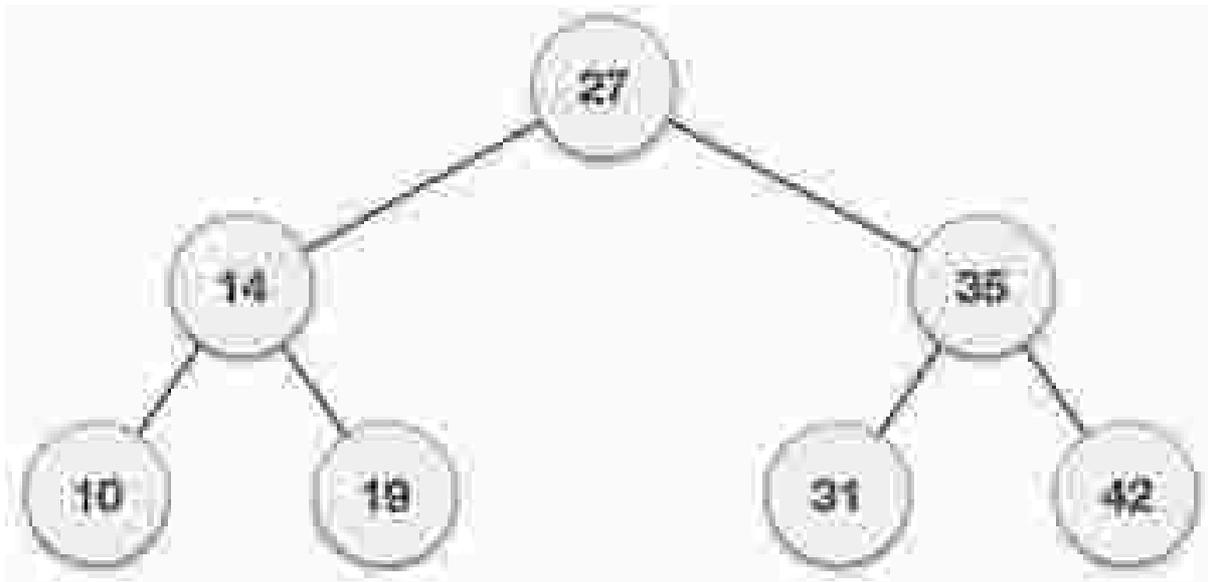
Following are the important terms with respect to tree.

- **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.

## 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;
                }
            }
        }
    }
```



```
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;
        }

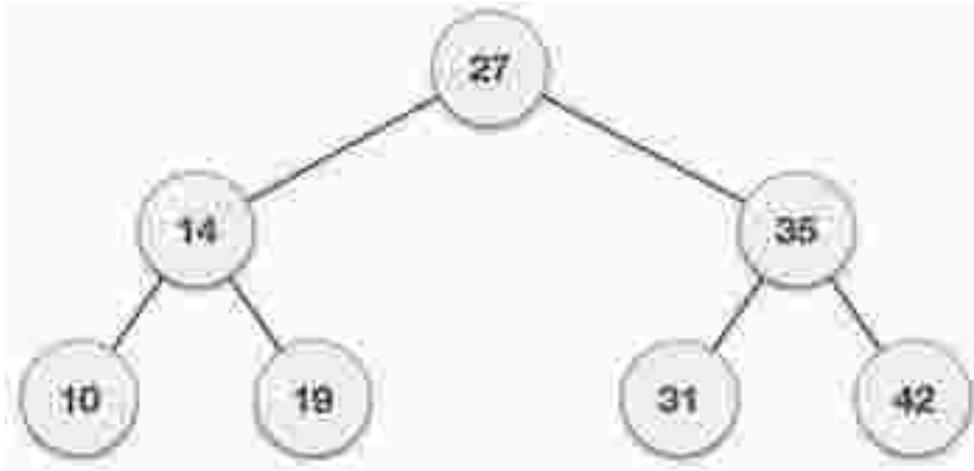
        return current;
    }
}
```

## 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;
```



```

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
```

## 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 –

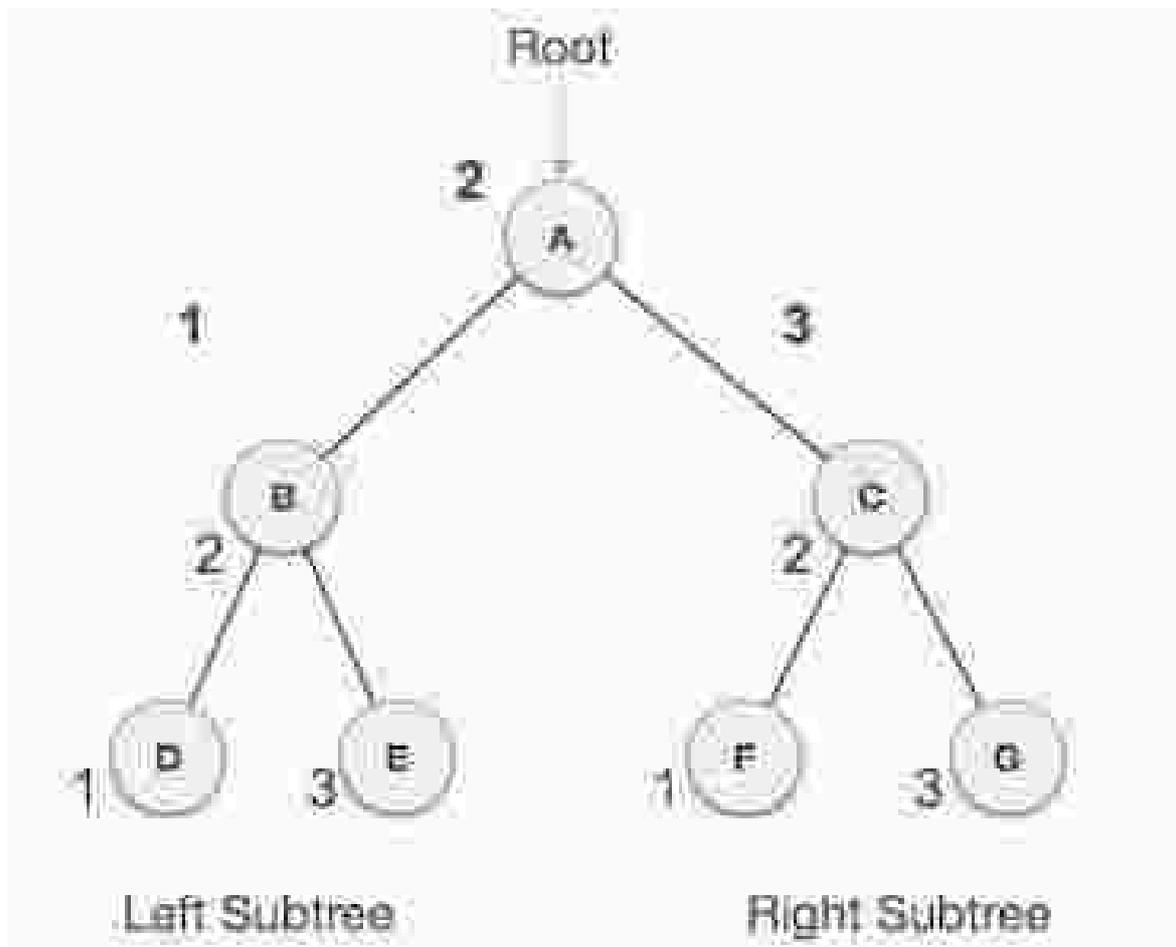
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

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 –

**$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$**

### Algorithm

Until all nodes are traversed –

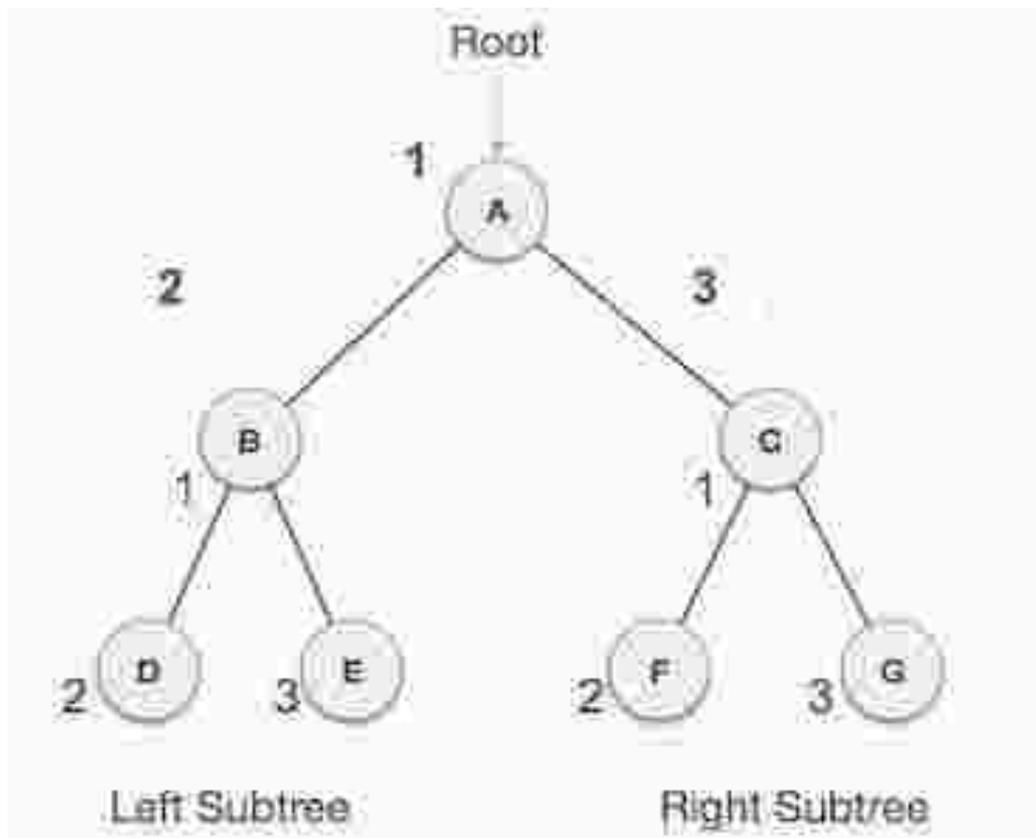
**Step 1** – Recursively traverse left subtree.

**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 –

**$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$**

## Algorithm

Until all nodes are traversed -

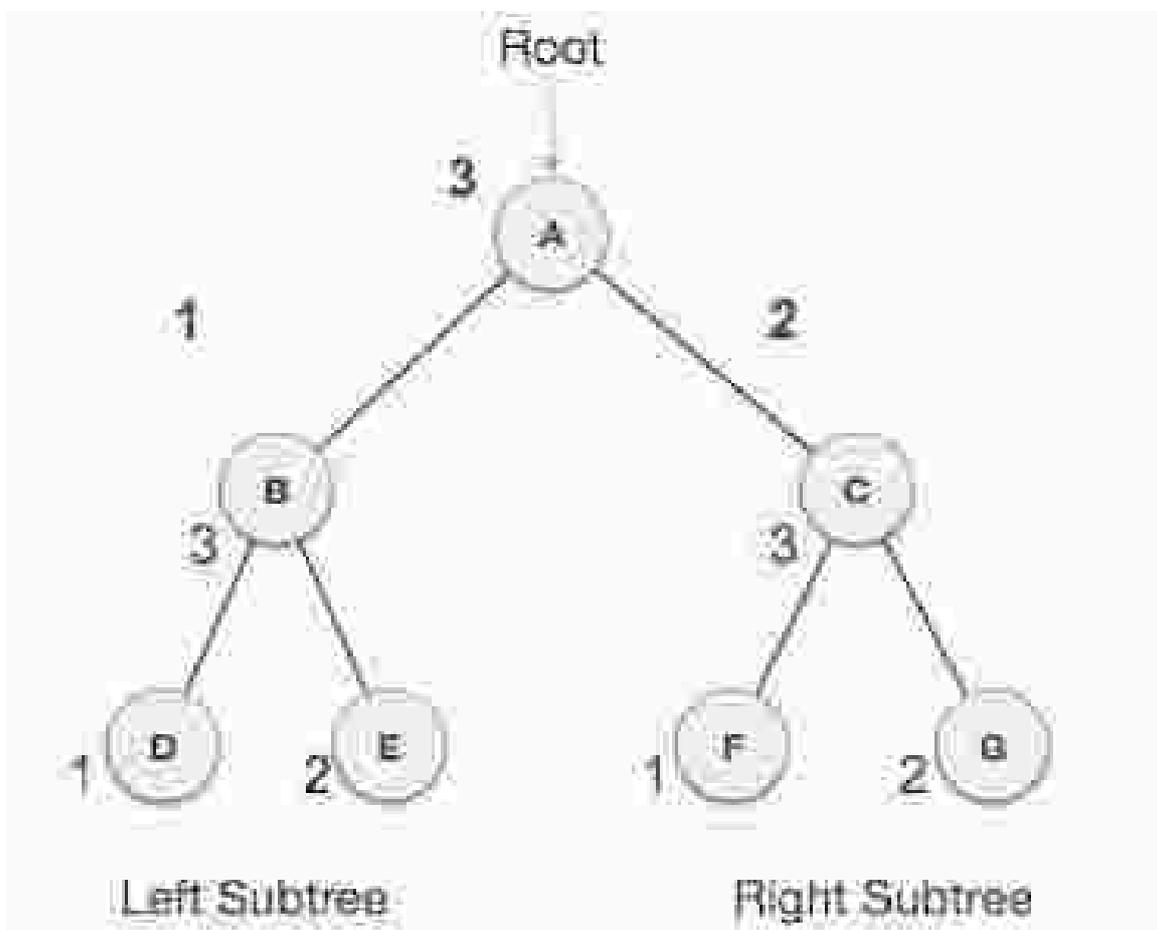
**Step 1** - Visit root node.

**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 -

**$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$**

## Algorithm

Until all nodes are traversed -

**Step 1** - Recursively traverse left subtree.

**Step 2** - Recursively traverse right subtree.

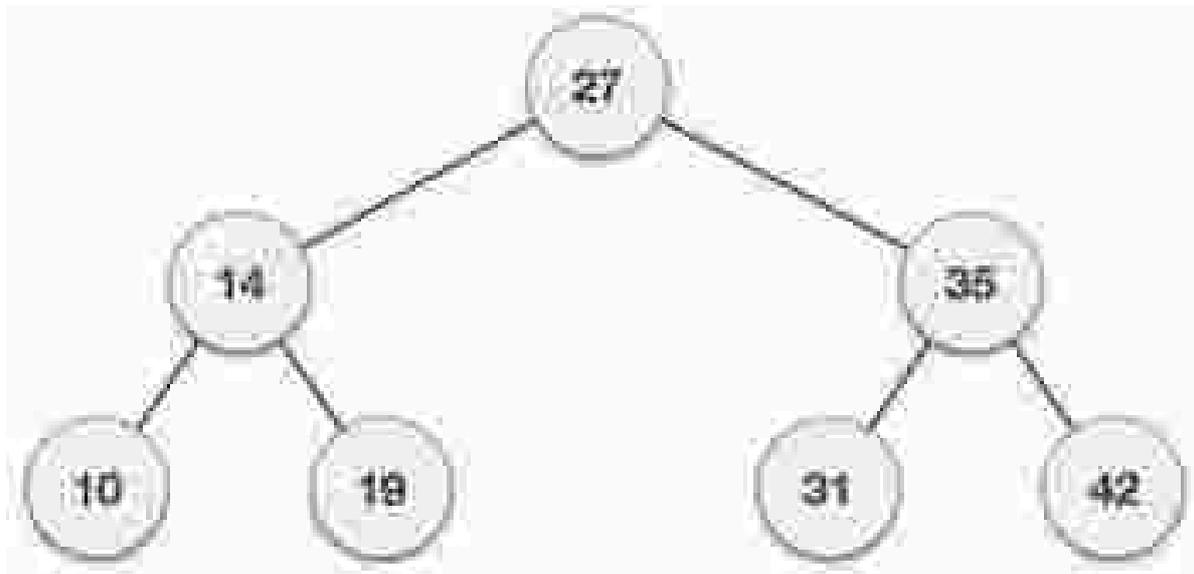
**Step 3** - Visit root node.

## 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
```



```

        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
```

# 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.

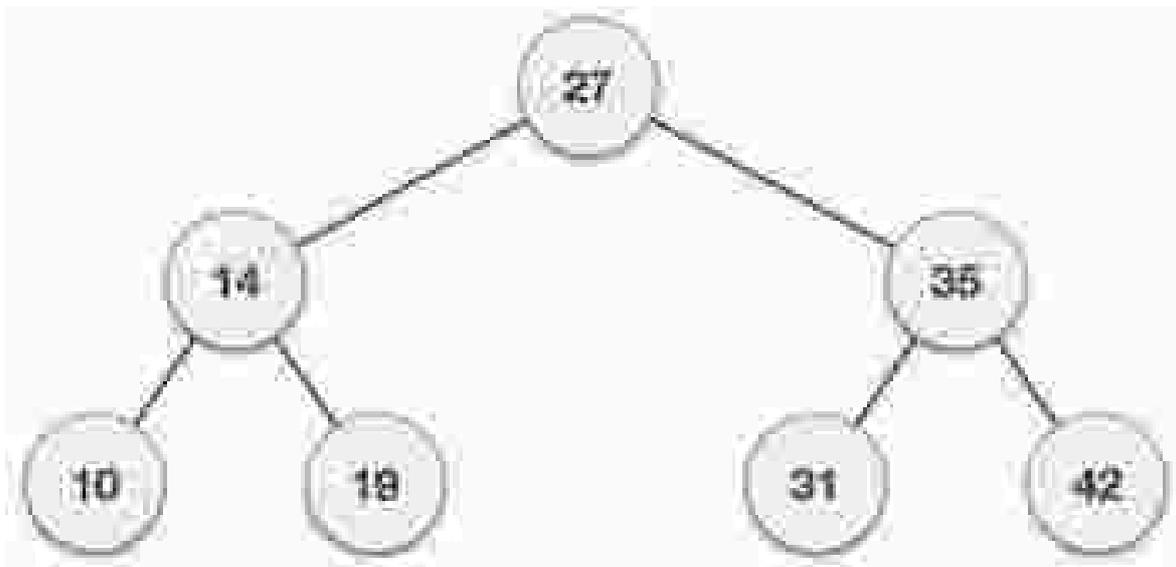
Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as –

$$\text{left\_subtree (keys)} \leq \text{node (key)} \leq \text{right\_subtree (keys)}$$

## 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;

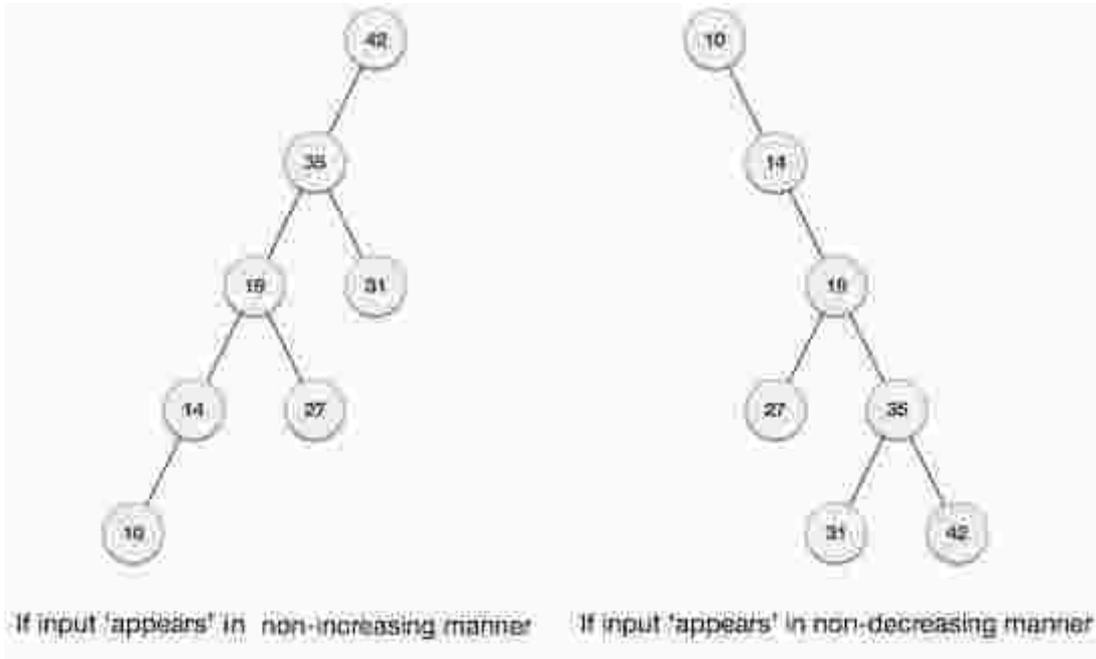
```

```
//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;
    }
}
}
}
}
```

# AVL tress:

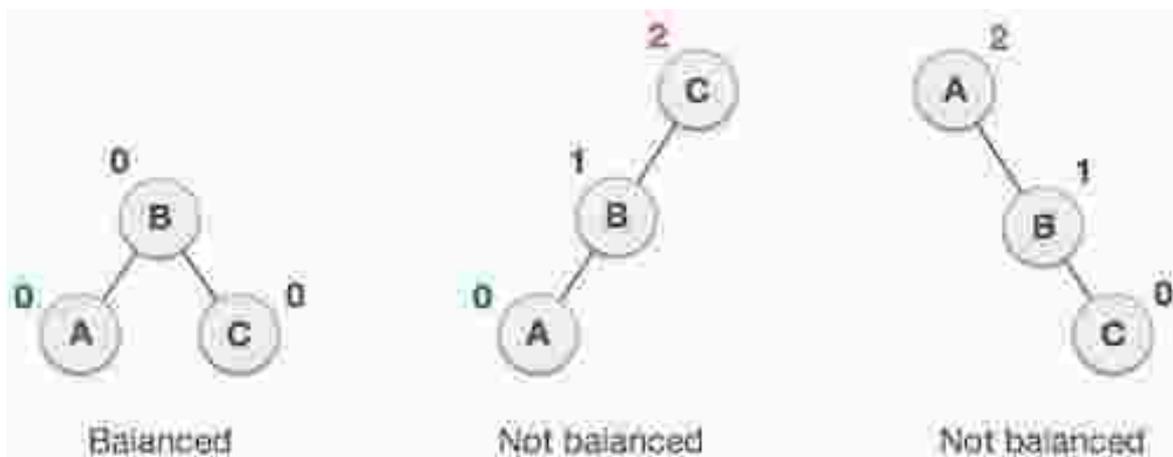
What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this –



It is observed that BST's worst-case performance is closest to linear search algorithms, that is  $O(n)$ . In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor **Adelson, Velski & Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

Here we see that the first tree is balanced and the next two trees are not balanced –



In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

$$\text{BalanceFactor} = \text{height}(\text{left-subtree}) - \text{height}(\text{right-subtree})$$

If the difference in the height of left and right sub-trees is more than 1, the tree is unbalanced using some rotation techniques.

## AVL Rotations:

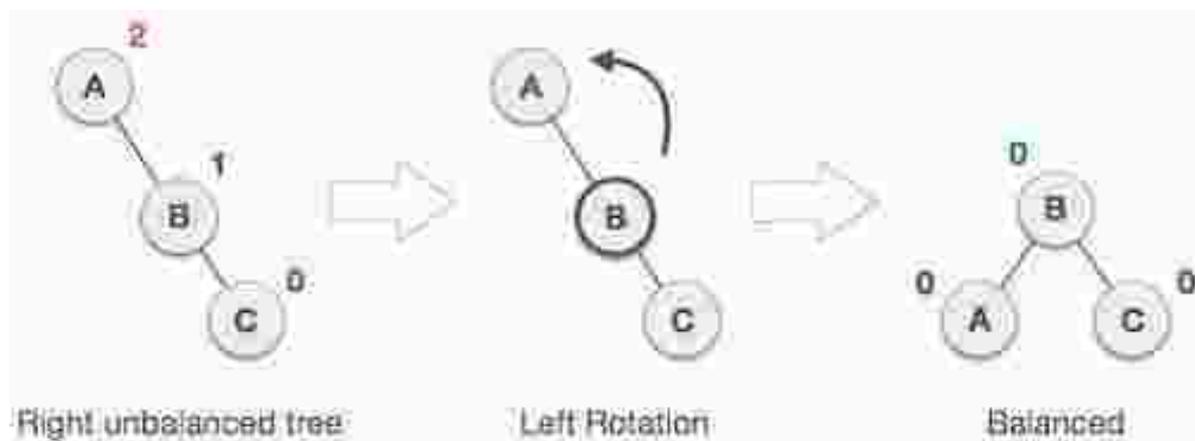
To balance itself, an AVL tree may perform the following four kinds of rotations –

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

### Left Rotation

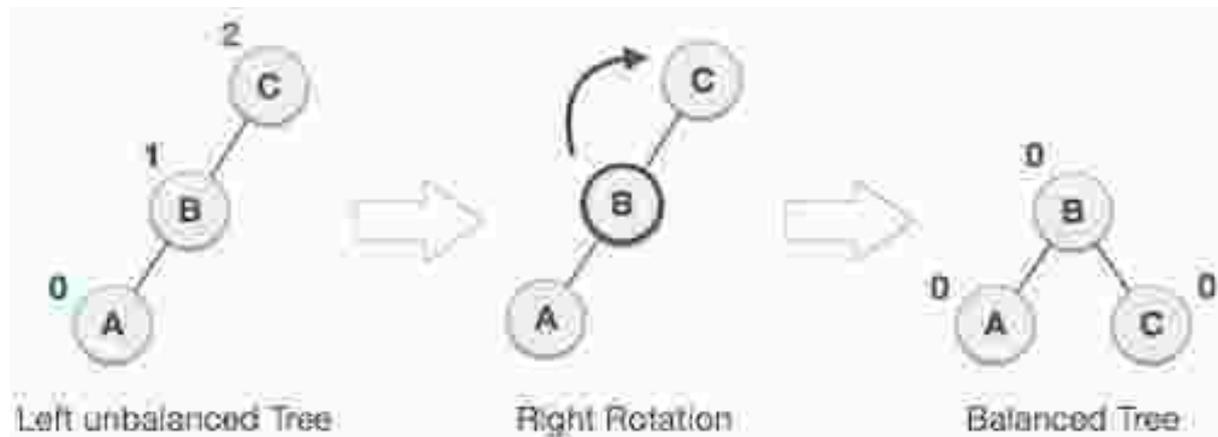
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



In our example, node **A** has become unbalanced as a node is inserted in the right subtree of **A**'s right subtree. We perform the left rotation by making **A** the left-subtree of **B**.

## Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

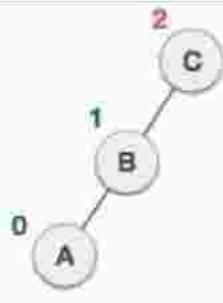
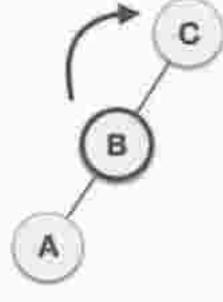
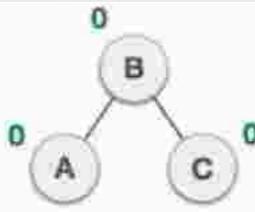


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

## Left-Right Rotation

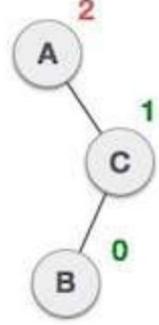
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

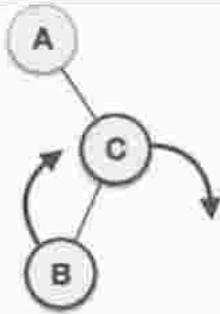
State	Action
	<p>A node has been inserted into the right subtree of the left subtree. This makes <b>C</b> an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.</p>
	<p>We first perform the left rotation on the left subtree of <b>C</b>. This makes <b>A</b>, the left subtree of <b>B</b>.</p>

	<p>Node <b>C</b> is still unbalanced, however now, it is because of the left-subtree of the left-subtree.</p>
	<p>We shall now right-rotate the tree, making <b>B</b> the new root node of this subtree. <b>C</b> now becomes the right subtree of its own left subtree.</p>
	<p>The tree is now balanced.</p>

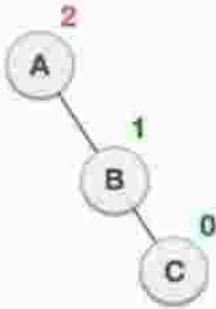
### Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

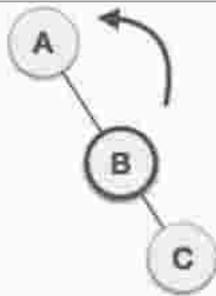
State	Action
	<p>A node has been inserted into the left subtree of the right subtree. This makes <b>A</b>, an unbalanced node with balance factor 2.</p>



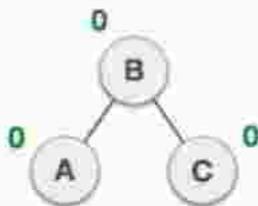
First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.



Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.



A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.

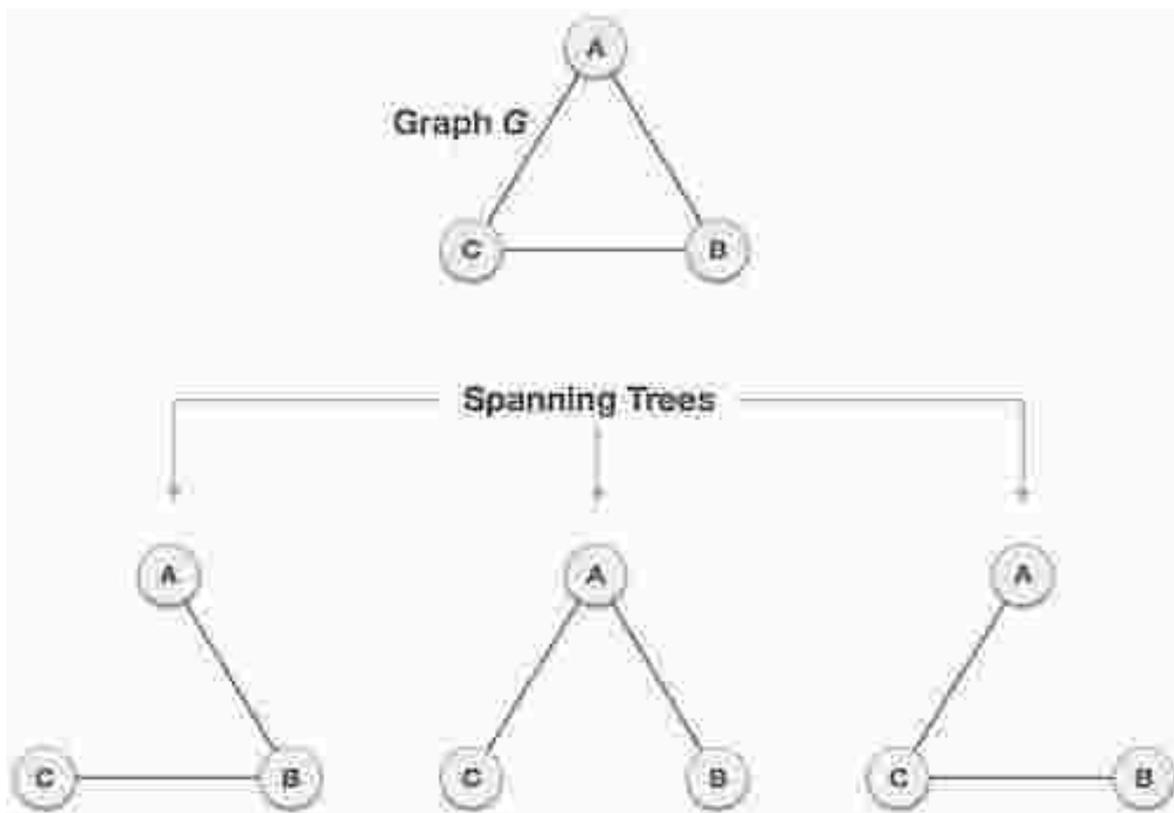


The tree is now balanced.

# 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  $n^{n-2}$  number of spanning trees, where  $n$  is the number of nodes. In the above addressed example,  $n$  is 3, hence  $3^{3-2} = 3$  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 –

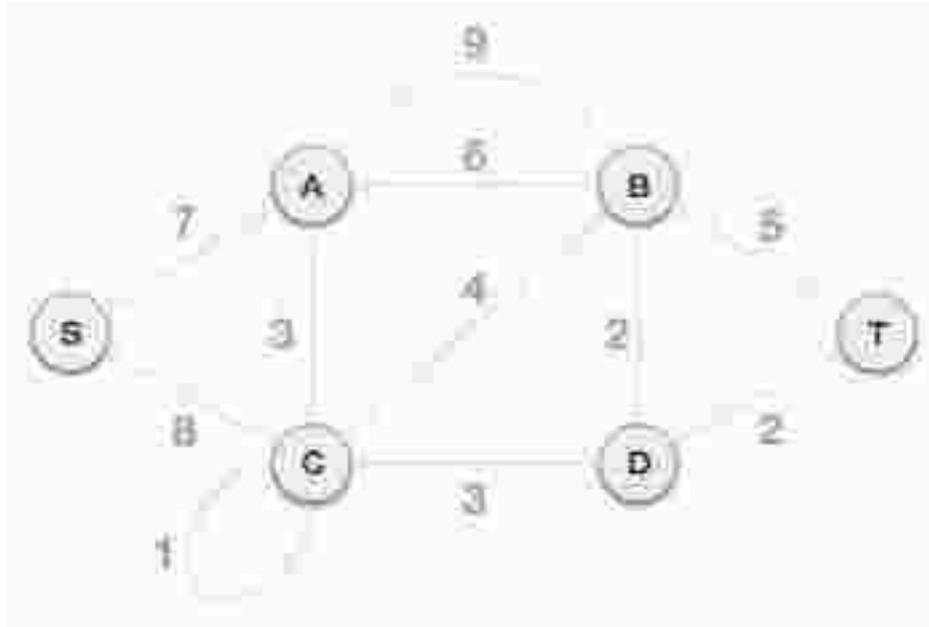
- Kruskal's Algorithm
- Prim's Algorithm Both

are greedy algorithms.

## Kruskal's Spanning Tree Algorithm:

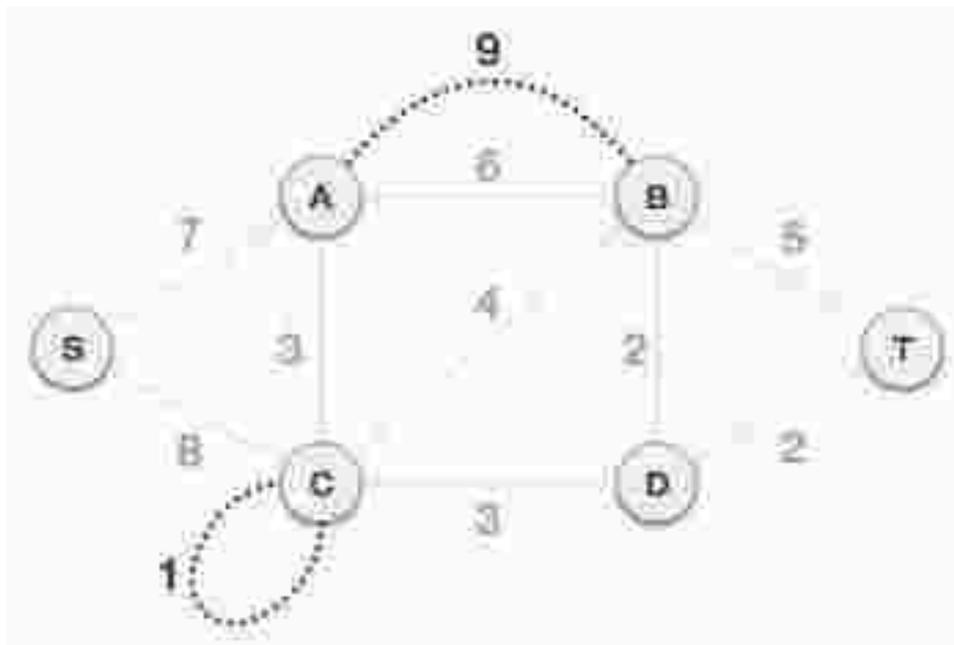
Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only if it has the least cost among all available options and does not violate MST properties.

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

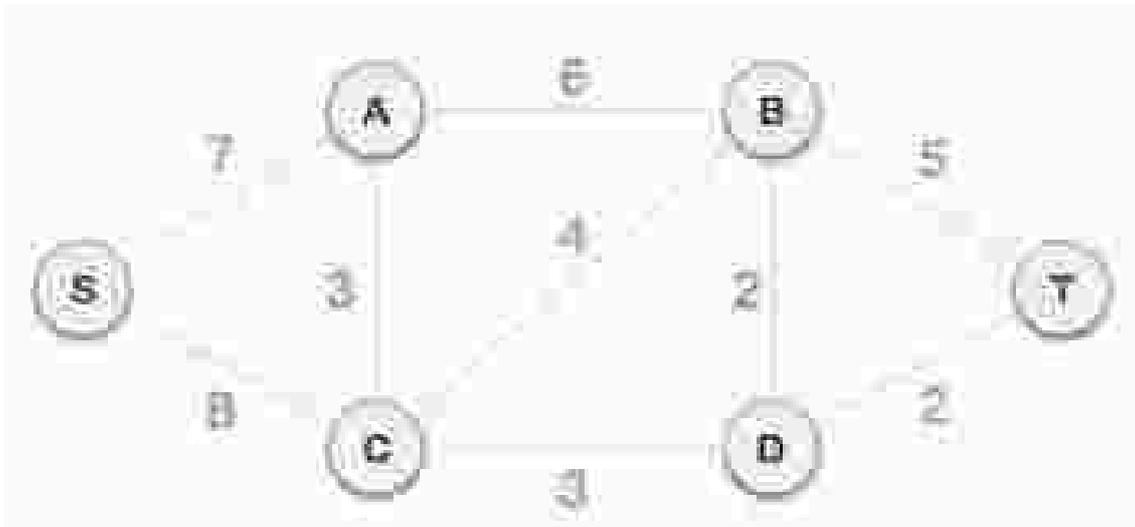


### Step 1 - Remove all loops and parallel edges

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.



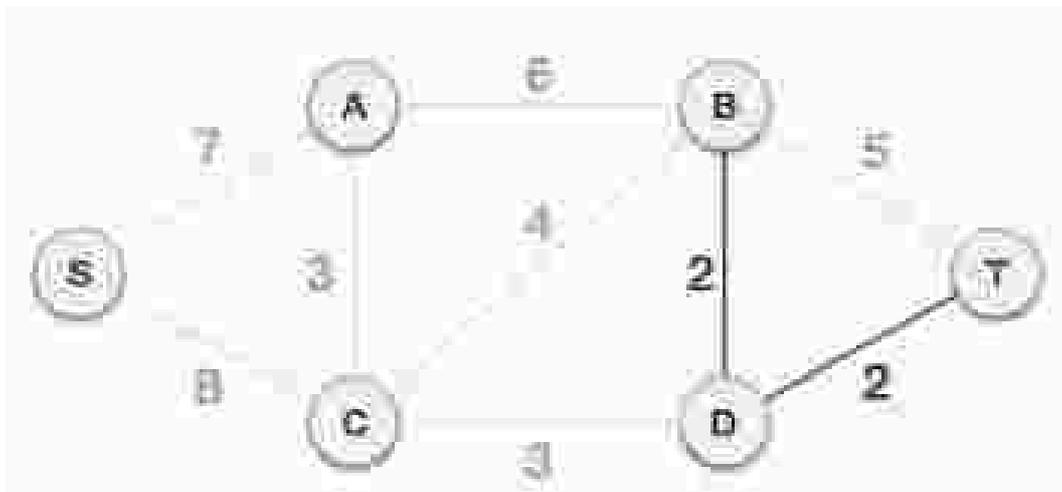
**Step 2 - Arrange all edges in their increasing order of weight**

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

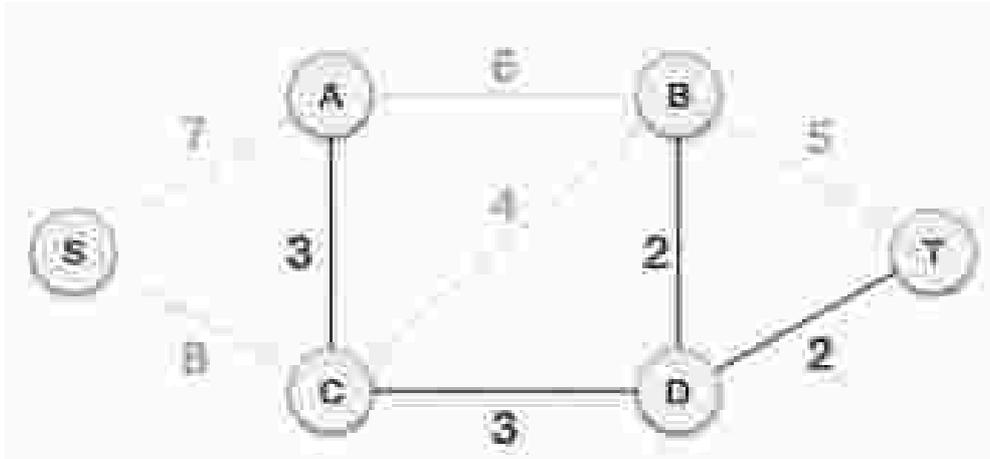
**Step 3 - Add the edge which has the least weightage**

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.

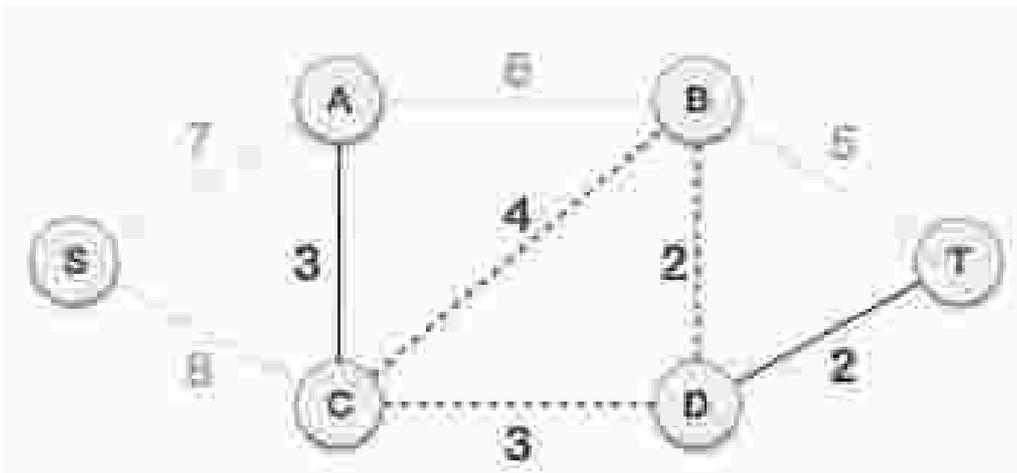


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

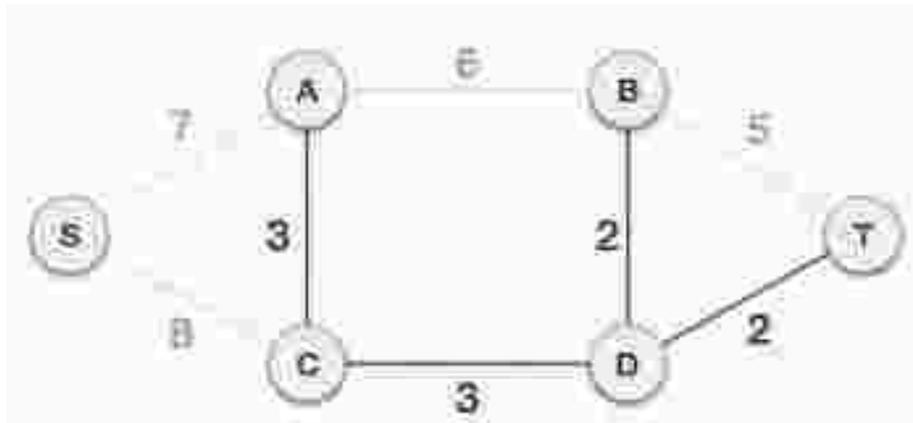
Next cost is 3, and associated edges are A,C and C,D. We add them again –



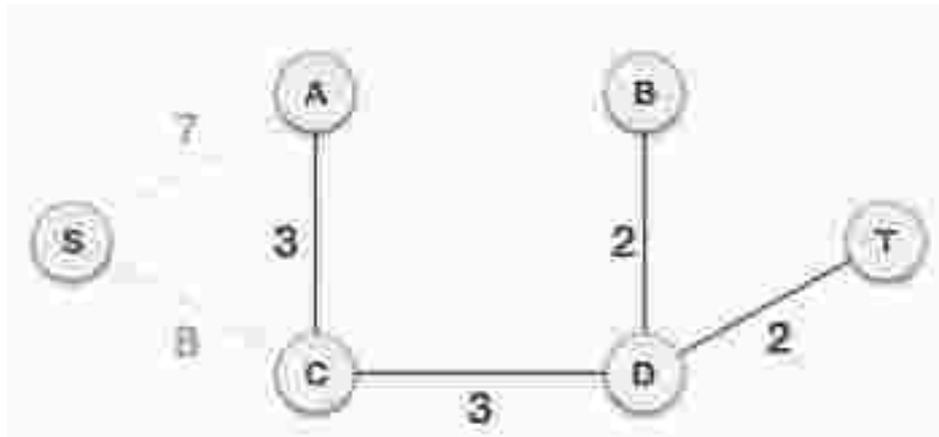
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph.



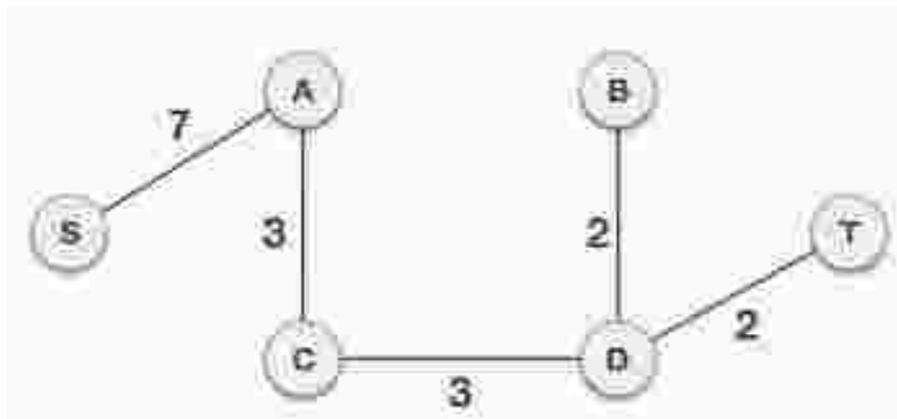
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



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.



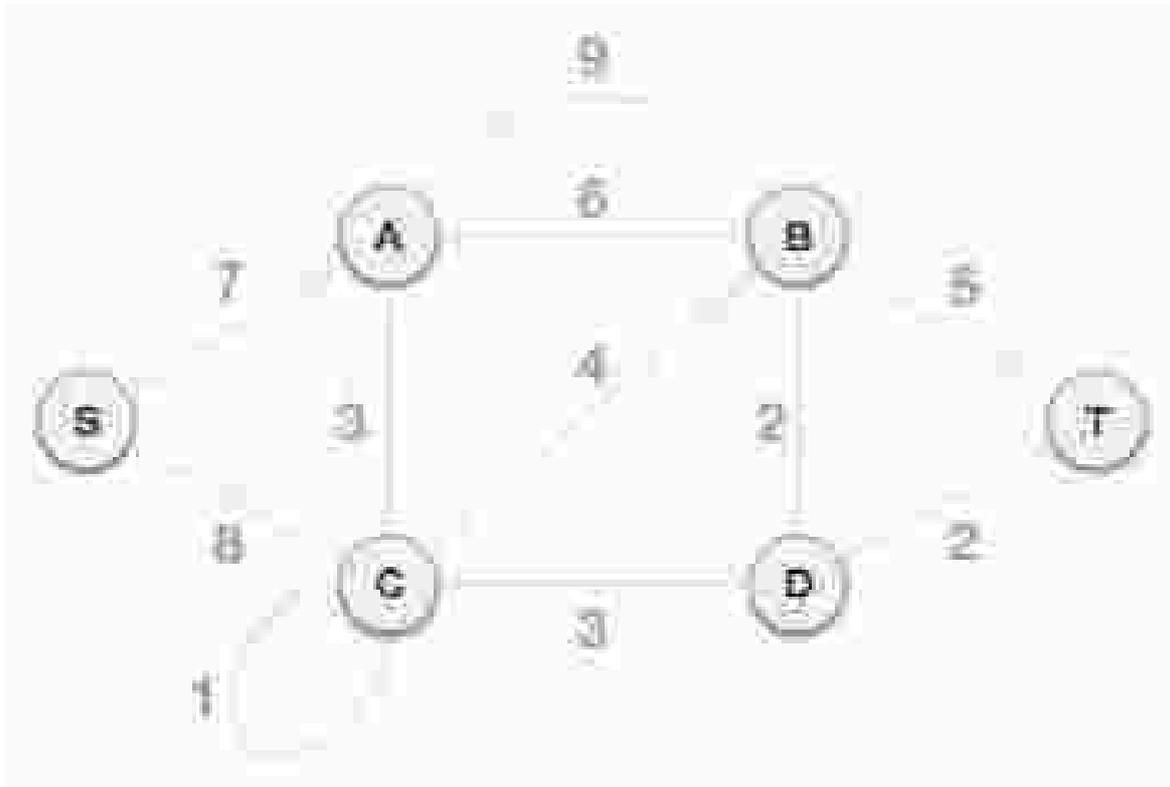
By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

### Prim's Spanning Tree Algorithm:

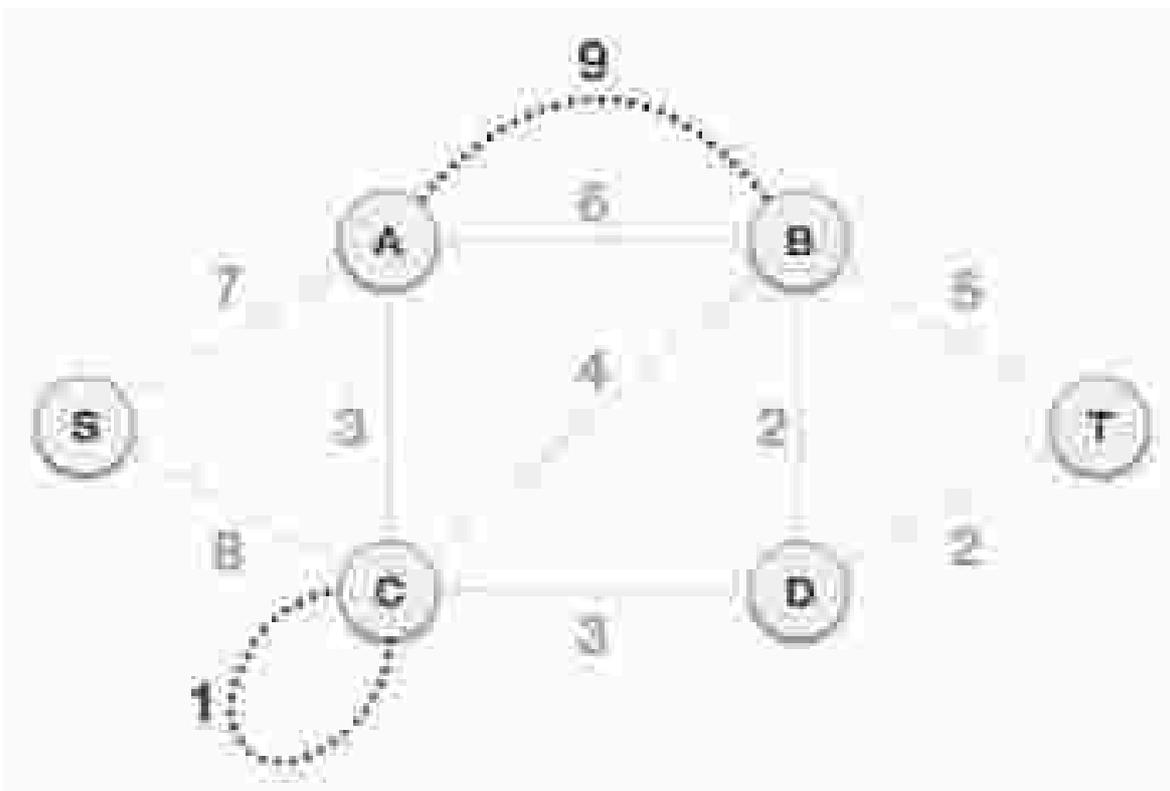
Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

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.

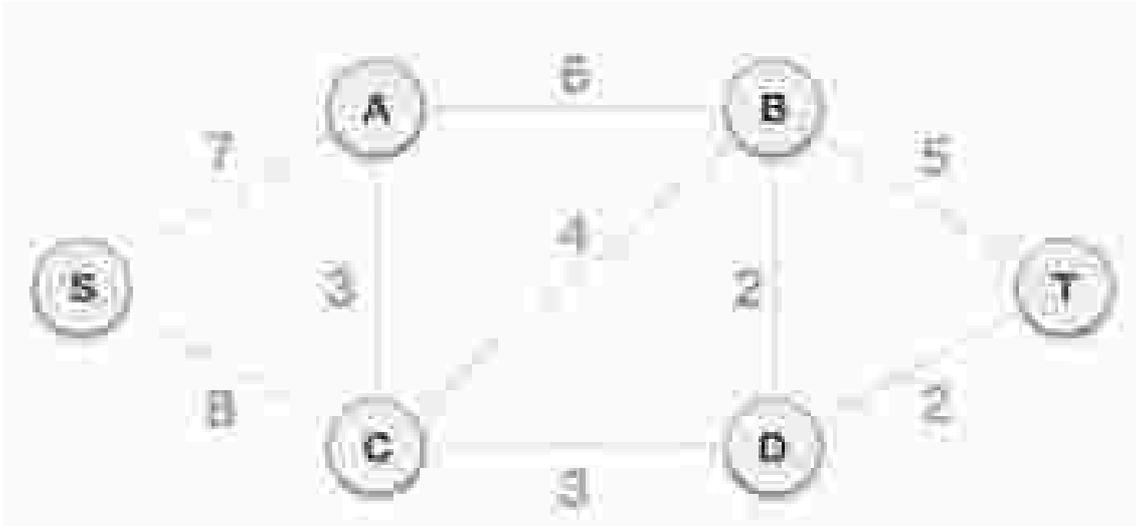
To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example –



**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.

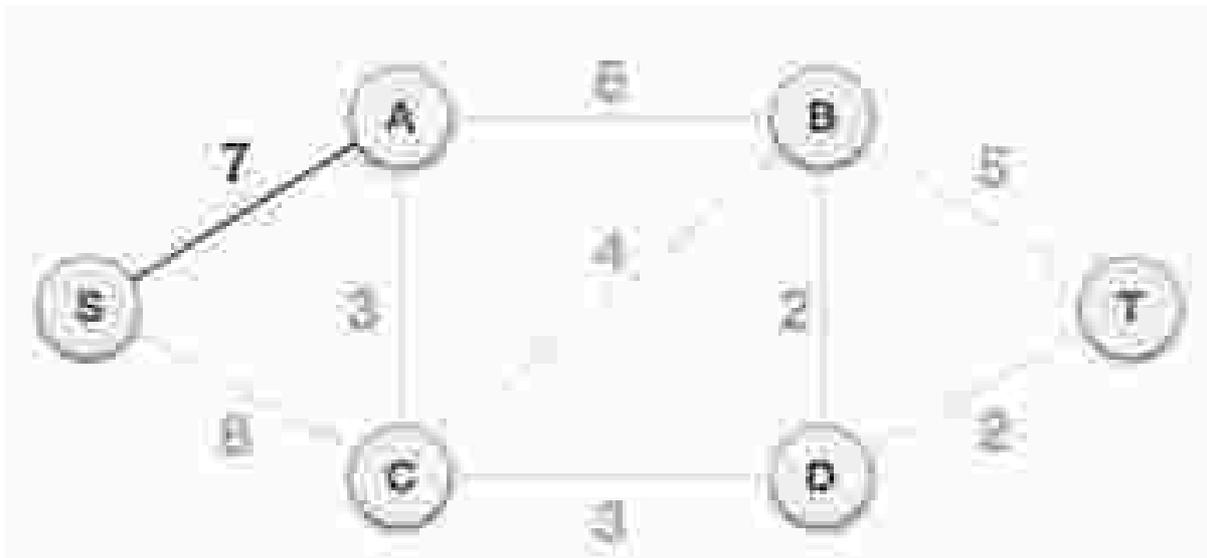


### Step 2 - Choose any arbitrary node as root node

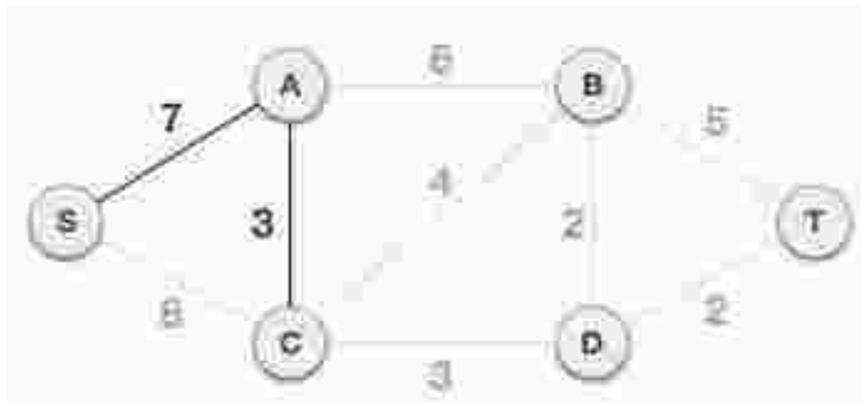
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then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

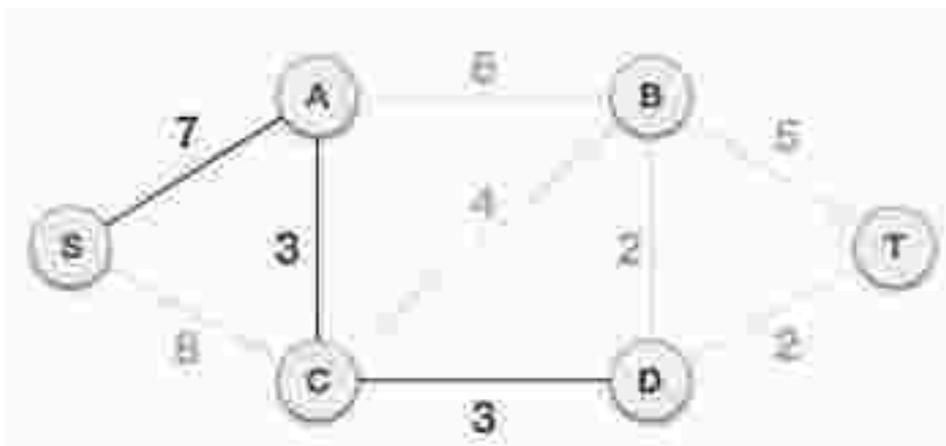
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



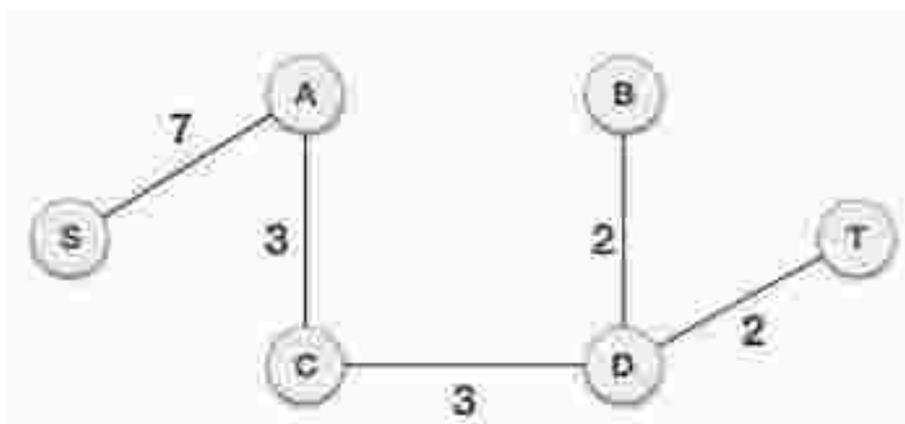
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



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.

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

## References:

1. "Data Structures & Algorithm" , Tutorials Point India (<https://www.tutorialspoint.com>)