

UNIT IV

Data Structure - Binary Search Tree

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

- The value of the key of the left sub-tree is less than the value of its parent (root) node's key.
- The value of the key of the right sub-tree is greater than or equal to the value of its parent (root) node's key.

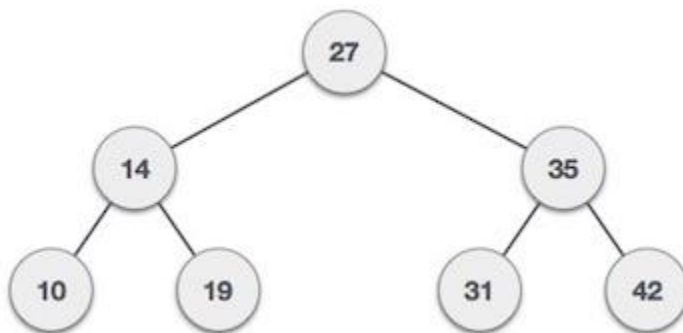
Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as –

$\text{left_subtree (keys)} < \text{node (key)} \leq \text{right_subtree (keys)}$

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 –

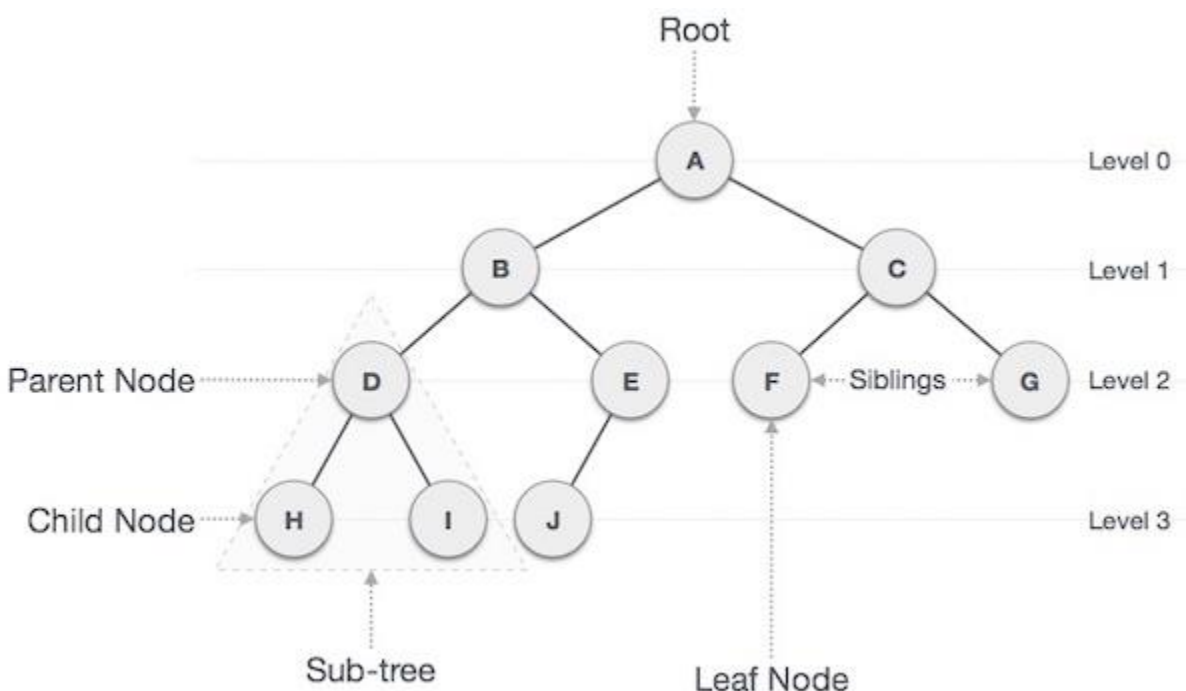
- **Search** – Searches an element in a tree.

- **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.

Data Structure and Algorithms - Tree

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

Binary Tree is a special data structure 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.



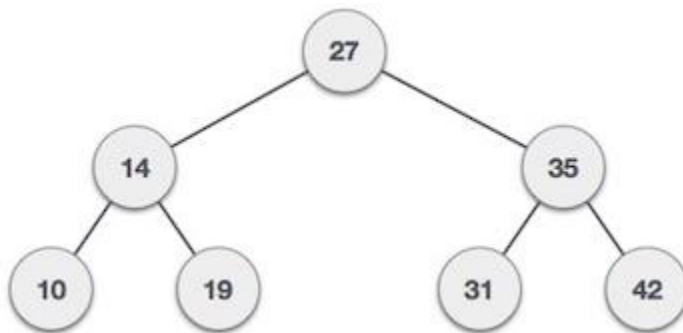
Important Terms

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.
- **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.
- **Preorder Traversal** – Traverses a tree in a pre-order manner.
- **Inorder Traversal** – Traverses a tree in an in-order manner.
- **Postorder 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
```

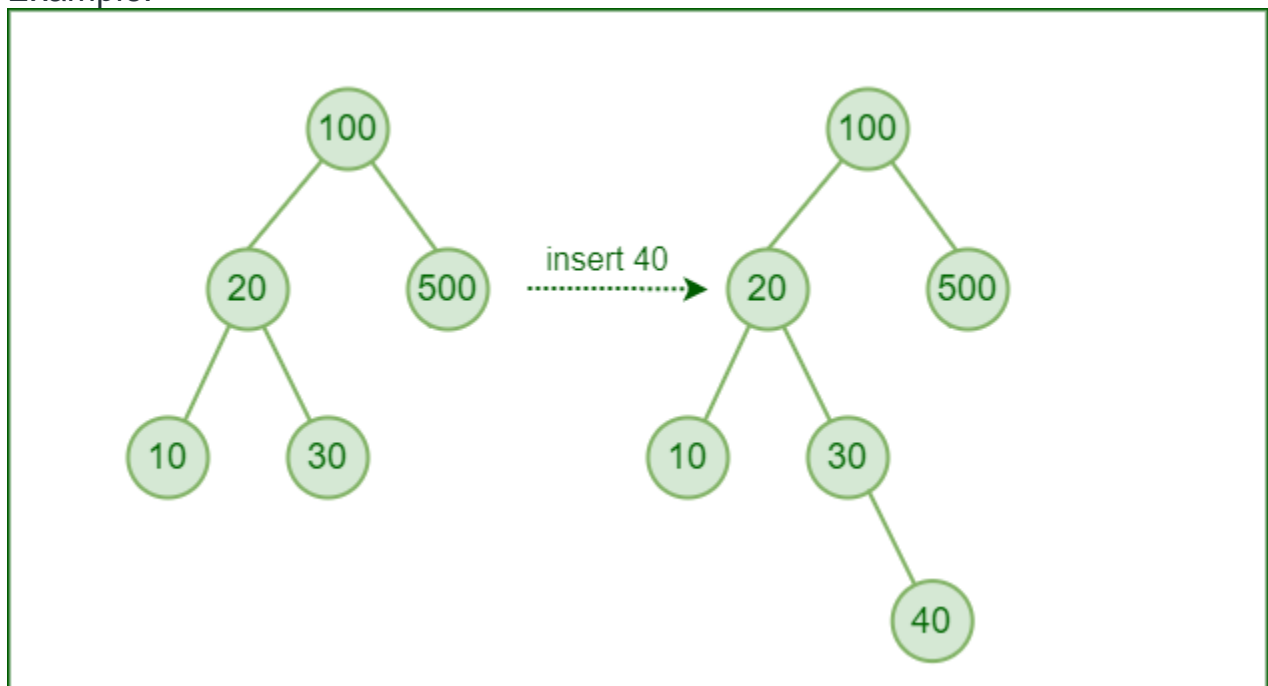
```
insert data
```

```
end If
```

Insertion in Binary Search Tree (BST)

Given a **BST**, the task is to insert a new node in this **BST**.

Example:



Insertion in Binary Search Tree

How to Insert a value in a Binary Search Tree:

A new key is always inserted at the leaf by maintaining the property of the binary search tree. We start searching for a key from the root until we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf

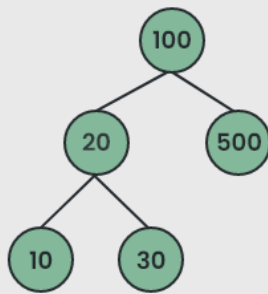
node. The below steps are followed while we try to insert a node into a binary search tree:

- Check the value to be inserted (say **X**) with the value of the current node (say **val**) we are in:
 - If **X** is less than **val** move to the left subtree.
 - Otherwise, move to the right subtree.
- Once the leaf node is reached, insert **X** to its right or left based on the relation between **X** and the leaf node's value.

Follow the below illustration for a better understanding:

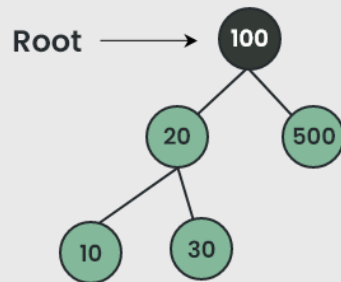
Illustration:

Consider The Following BST



X = 40 (The Node To Be Inserted)

STEP 1 : Comparing X with Root Node



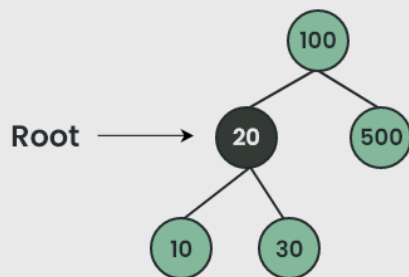
Since 100 Is Greater Than 40.
Move Pointer To The Left Child (20)

Insertion In BST



Insertion in BST

STEP 2 : Comparing X with left child of root node



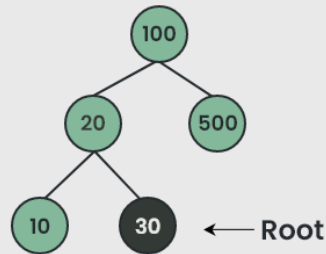
Since 20 Is Less Than 40, Move
Pointer To The Right Child (30)

Insertion In BST



Insertion in BST

STEP 3 : Comparing x with the right child of 20



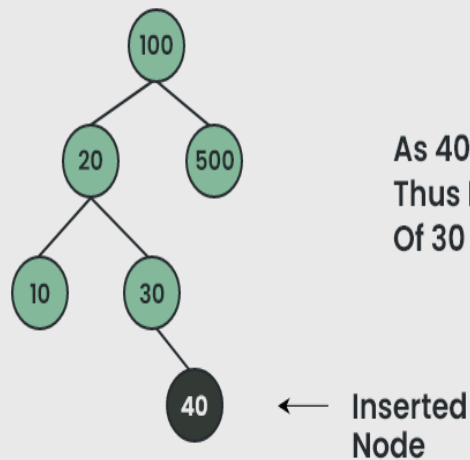
Again 40 Is Greater Than 30
Move Pointer To The Right Side
Of 30

Insertion In BST



Insertion in BST

STEP 4 :Insert item to the right of 30



As 40 Is Greater Than The Node 30,
Thus It Will Be Inserted To The Right
Of 30

Insertion In BST



Insertion in BST

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
```

Searching in Binary Search Tree (BST)

Given a **BST**, the task is to search a node in this **BST**.

*For searching a value in BST, consider it as a sorted array. Now we can easily perform search operation in BST using **Binary Search Algorithm**.*

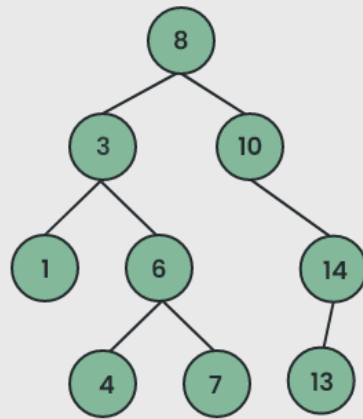
Algorithm to search for a key in a given Binary Search Tree:

Let's say we want to search for the number **X**, We start at the root. Then:

- We compare the value to be searched with the value of the root.
 - If it's equal we are done with the search if it's smaller we know that we need to go to the left subtree because in a binary search tree all the elements in the left subtree are smaller and all the elements in the right subtree are larger.
- Repeat the above step till no more traversal is possible
- If at any iteration, key is found, return True. Else False.

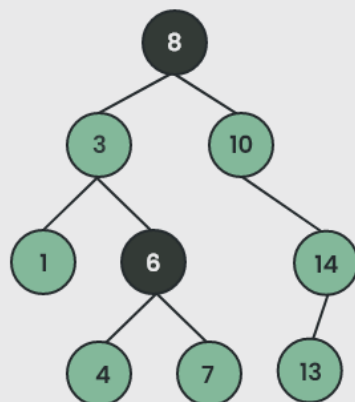
Illustration of searching in a BST:

See the illustration below for a better understanding:



Consider The Following BST
Key = 6

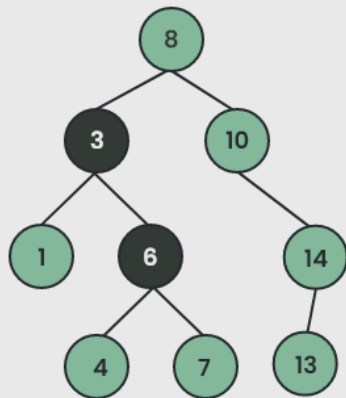
Searching In BST



Compare Key With Root, i.e 8
as $6 < 8$, search in left subtree
of 8

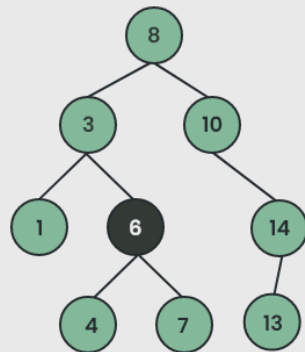
Searching In BST





As Key (6) Is Greater Than 3,
Search In The Right Subtree Of 3

Searching In BST



As 6 Is Equal To Key (6), So We Have
Found The Key

Searching In BST



Deletion in Binary Search Tree (BST)

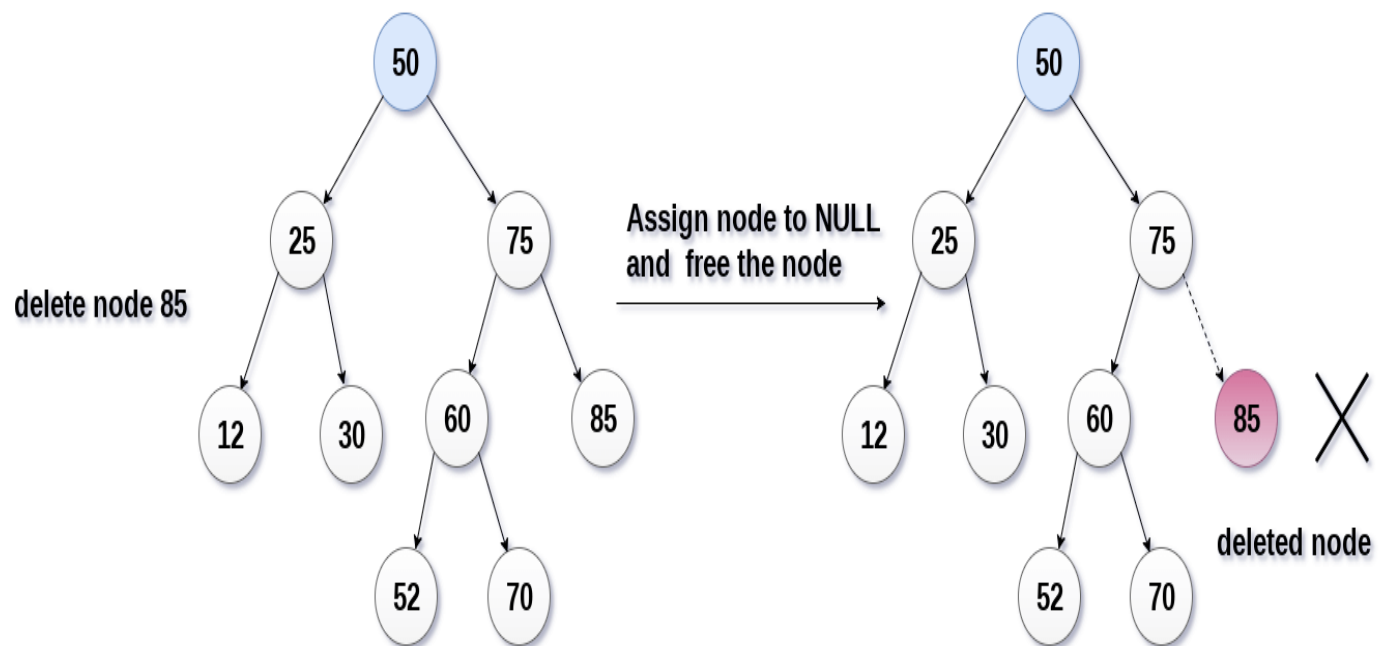
Deletion

Delete function is used to delete the specified node from a binary search tree. However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate. There are three situations of deleting a node from binary search tree.

The node to be deleted is a leaf node

It is the simplest case, in this case, replace the leaf node with the NULL and simply free the allocated space.

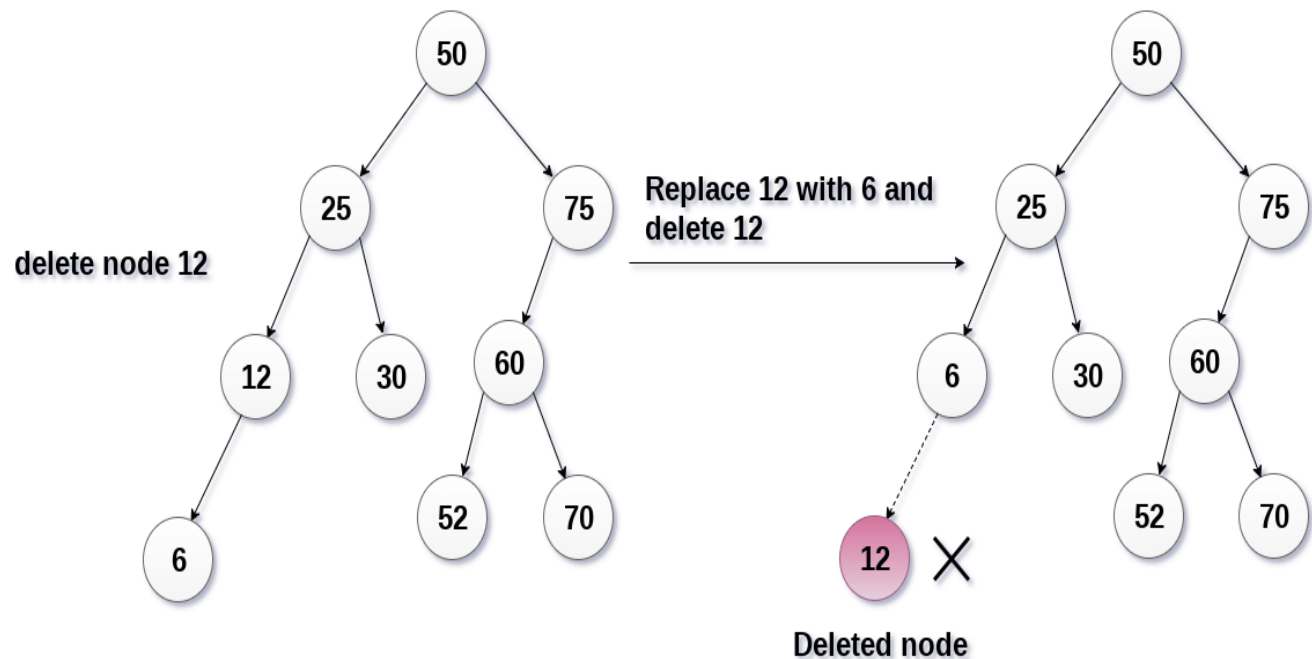
In the following image, we are deleting the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.



The node to be deleted has only one child.

In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.

In the following image, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node) will simply be deleted.



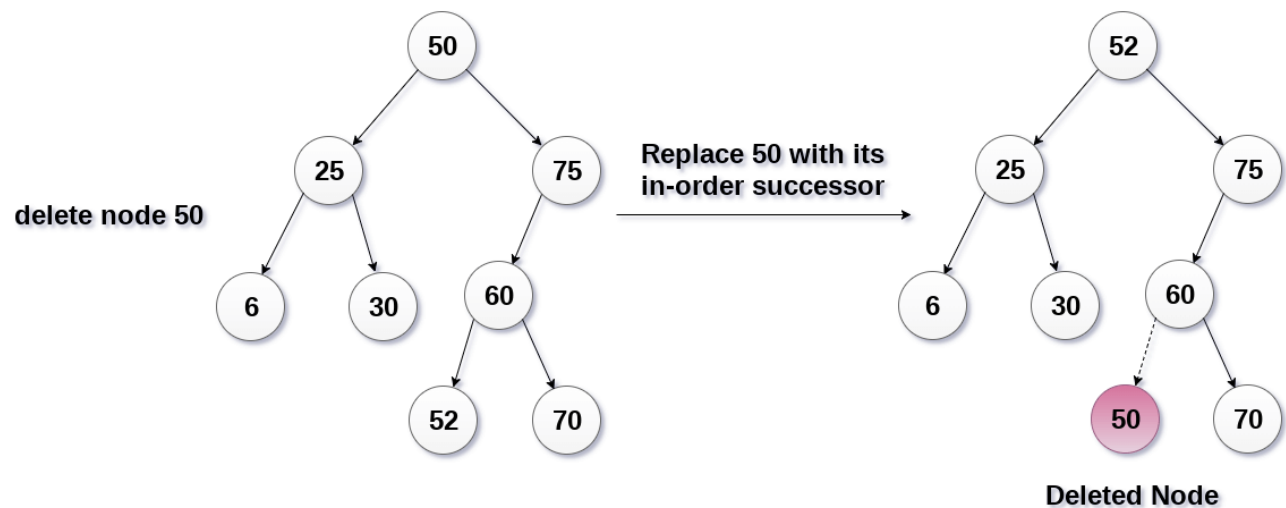
The node to be deleted has two children.

It is a bit complex case compare to other two cases. However, the node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.

In the following image, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below.

6, 25, 30, 50, 52, 60, 70, 75.

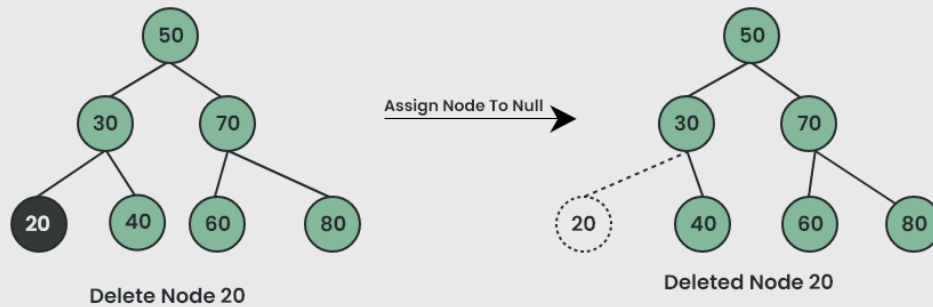
replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.



Given a **BST**, the task is to delete a node in this **BST**, which can be broken down into 3 scenarios:

Case 1. Delete a Leaf Node in BST

Case 1 : Delete A Leaf Node In BST



Deletion In BST

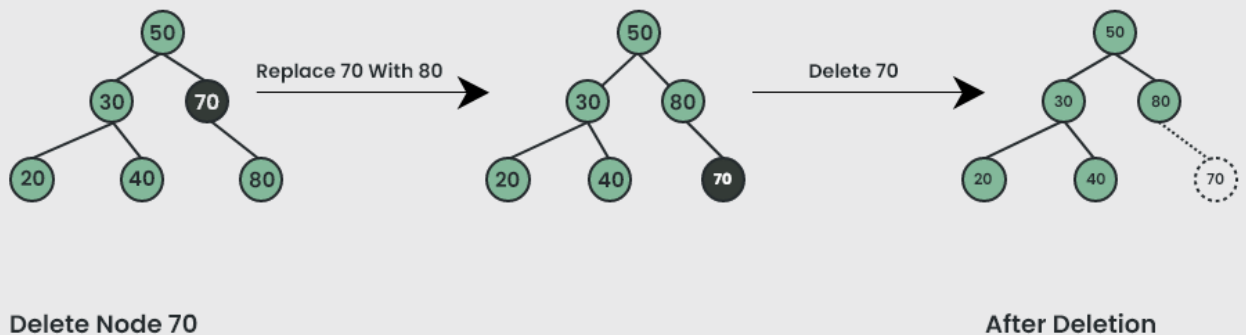


Deletion in BST

Case 2. Delete a Node with Single Child in BST

Deleting a single child node is also simple in BST. Copy the child to the node and delete the node.

Case 2: Delete A Node With Single Child In BST



Deletion In BST

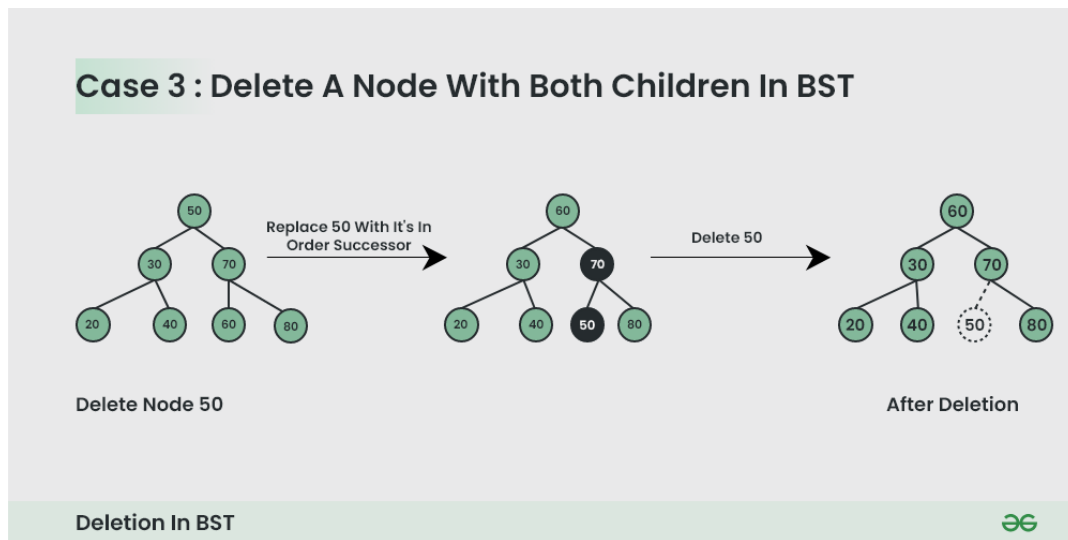


Deletion in BST

Case 3. Delete a Node with Both Children in BST

Deleting a node with both children is not so simple. Here we have to delete the node in such a way, that the resulting tree follows the properties of a BST.

The trick is to find the inorder successor of the node. Copy contents of the inorder successor to the node, and delete the inorder successor.



Deletion in Binary Tree

Note: Inorder successor is needed only when the right child is not empty. In this particular case, the inorder successor can be obtained by finding the minimum value in the right child of the node.

Data Structure & Algorithms - 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 –

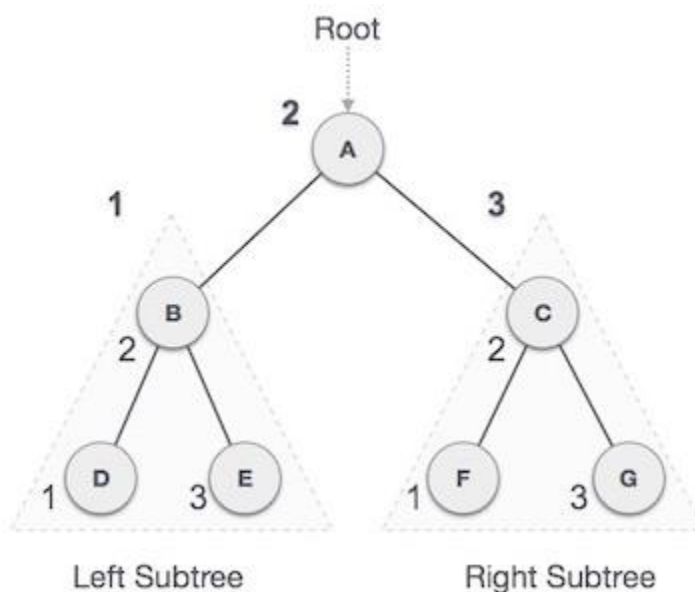
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

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 inorder 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 –

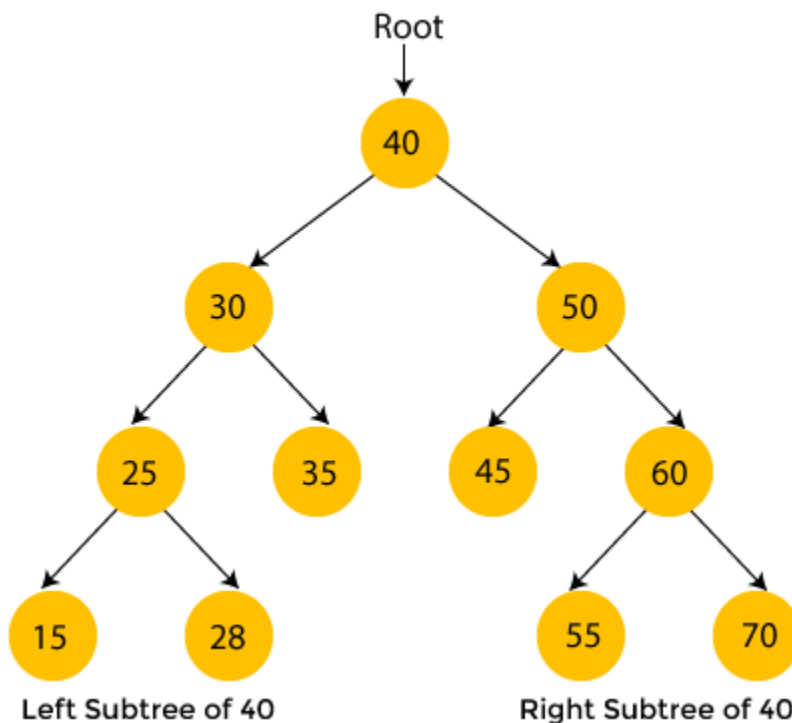
Step 1 – Recursively traverse left subtree.

Step 2 – Visit root node.

Step 3 – Recursively traverse right subtree.

Example of inorder traversal

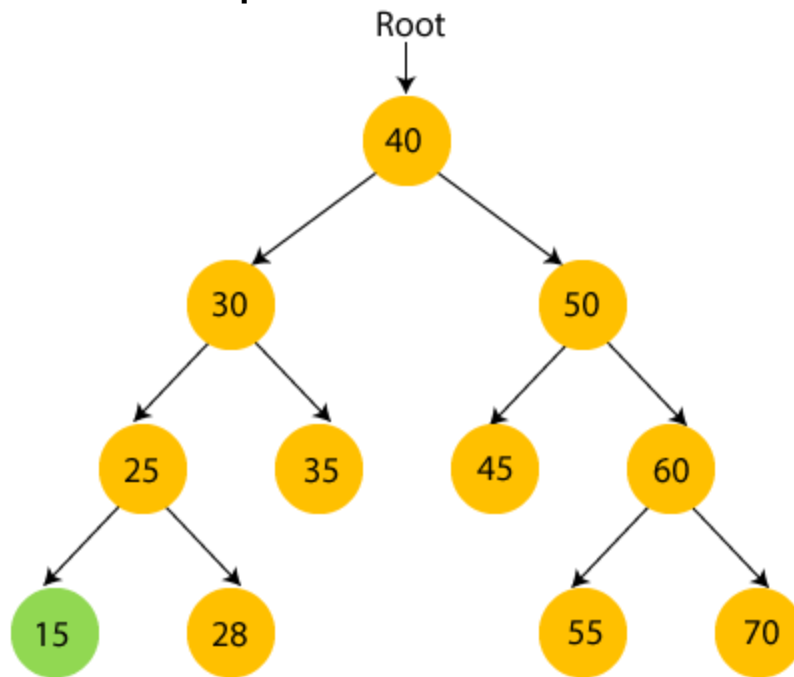
Now, let's see an example of inorder traversal. It will be easier to understand the procedure of inorder traversal using an example.



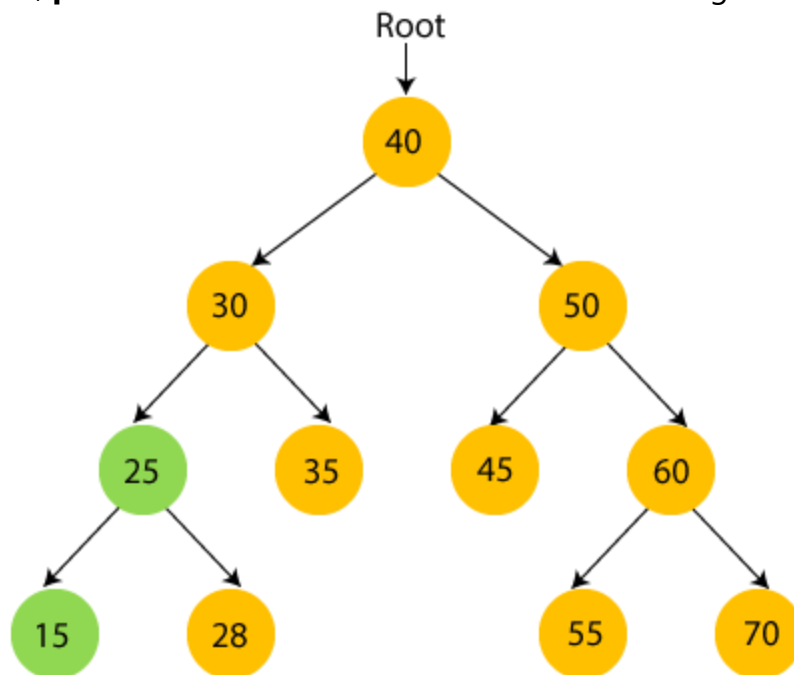
The nodes with yellow color are not visited yet. Now, we will traverse the nodes of the above tree using inorder traversal.

- Here, 40 is the root node. We move to the left subtree of 40, that is 30, and it also has subtree 25, so we again move to the left subtree of 25 that is 15. Here, 15 has

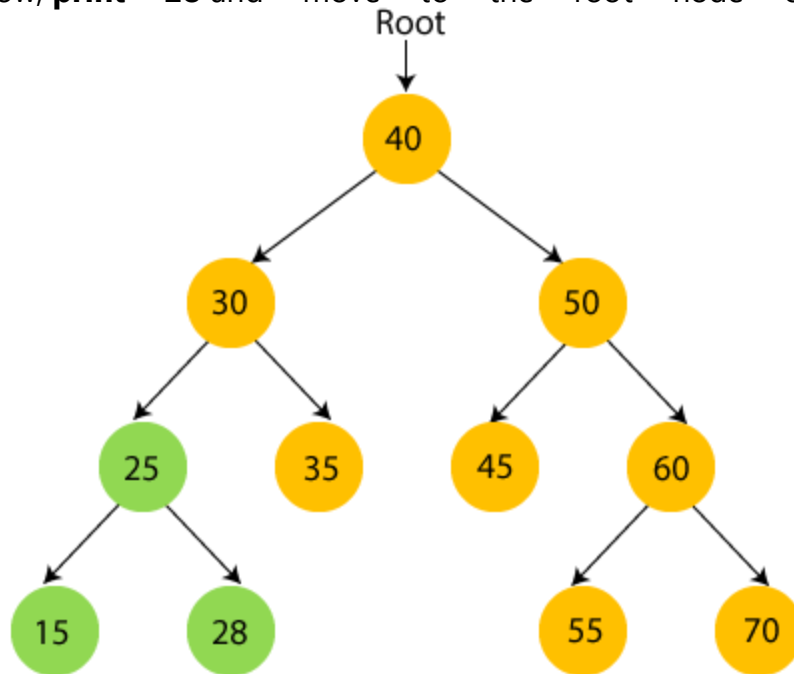
no subtree, so **print 15** and move towards its parent node, 25.



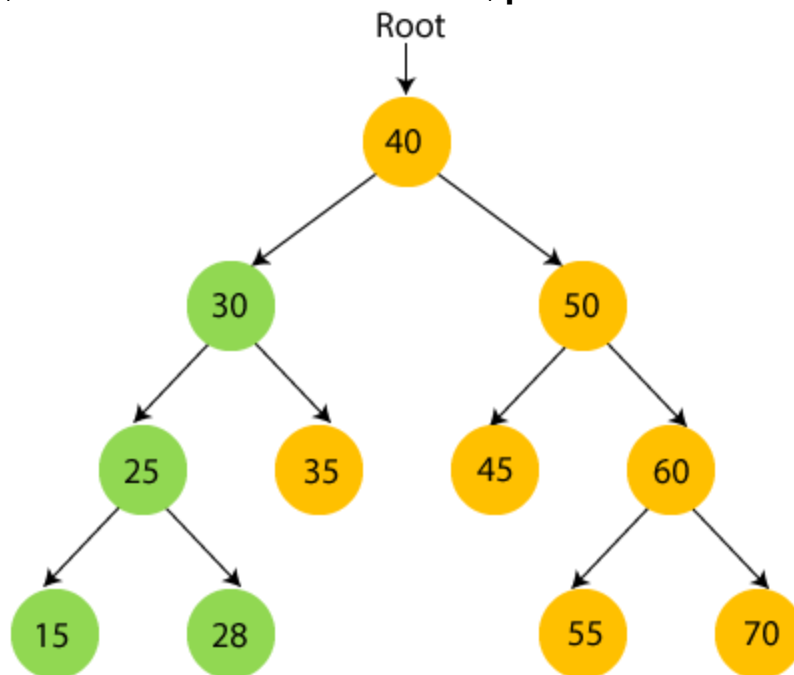
- Now, **print 25** and move to the right subtree of 25.



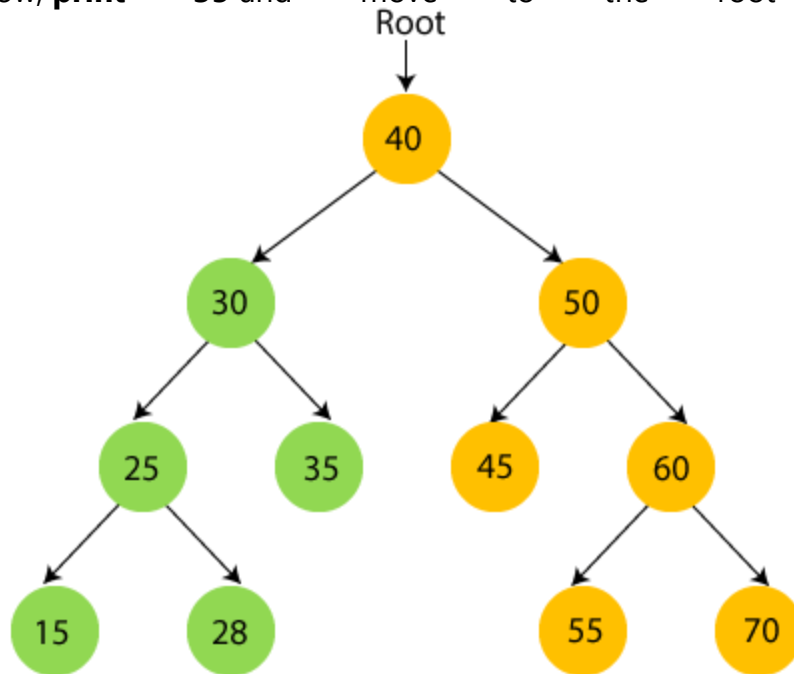
- Now, **print 28** and move to the root node of 25 that is 30.



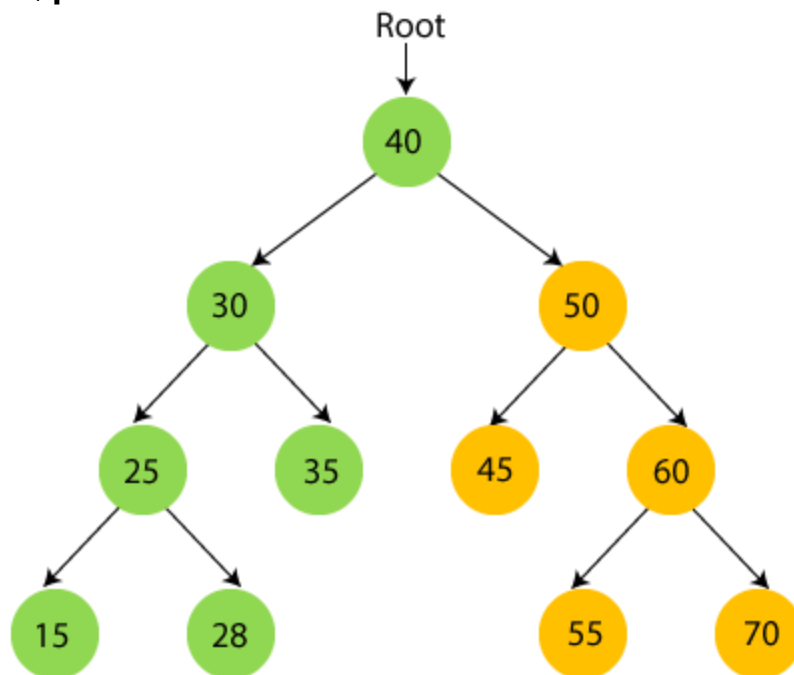
- So, left subtree of 30 is visited. Now, **print 30** and move to the right child of 30.



- Now, **print 35** and move to the root node of 30.

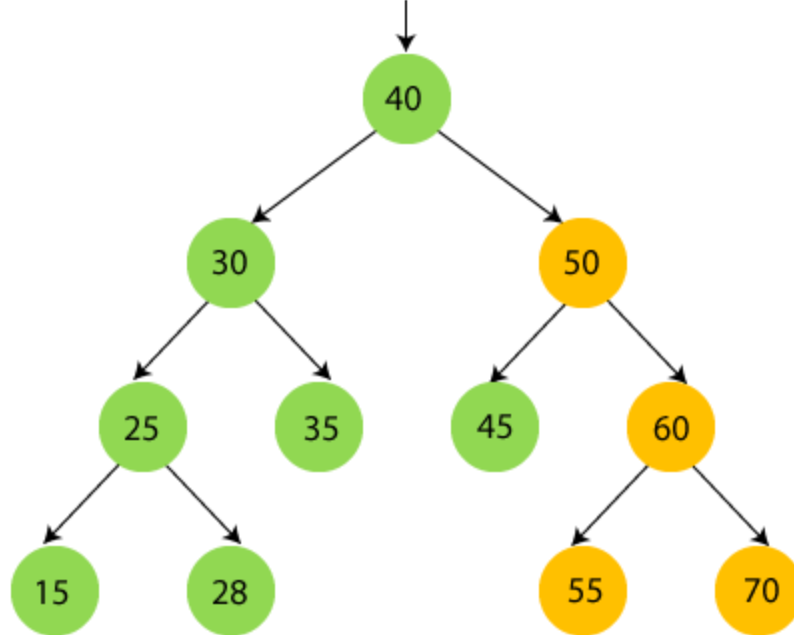


- Now, **print root node 40** and move to its right subtree.

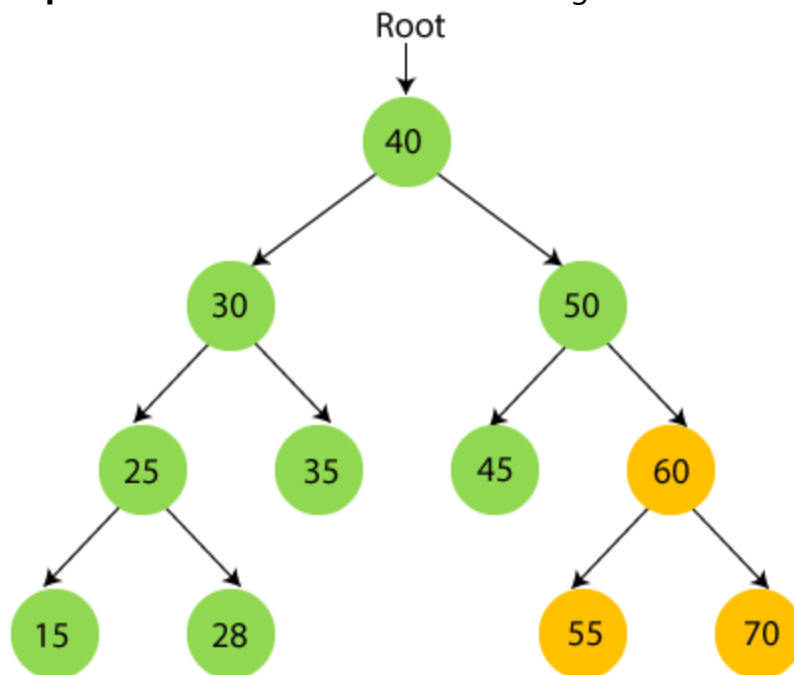


- Now recursively traverse the right subtree of 40 that is 50. 50 has subtree so first traverse the left subtree of 50 that is 45. 45 has no

children, so **print** **45** and move to its root node.

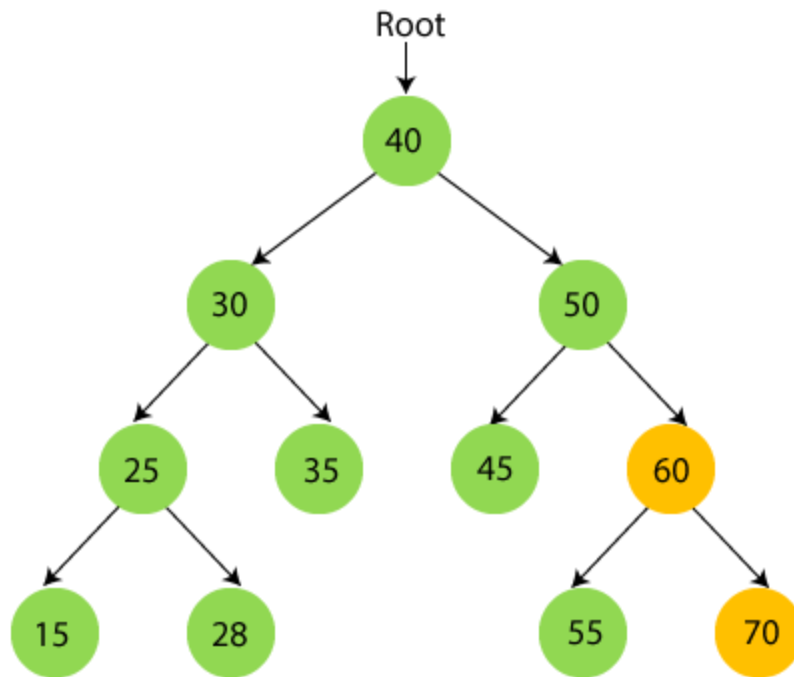


- Now **print** **50** and move to the right subtree of 50 that is 60.

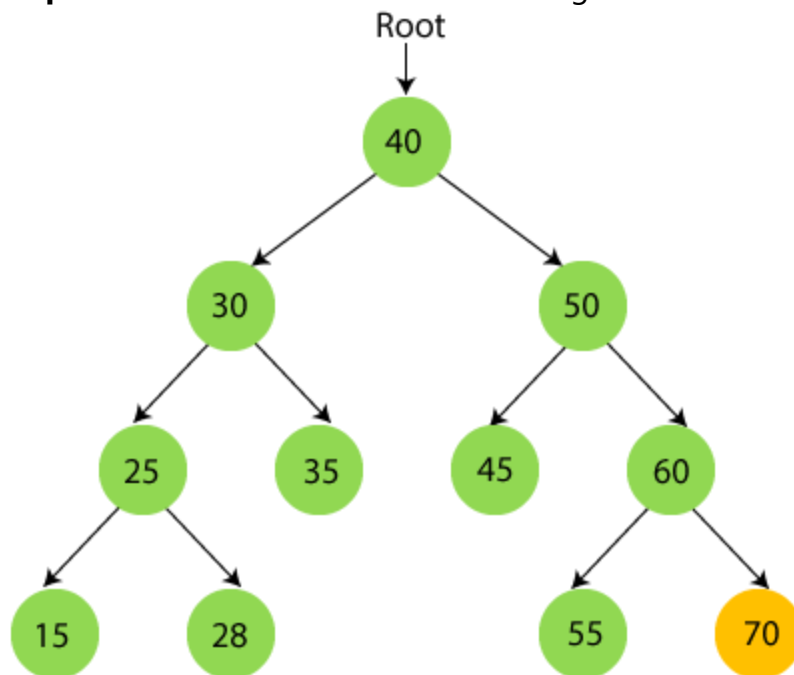


- Now recursively traverse the right subtree of 50 that is 60. 60 have subtree so first traverse the left subtree of 60 that is 55. 55 has no children, so **print** **55** and

move to its root node.

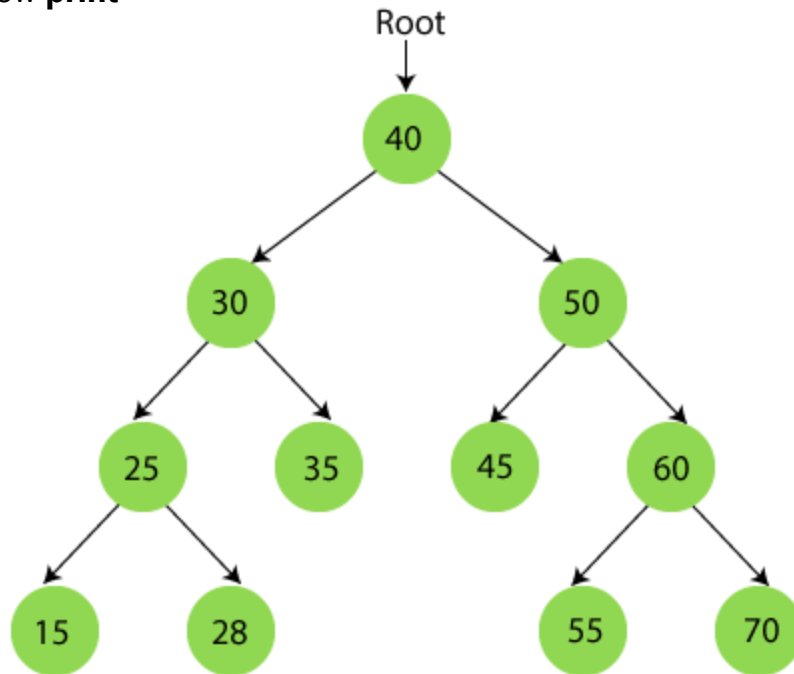


- Now **print 60** and move to the right subtree of 60 that is 70.



- Now **print**

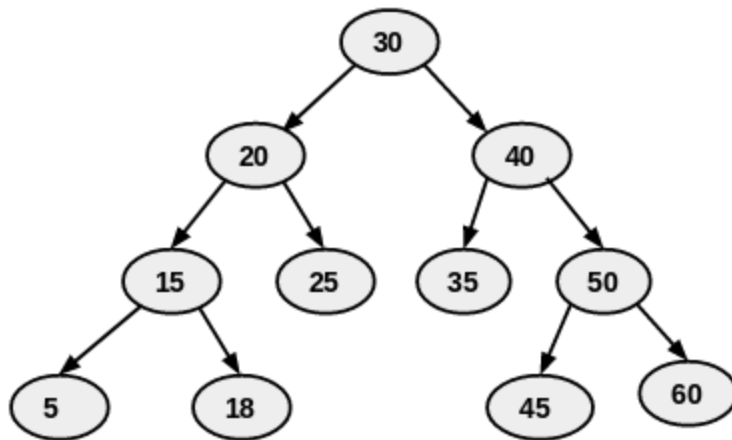
70.



After the completion of inorder traversal, the final output is -

{15, 25, 28, 30, 35, 40, 45, 50, 55, 60, 70}

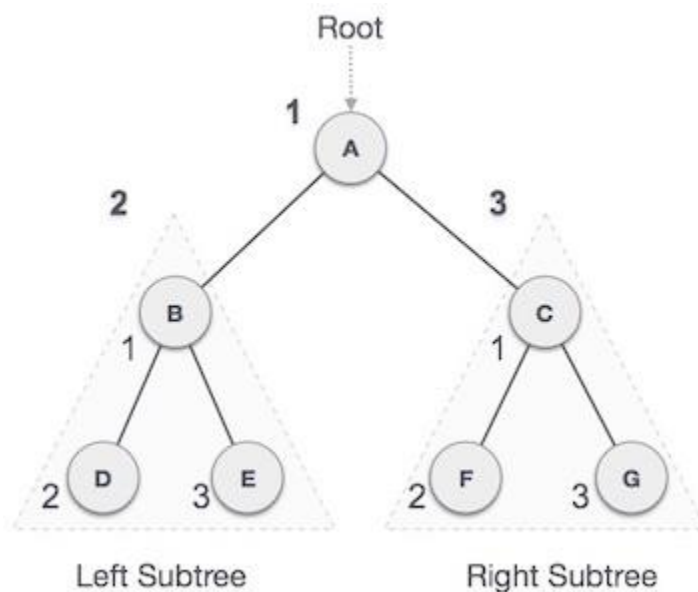
Example of inorder traversal



- we start recursive call from 30(root) then move to 20 (20 also have sub tree so apply in order on it),15 and 5.
- 5 have no child .so print 5 then move to it's parent node which is 15 print and then move to 15's right node which is 18.
- 18 have no child print 18 and move to 20 .print 20 then move it right node which is 25 .25 have no subtree so print 25.
- print root node 30 .
- now recursively traverse to right subtree of root node . so move to 40. 40 have subtree so traverse to left subtree of 40.
- left subtree of 40 have only one node which is 35. 35 had no further subtree so print 35. move to 40 and print 40.
- traverse to right subtree of 40. so move to 50 now have subtree so traverse to left subtree of 50 .move to 45 , 45 have no further subtree so print 45.
- move to 50 and print 50. now traverse to right subtree of 50 hence move to 60 and print 60.
- **our final output is {5 , 15 , 18 , 20 , 25 , 30 , 35 , 40 , 45 , 50 , 60}**

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 –

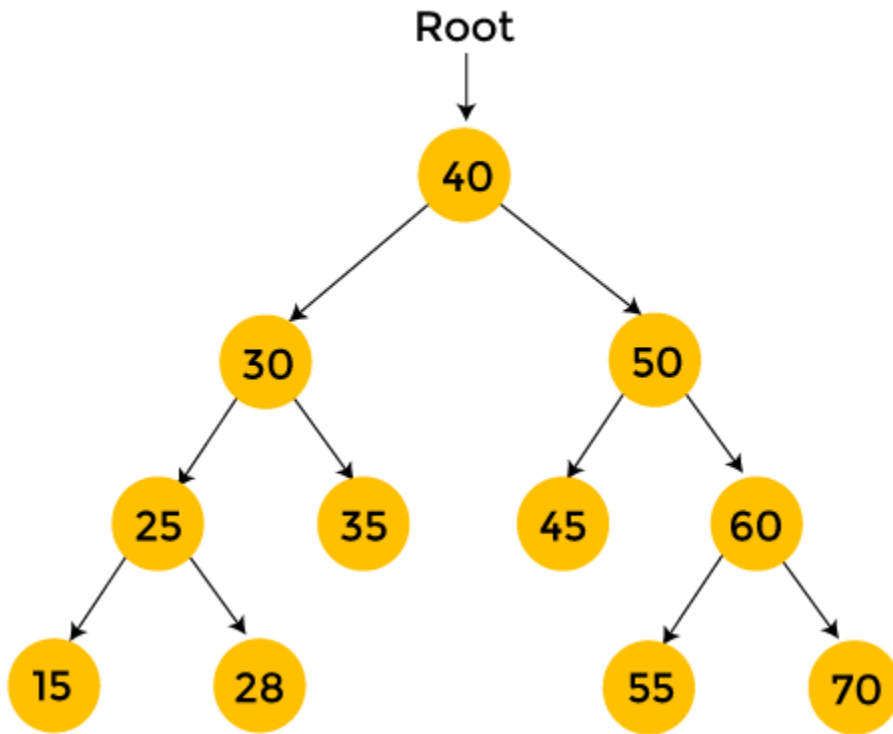
Step 1 – Visit root node.

Step 2 – Recursively traverse left subtree.

Step 3 – Recursively traverse right subtree.

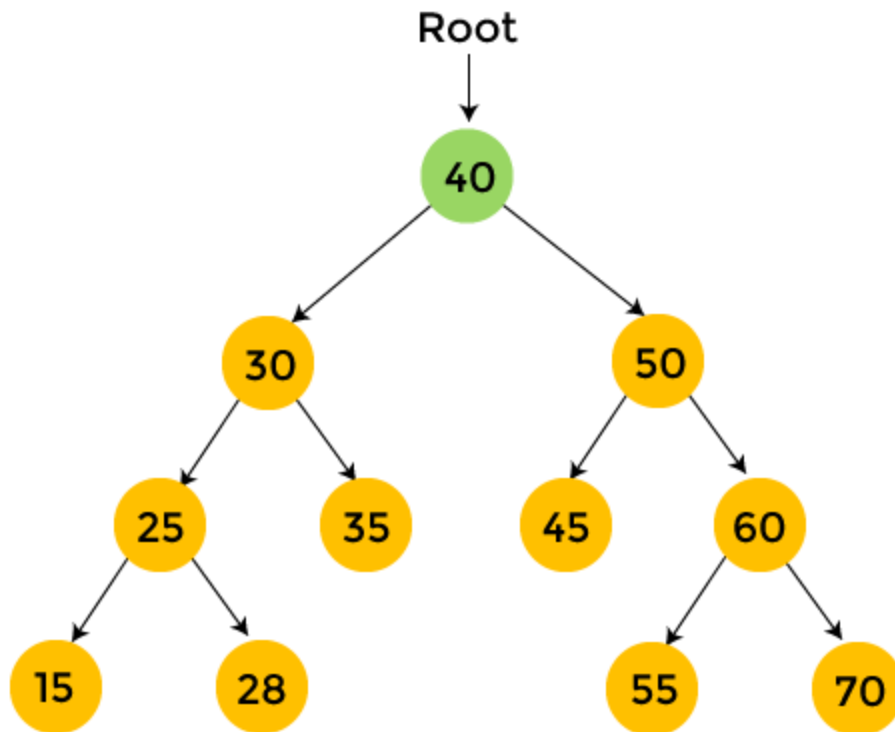
Example of preorder traversal

Now, let's see an example of preorder traversal. It will be easier to understand the process of preorder traversal using an example.

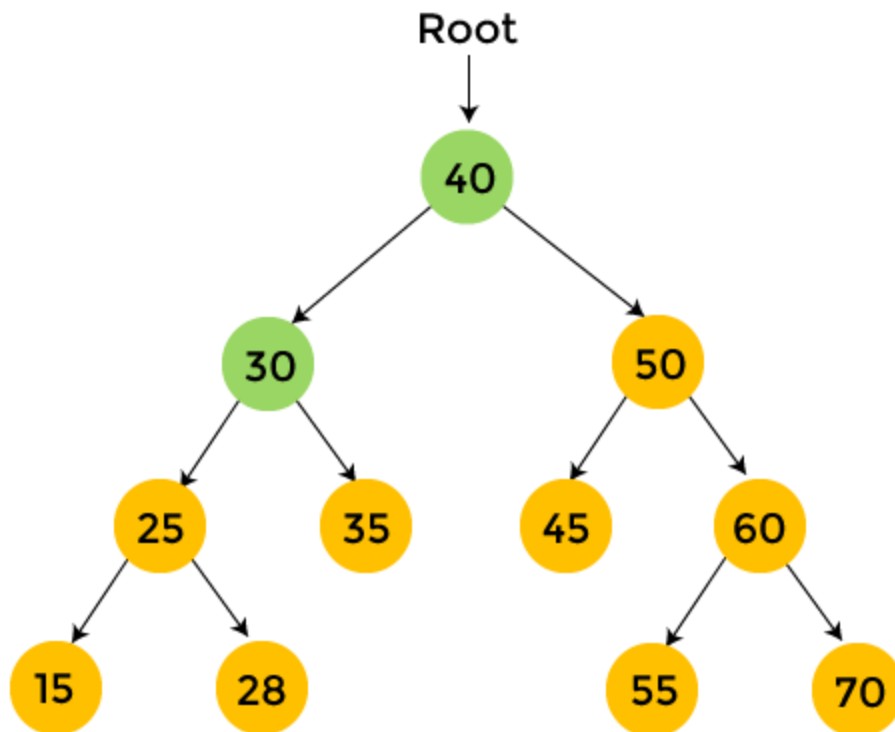


The nodes with yellow color are not visited yet. Now, we will traverse the nodes of the above tree using preorder traversal.

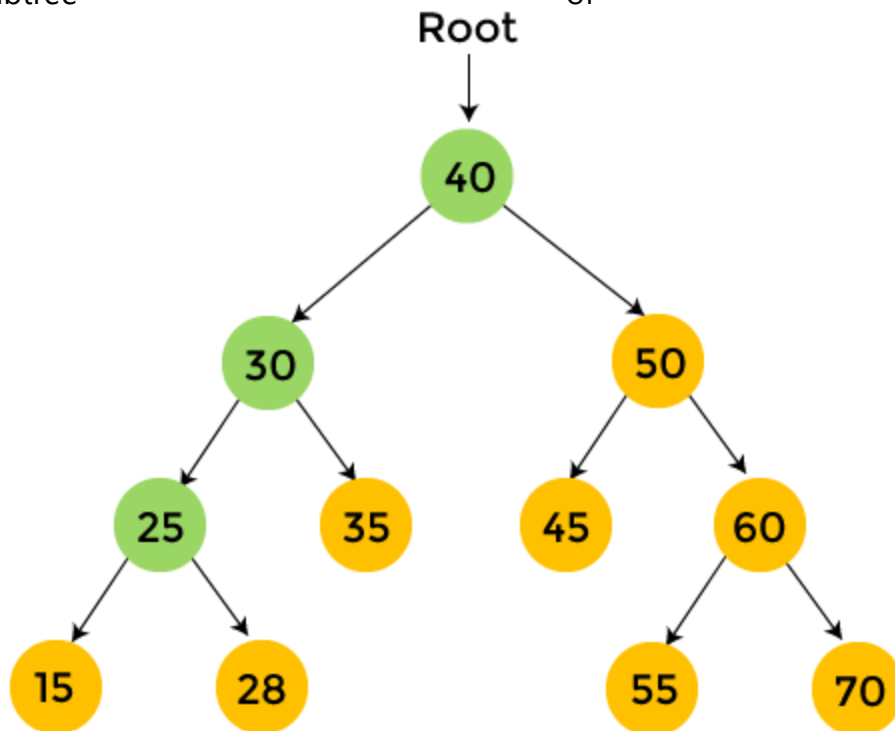
- Start with the root node 40. First, **print 40** and then recursively traverse the left subtree.



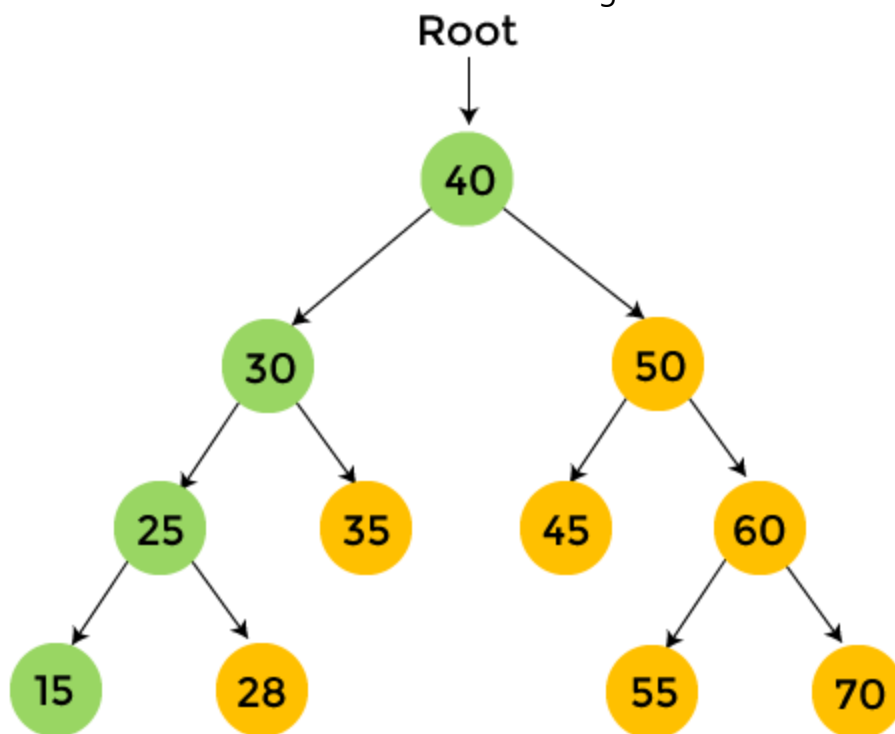
- Now, move to the left subtree. For left subtree, the root node is 30. **Print 30**, and move towards the left subtree of 30.



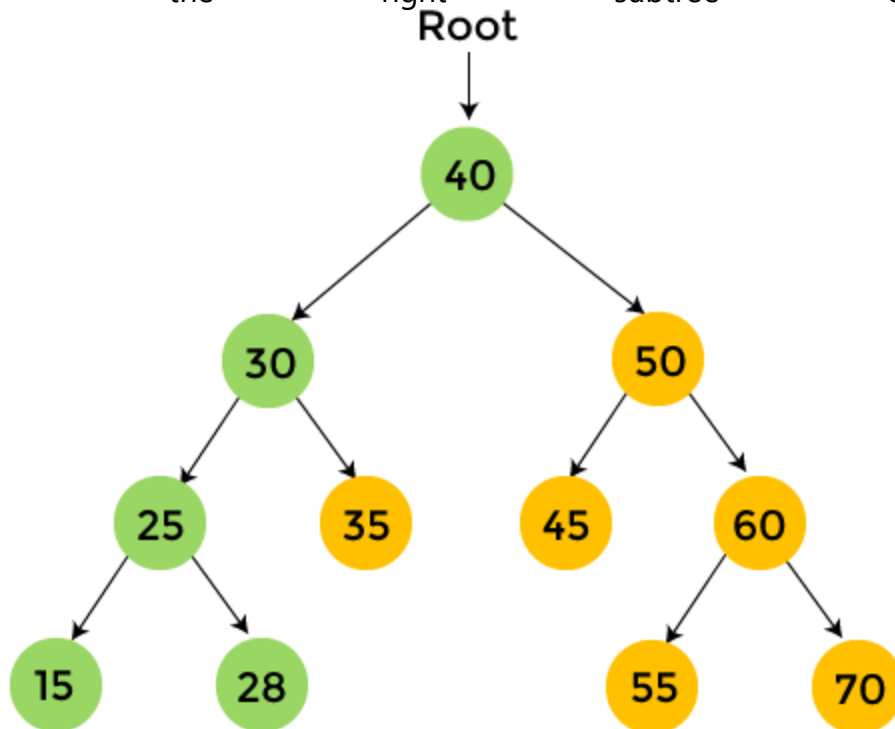
- In left subtree of 30, there is an element 25, so **print 25**, and traverse the left subtree of 25.



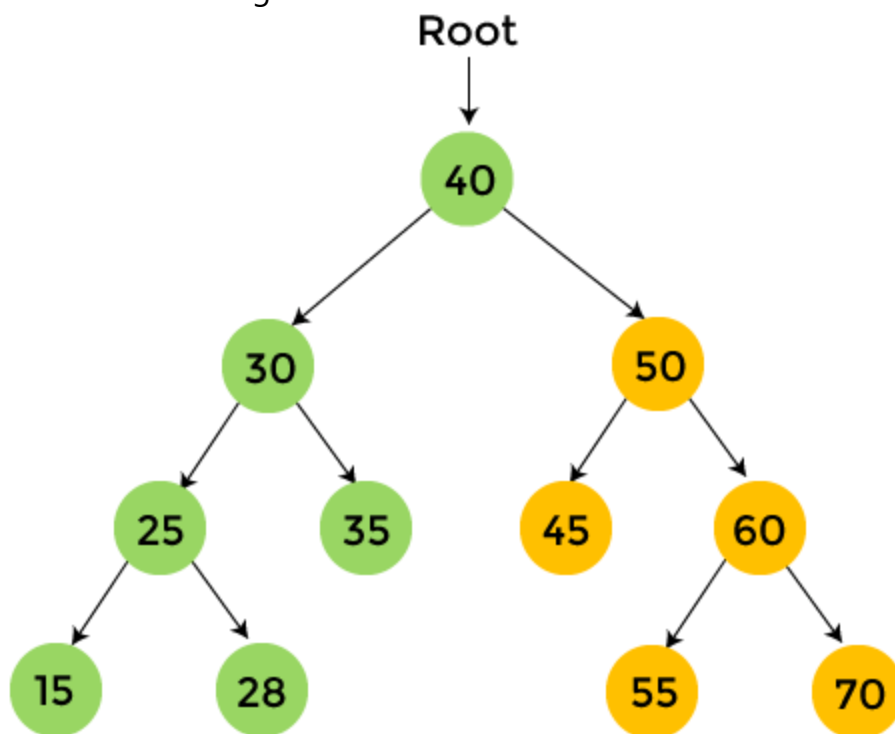
- In left subtree of 25, there is an element 15, and 15 has no subtree. So, **print 15**, and move to the right subtree of 25.



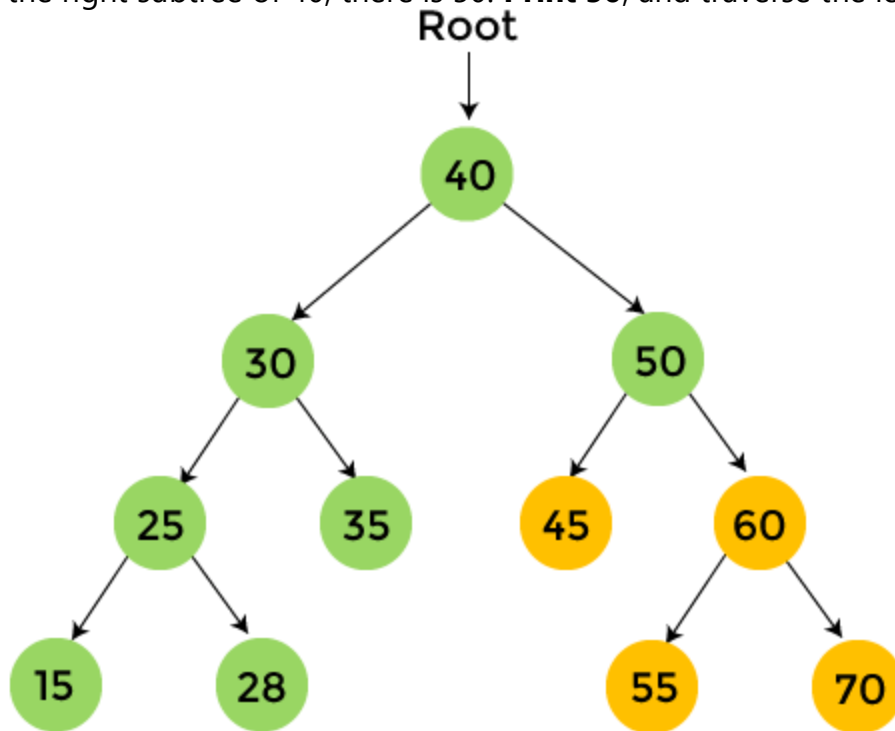
- In right subtree of 25, there is 28, and 28 has no subtree. So, **print 28**, and move to the right subtree of 30.



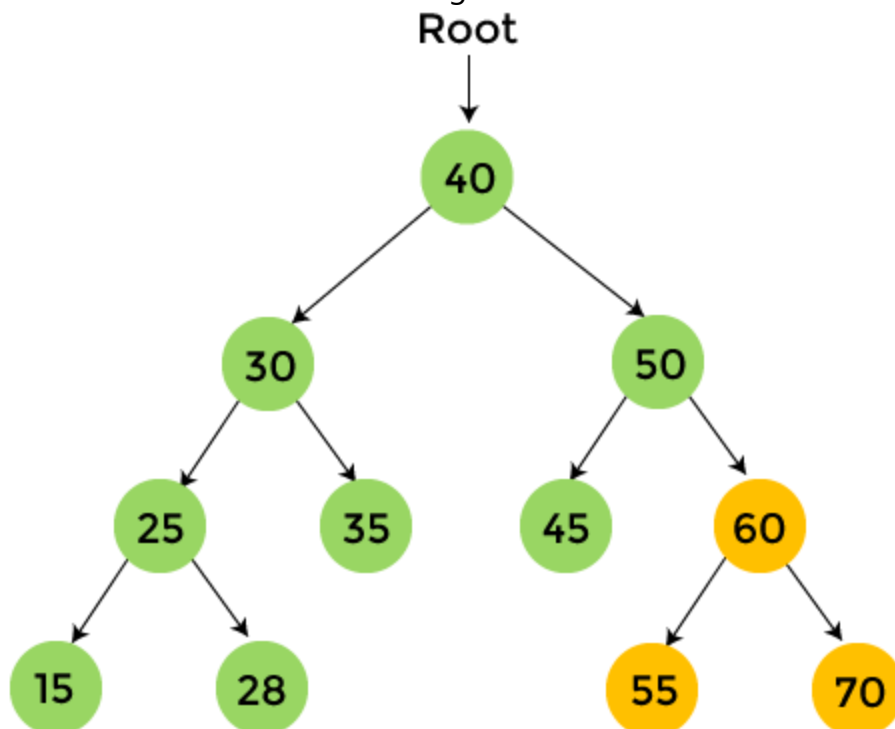
- In right subtree of 30, there is 35 that has no subtree. So **print 35**, and traverse the right subtree of 40.



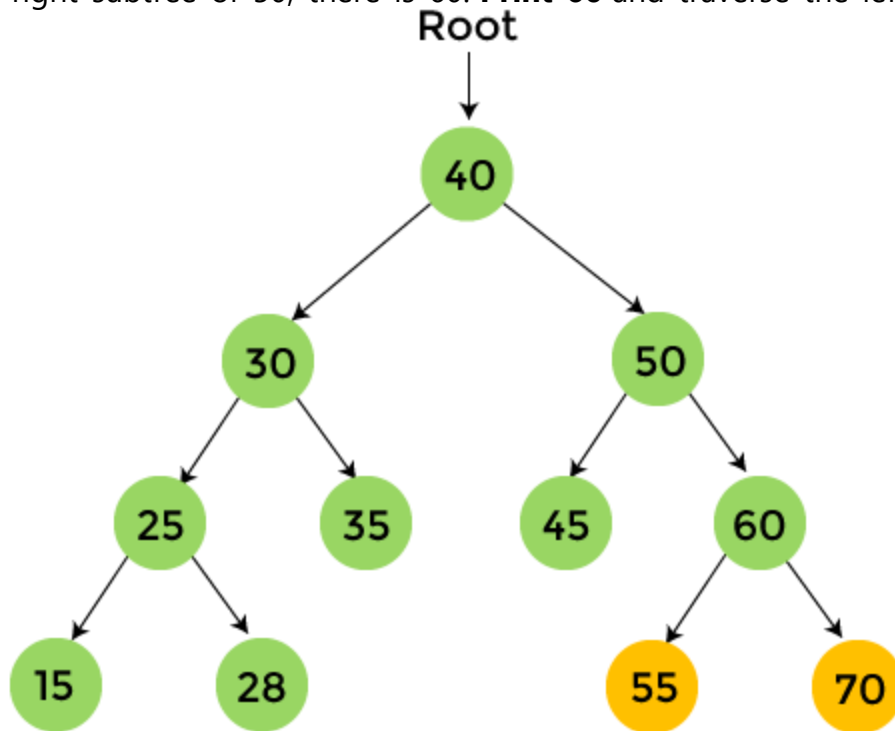
- In the right subtree of 40, there is 50. **Print 50**, and traverse the left subtree of 50.



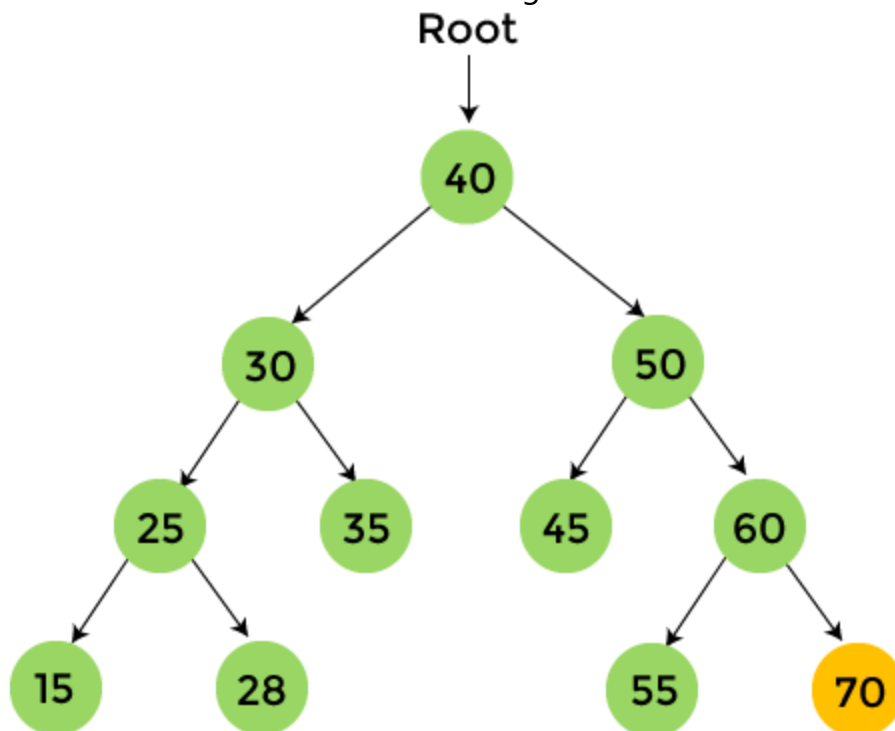
- In the left subtree of 50, there is 45 that do not have any child. So, **print 45**, and traverse the right subtree of 50.



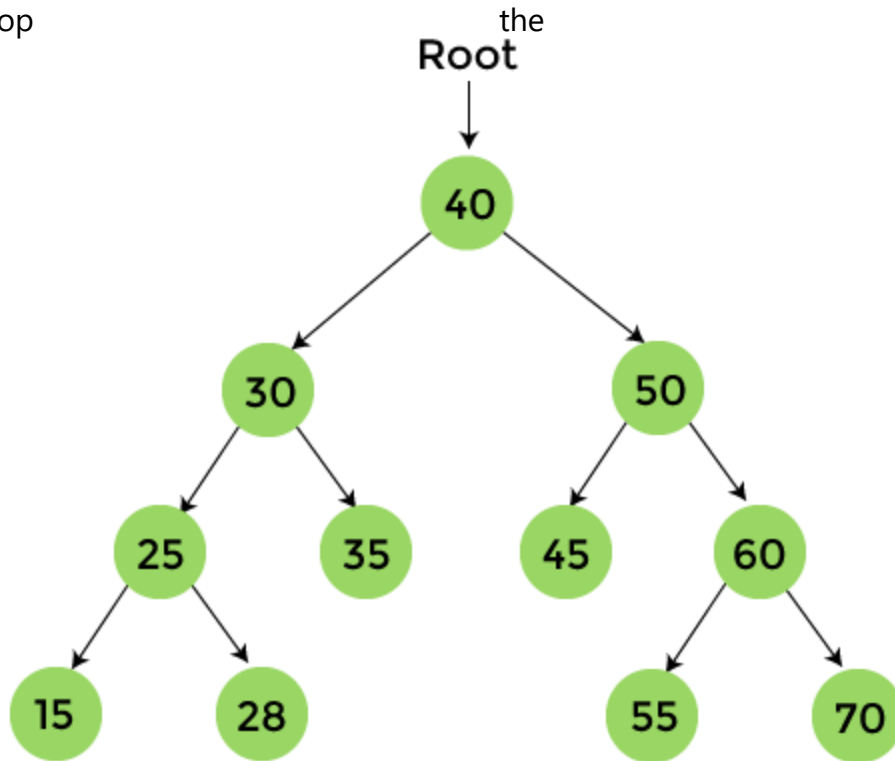
- In right subtree of 50, there is 60. **Print 60** and traverse the left subtree of 60.



- In the left subtree of 60, there is 55 that does not have any child. So, **print 55** and move to the right subtree of 60.



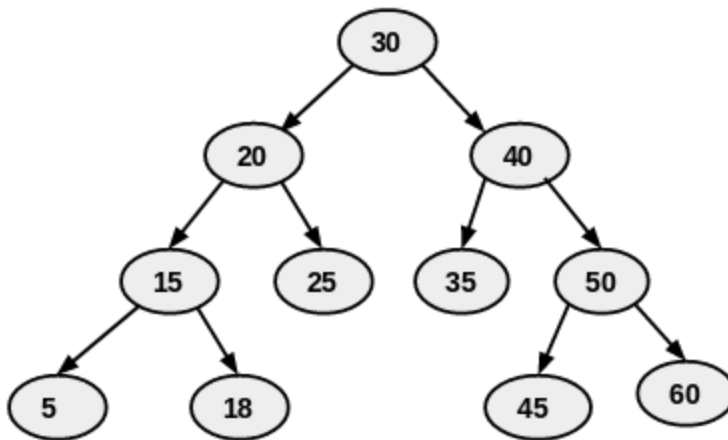
- In the right subtree of 60, there is 70 that do not have any child. So, **print 70** and stop the process.



After the completion of preorder traversal, the final output is -

40, 30, 25, 15, 28, 35, 50, 45, 60, 55, 70

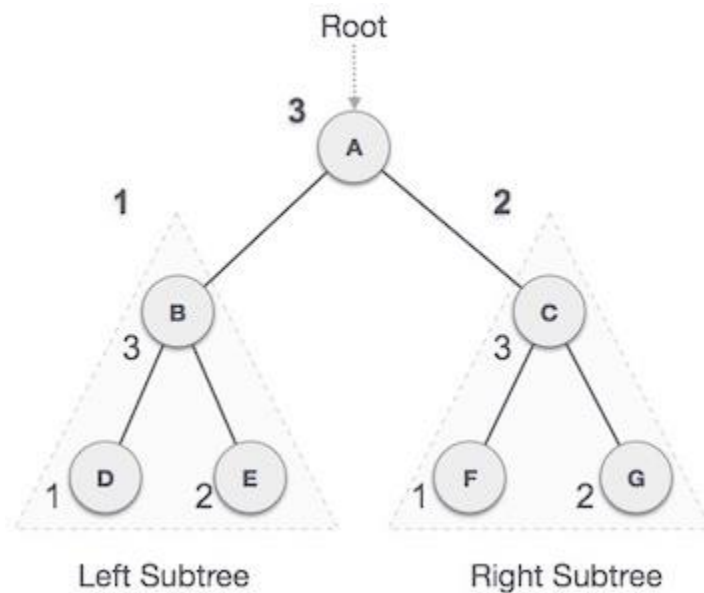
Example of preorder traversal



- Start with root node 30 .print 30 and recursively traverse the left subtree.
- next node is 20. now 20 have subtree so print 20 and traverse to left subtree of 20 .
- next node is 15 and 15 have subtree so print 15 and traverse to left subtree of 15.
- 5 is next node and 5 have no subtree so print 5 and traverse to right subtree of 15.
- next node is 18 and 18 have no child so print 18 and traverse to right subtree of 20.
- 25 is right subtree of 20 .25 have no child so print 25 and start traverse to right subtree of 30.
- next node is 40. node 40 have subtree so print 40 and then traverse to left subtree of 40.
- next node is 35. 35 have no subtree so print 35 and then traverse to right subtree of 40.
- next node is 50. 50 have subtree so print 50 and traverse to left subtree of 50.
- next node is 45. 45 have no subtree so print 45 and then print 60(right subtree) of 50.
- **our final output is {30 , 20 , 15 , 5 , 18 , 25 , 40 , 35 , 50 , 45 , 60}**

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 Post-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 –

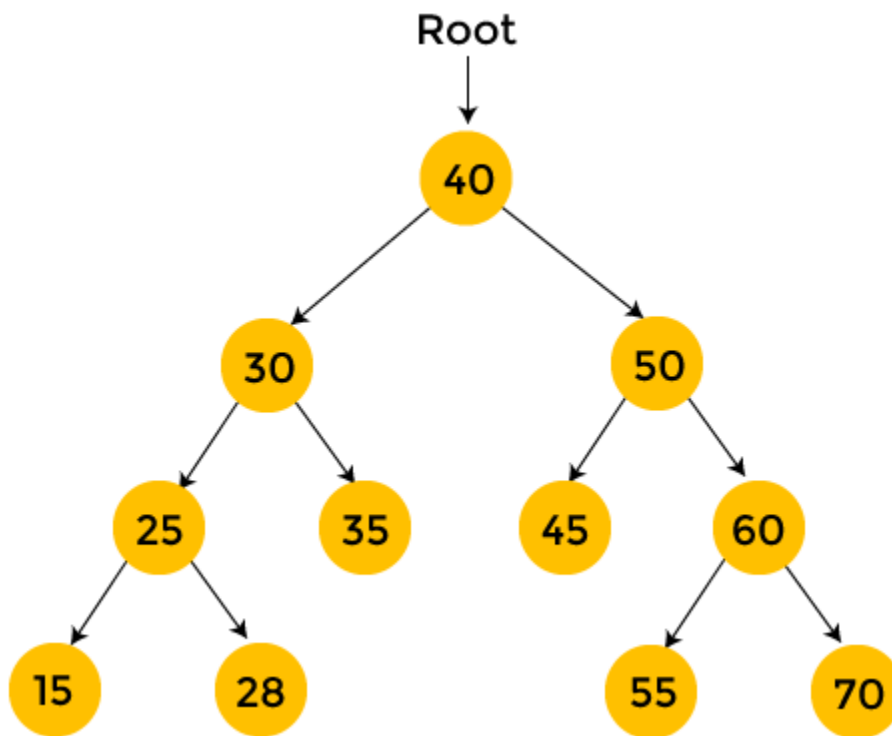
Step 1 – Recursively traverse left subtree.

Step 2 – Recursively traverse right subtree.

Step 3 – Visit root node.

Example of postorder traversal

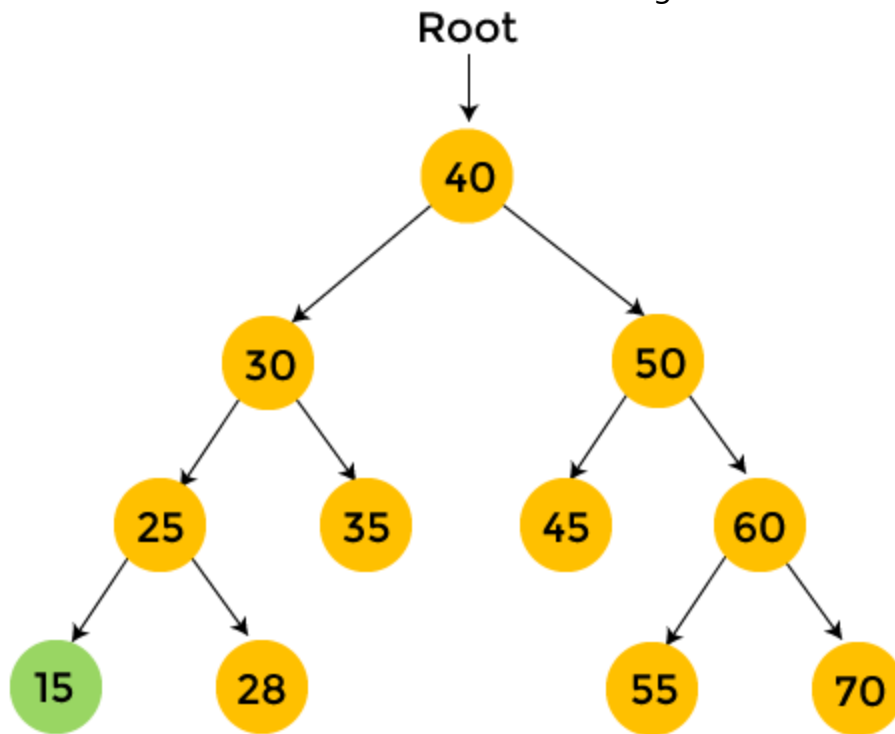
Now, let's see an example of postorder traversal. It will be easier to understand the process of postorder traversal using an example.



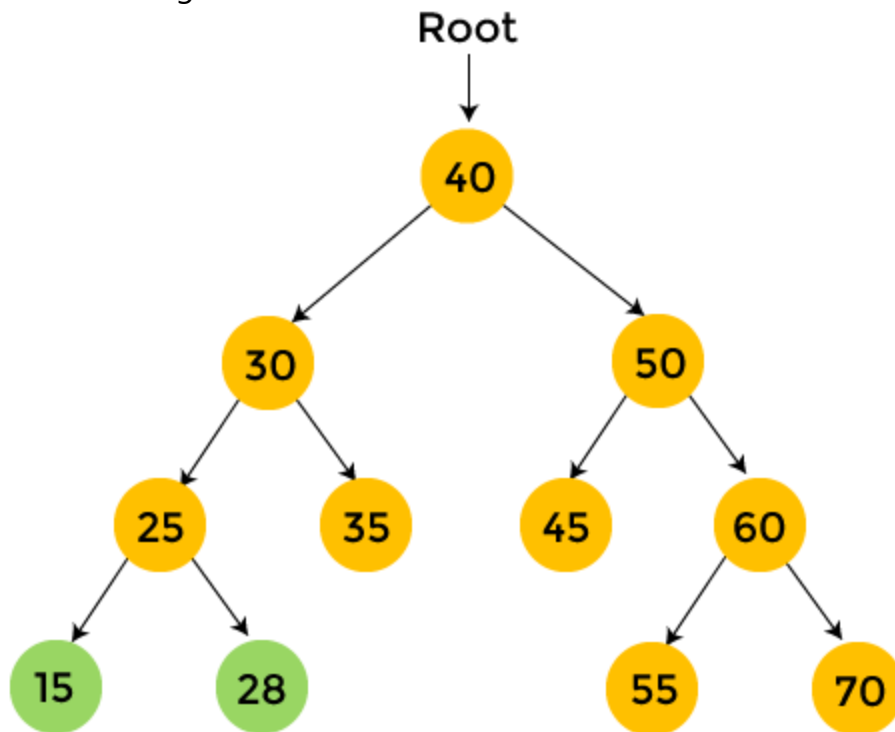
The nodes with yellow color are not visited yet. Now, we will traverse the nodes of above tree using postorder traversal.

- Here, 40 is the root node. We first visit the left subtree of 40, i.e., 30. Node 30 will also traverse in post order. 25 is the left subtree of 30, so it is also traversed in post order. Then 15 is the left subtree of 25. But 15 has no subtree, so **print**

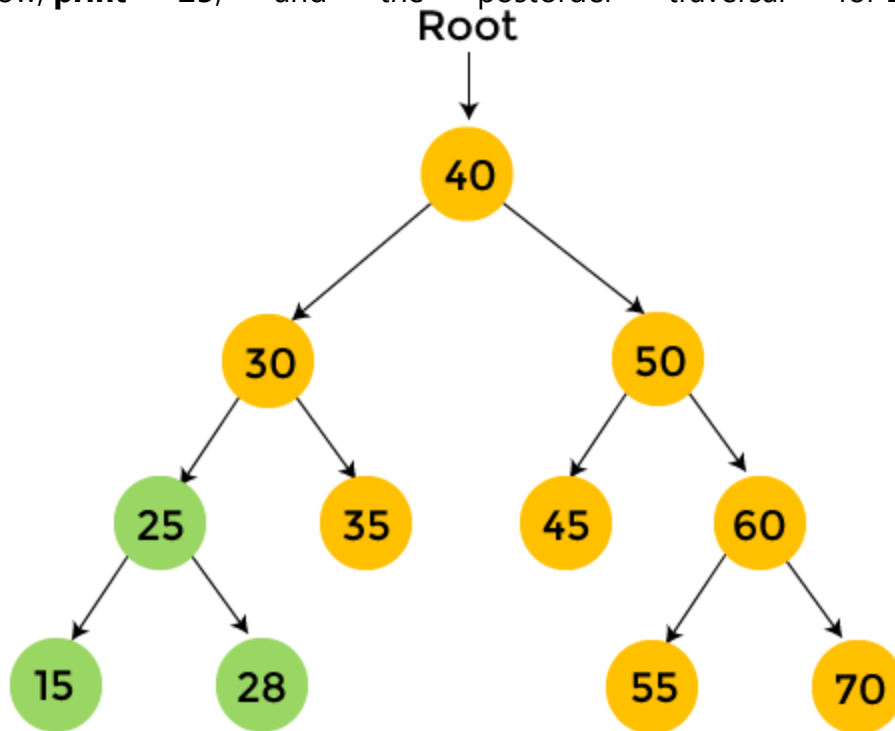
15 and move towards the right subtree of 25.



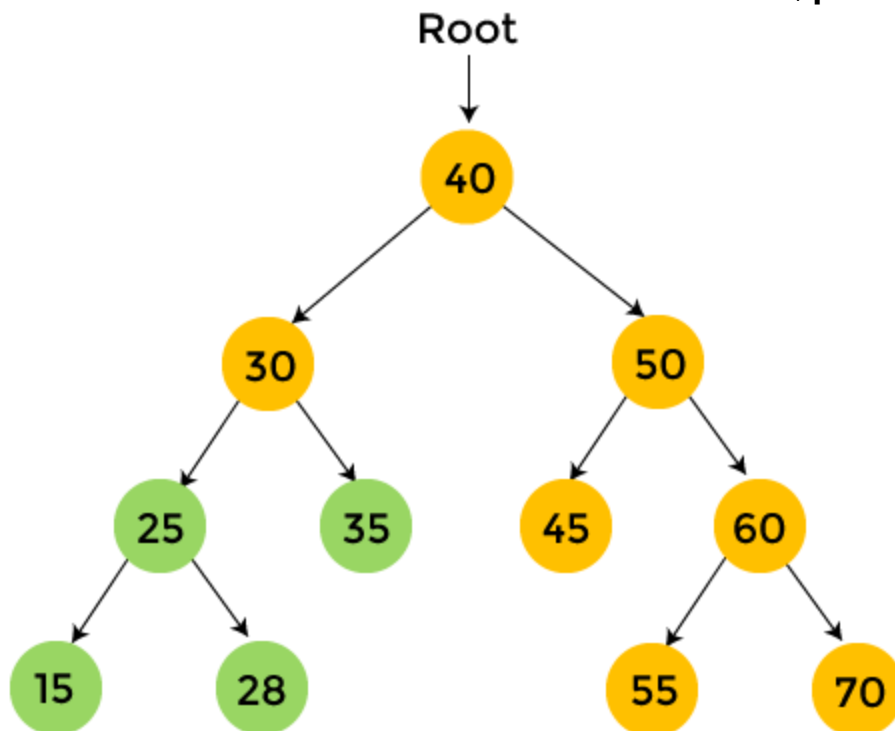
- 28 is the right subtree of 25, and it has no children. So, **print 28**.



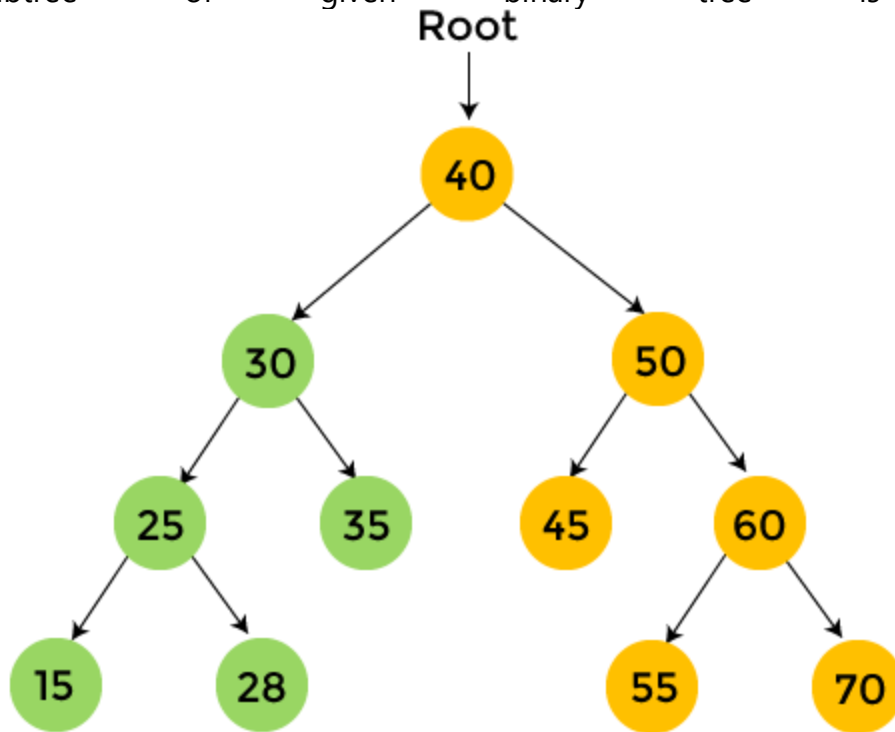
- Now, **print 25**, and the postorder traversal for **25** is finished.



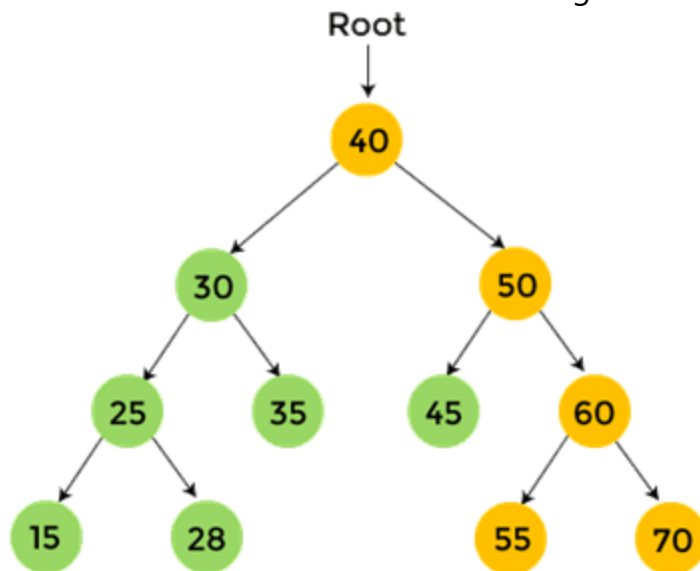
- Next, move towards the right subtree of 30. 35 is the right subtree of 30, and it has no children. So, **print 35**.



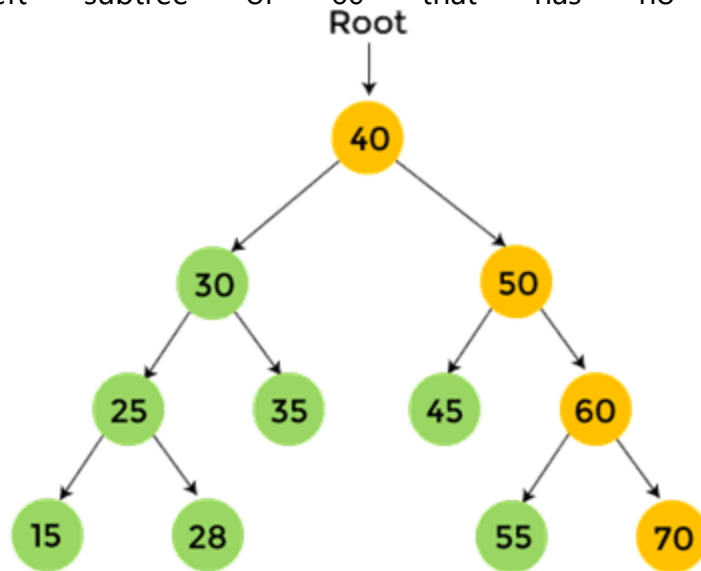
- After that, **print 30**, and the postorder traversal for **30** is finished. So, the left subtree of given binary tree is traversed.



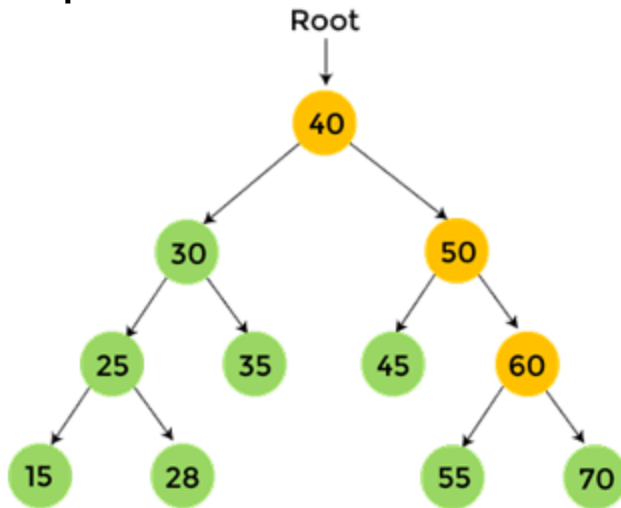
- Now, move towards the right subtree of 40 that is 50, and it will also traverse in post order. 45 is the left subtree of 50, and it has no children. So, **print 45** and move towards the right subtree of 50.



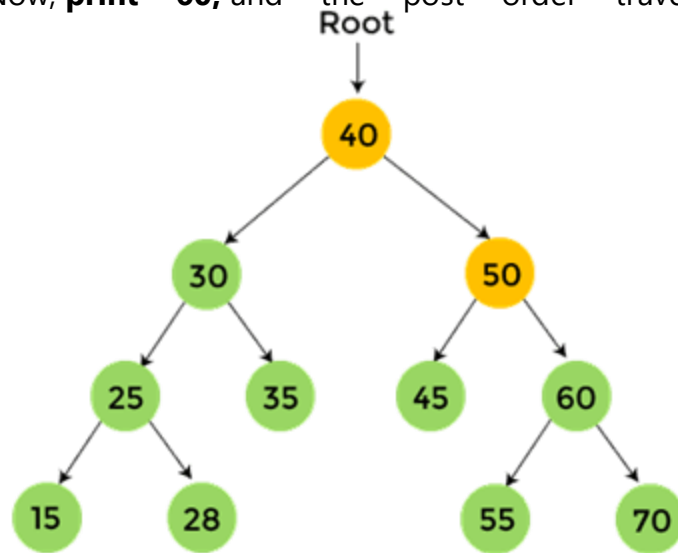
- 60 is the right subtree of 50, which will also be traversed in post order. 55 is the left subtree of 60 that has no children. So, **print 55**.



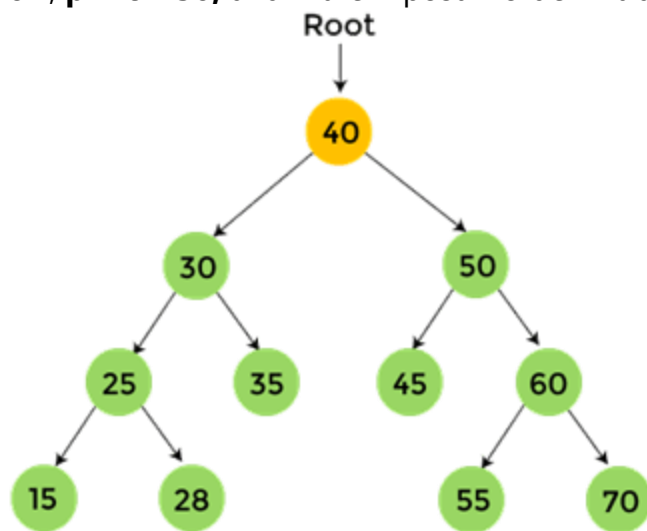
- Now, **print 70**, which is the right subtree of 60.



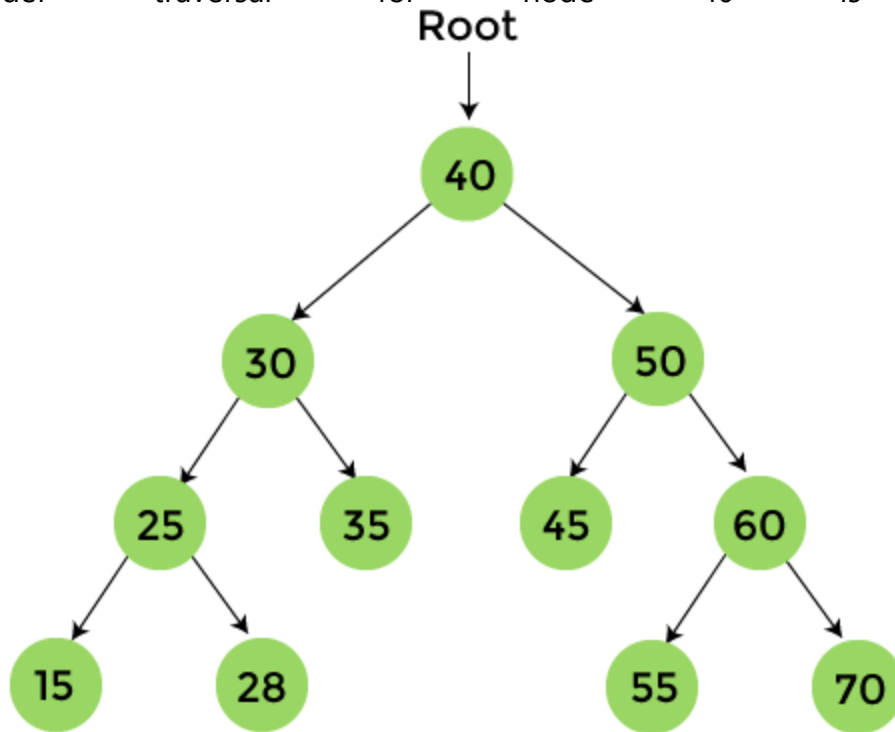
- Now, **print 60**, and the post order traversal for 60 is completed.



- Now, **print 50**, and the post order traversal for 50 is completed.



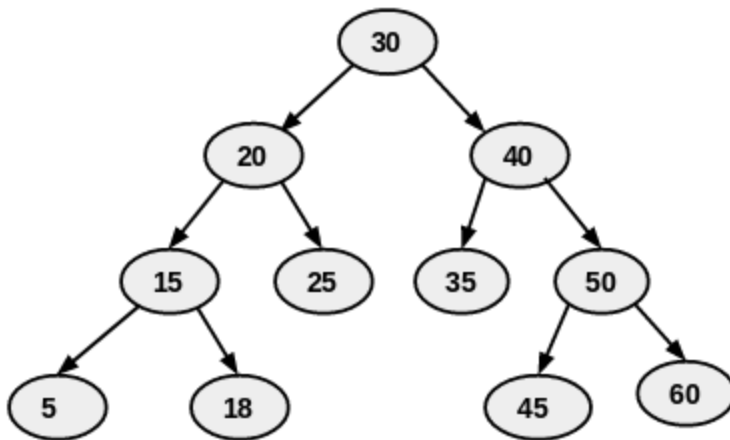
- At last, **print 40**, which is the root node of the given binary tree, and the post order traversal for node 40 is completed.



The final output that we will get after postorder traversal is -

{15, 28, 25, 35, 30, 45, 55, 70, 60, 50, 40}

Example of postorder traversal



- We start from 30, and following Post-order traversal, we first visit the left subtree 20. 20 is also traversed post-order.
- 15 is left subtree of 20 .15 is also traversed post order.
- 5 is left subtree of 15. 5 have no subtree so print 5 and traverse to right subtree of 15 .
- 18 is right subtree of 15. 18 have no subtree so print 18 and then print 15. post order traversal for 15 is finished.
- next move to right subtree of 20.
- 25 is right subtree of 20. 25 have no subtree so print 25 and then print 20. post order traversal for 20 is finished.
- next visit the right subtree of 30 which is 40 .40 is also traversed post-order(40 have subtree).
- 35 is left subtree of 40. 35 have no more subtree so print 35 and traverse to right subtree of 40.
- 50 is right subtree of 40. 50 should also traversed post order.
- 45 is left subtree of 50. 45 have no more subtree so print 45 and then print 60 which is right subtree of 50.
- next print 50 . post order traversal for 50 is finished.

- now print 40 ,and post order traversal for 40 is finished.
- print 30. post order traversal for 30 is finished.
- **our final output is {5 , 18 , 15 , 25 , 20 , 35 , 45 , 60 , 50 , 40 , 30}**

Binary tree vs Binary Search tree

First, we will understand the **binary tree** and **binary search tree** separately, and then we will look at the differences between a binary tree and a binary search tree.

What is a Binary tree?

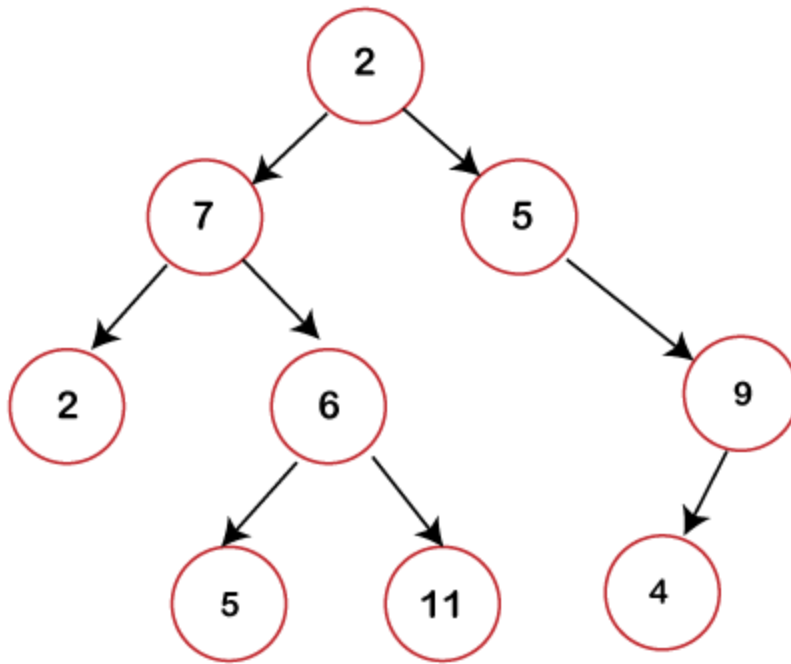
A Binary tree is a non-linear data structure in which a node can have either **0**, **1** or **maximum 2 nodes**. Each node in a binary tree is represented either as a parent node or a child node. There can be two children of the parent node, i.e., **left child** and **right child**.

There is only one way to reach from one node to its next node in a binary tree.

A node in a binary tree has three fields:

- **Pointer to the left child:** It stores the reference of the left-child node.
- **Pointer to the right child:** It stores the reference of the right-child node.
- **Data element:** The data element is the value of the data which is stored by the node.

The binary tree can be represented as:



In the above figure, we can observe that each node contains utmost 2 children. If any node does not contain left or right child then the value of the pointer with respect to that child would be NULL.

Basic terminologies used in a Binary tree are:

- **Root node:** The root node is the first or the topmost node in a binary tree.
- **Parent node:** When a node is connected to another node through edges, then that node is known as a parent node. In a binary tree, parent node can have a maximum of 2 children.
- **Child node:** If a node has its predecessor, then that node is known as a **child node**.
- **Leaf node:** The node which does not contain any child known as a **leaf node**.
- **Internal node:** The node that has atleast 2 children known as an **internal node**.
- **Depth of a node:** The distance from the root node to the given node is known as a **depth of a node**. We provide labels to all the nodes like root node is labeled with 0 as it has no depth, children of the root nodes are labeled with 1, children of the root child are labeled with 2.

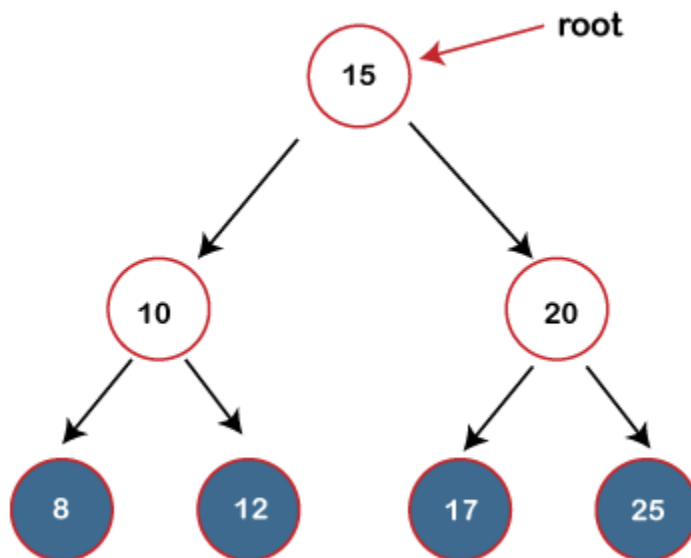
- **Height:** The longest distance from the root node to the leaf node is the **height of the node**.

In a binary tree, there is one tree known as a **perfect binary tree**. It is a [tree](#) in which all the internal nodes must contain two nodes, and all the leaf nodes must be at the same depth.

What is a Binary Search tree?

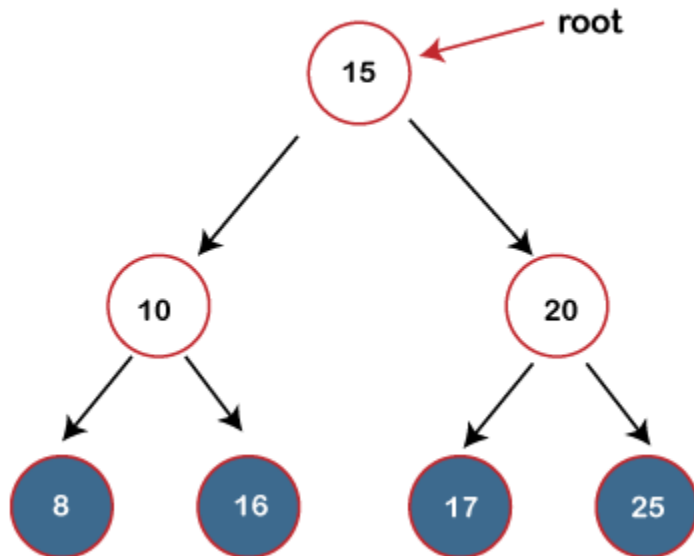
A [Binary search tree](#) is a tree that follows some order to arrange the elements, whereas the binary tree does not follow any order. In a Binary search tree, the value of the left node must be smaller than the parent node, and the value of the right node must be greater than the parent node.

Let's understand the concept of a binary search tree through examples.



In the above figure, we can observe that the value of the root node is 15, which is greater than the value of all the nodes in the left subtree. The value of root node is less than the values of all the nodes in a right-subtree. Now, we move to the left-child of the root node. 10 is greater than 8 and lesser than 12; it also satisfies the property of the Binary search tree. Now, we move to the right-child of the root node; the value 20 is greater than 17 and lesser than 25; it also satisfies the property of binary search tree. Therefore, we can say that the tree shown above is the binary search tree.

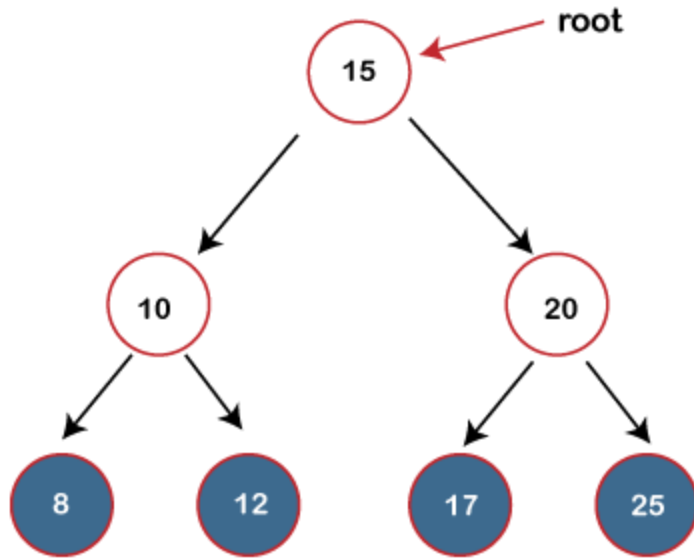
Now, if we change the value of 12 to 16 in the above binary tree, we have to find whether it is still a binary search tree or not.



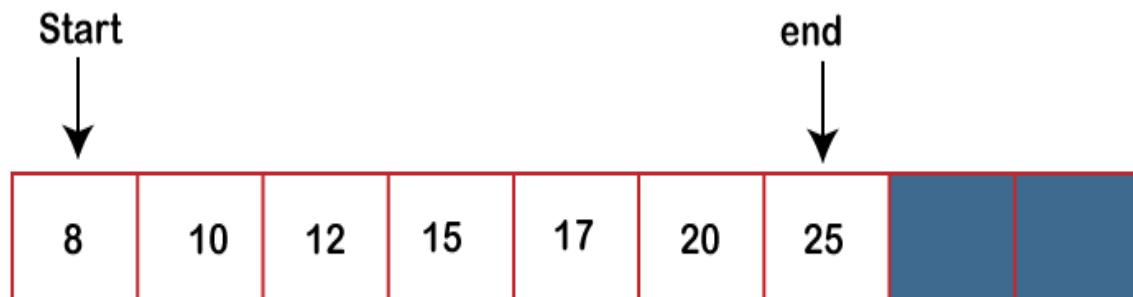
The value of the root node is 15 which is greater than 10 but lesser than 16, so it does not satisfy the property of the Binary search tree. Therefore, it is not a binary search tree.

Operations on Binary search tree

We can perform insert, delete and search operations on the binary search tree. Let's understand how a search is performed on a binary search. The binary tree is shown below on which we have to perform the search operation:



Suppose we have to search 10 in the above binary tree. To perform the binary search, we will consider all the integers in a sorted array. First, we create a complete list in a search space, and all the numbers will exist in the search space. The search space is marked by two pointers, i.e., start and end. The array of the above binary tree can be represented as



First, we will calculate the middle element and compare the middle element with the element, which is to be searched. The middle element is calculated by using $n/2$. The value of n is 7; therefore, the middle element is 15. The middle element is not equal to the searched element, i.e., 10.

Note: If the element is being searched is lesser than the mid element, then the searching will be performed in the left half; else, searching will be done on the right half. In the case of equality, the element is found.

As the element to be searched is lesser than the mid element, so searching will be performed on the left array. Now the search is reduced to half, as shown below:



The mid element in the left array is 10, which is equal to the searched element.

Types of Binary Trees

There are various **types of binary trees**, and each of these **binary tree types** has unique characteristics. Here are each of the **binary tree types** in detail:

Types of Binary Tree based on the number of children:

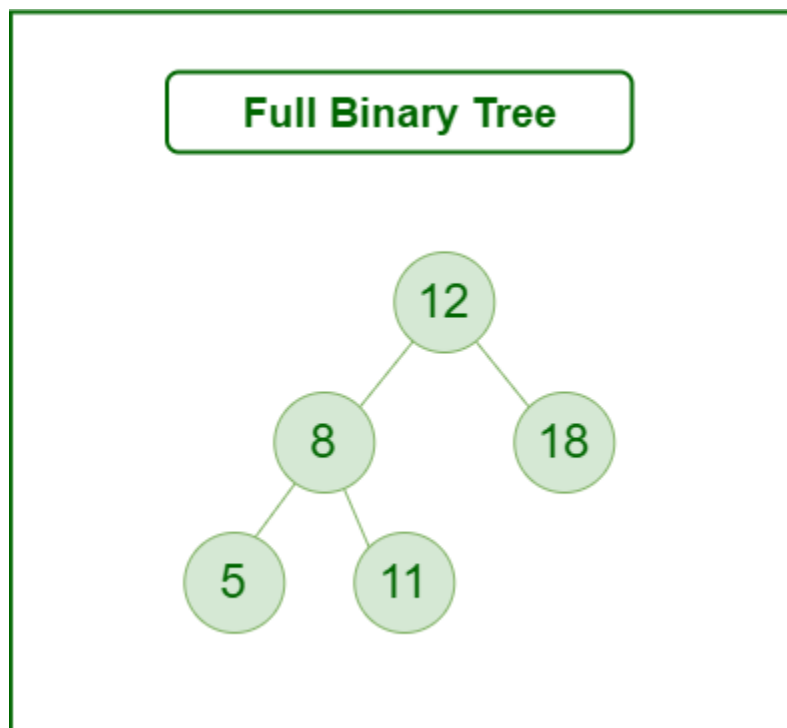
Following are the types of Binary Tree based on the number of children:

1. Full Binary Tree
2. Degenerate Binary Tree
3. Skewed Binary Trees

1. Full Binary Tree

A Binary Tree is a full binary tree if every node has 0 or 2 children. The following are examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.

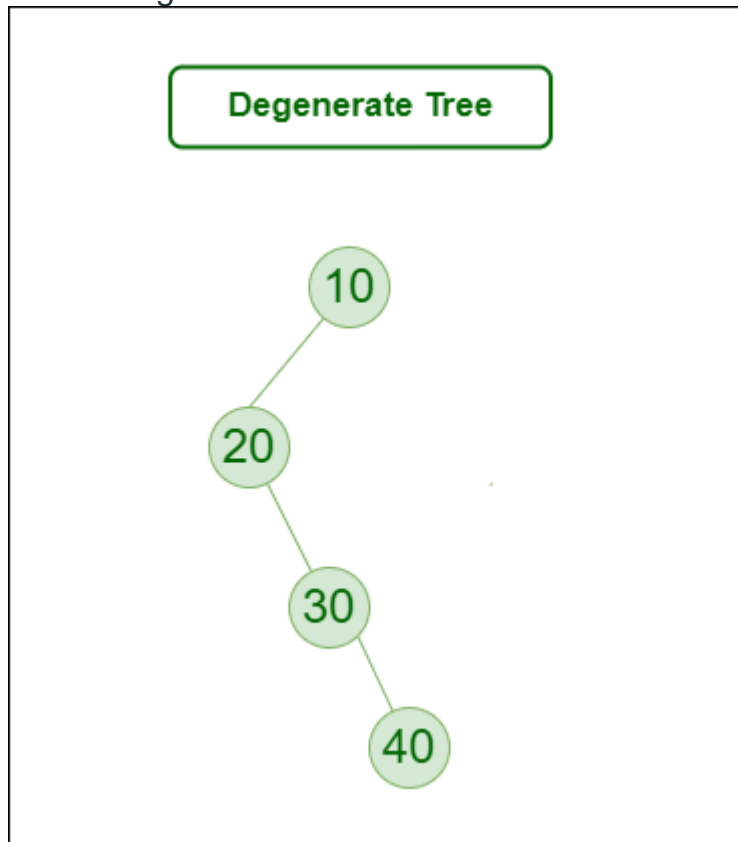
A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children. It is also known as a proper binary tree.



Full Binary Tree

2. Degenerate (or pathological) tree

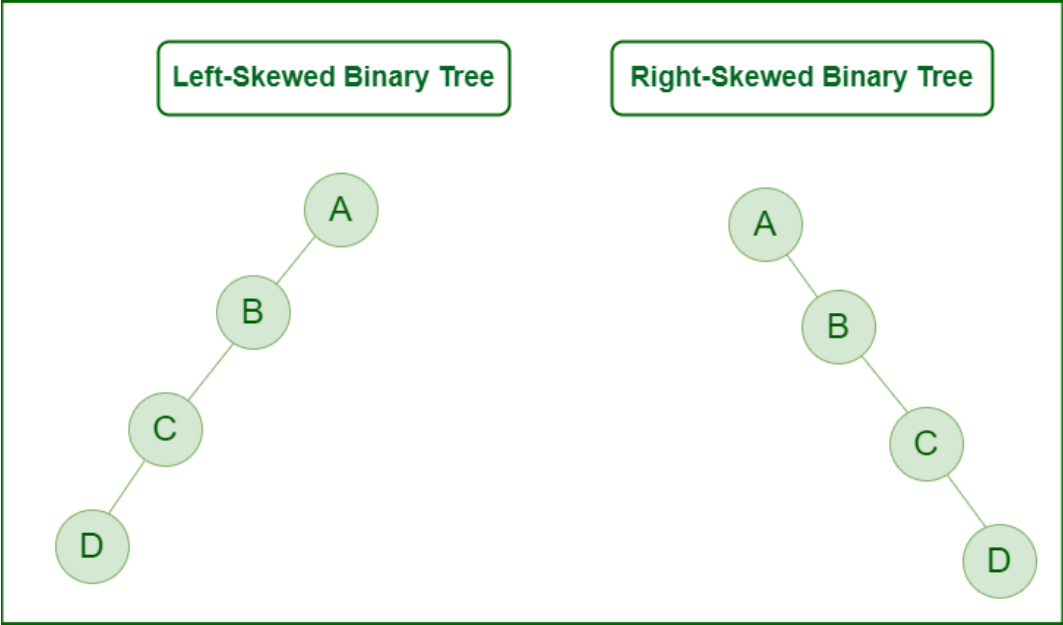
A Tree where every internal node has one child. Such trees are performance-wise same as linked list. A degenerate or pathological tree is a tree having a single child either left or right.



Degenerate (or pathological) tree

3. Skewed Binary Tree

A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: left-skewed binary tree and right-skewed binary tree.



Skewed Binary Tree

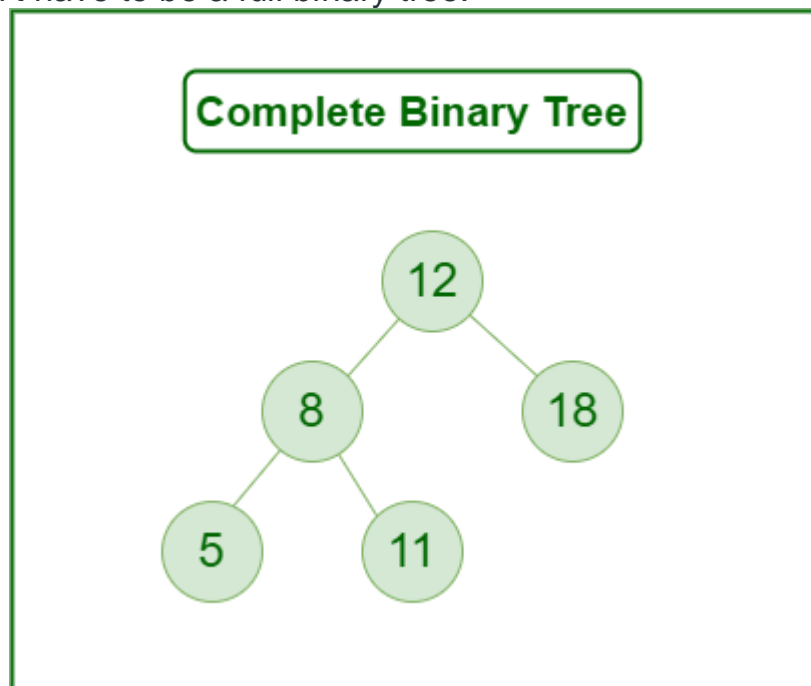
Types of Binary Tree On the basis of the completion of levels:

1. Complete Binary Tree
2. Perfect Binary Tree
3. Balanced Binary Tree

1. Complete Binary Tree

A Binary Tree is a Complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible. A complete binary tree is just like a full binary tree, but with two major differences:

- Every level except the last level must be completely filled.
- All the leaf elements must lean towards the left.
- The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



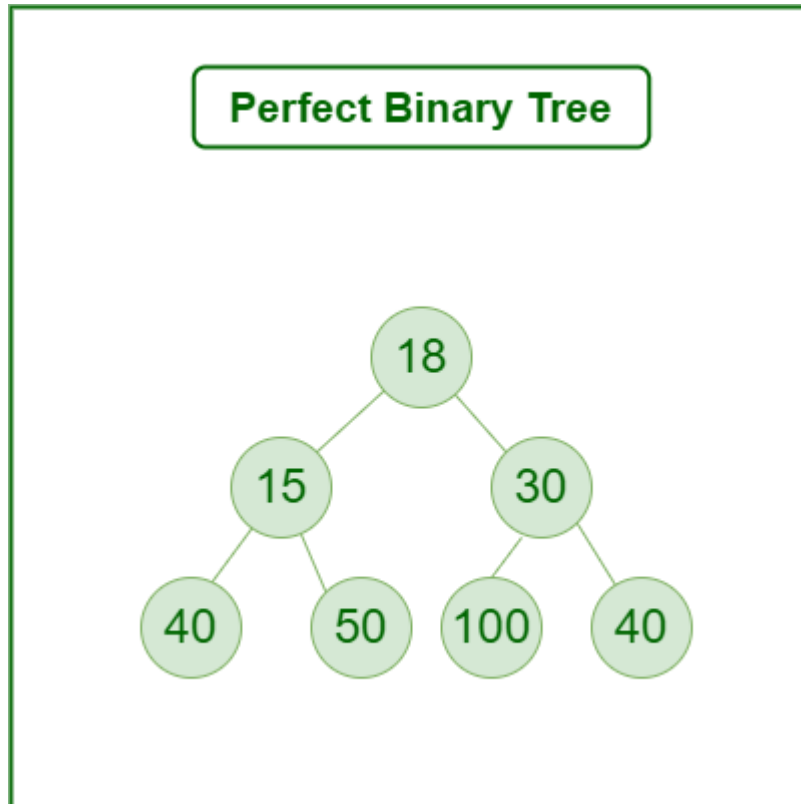
Complete Binary Tree

2. Perfect Binary Tree

A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.

The following are examples of Perfect Binary Trees.

A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.



Perfect Binary Tree

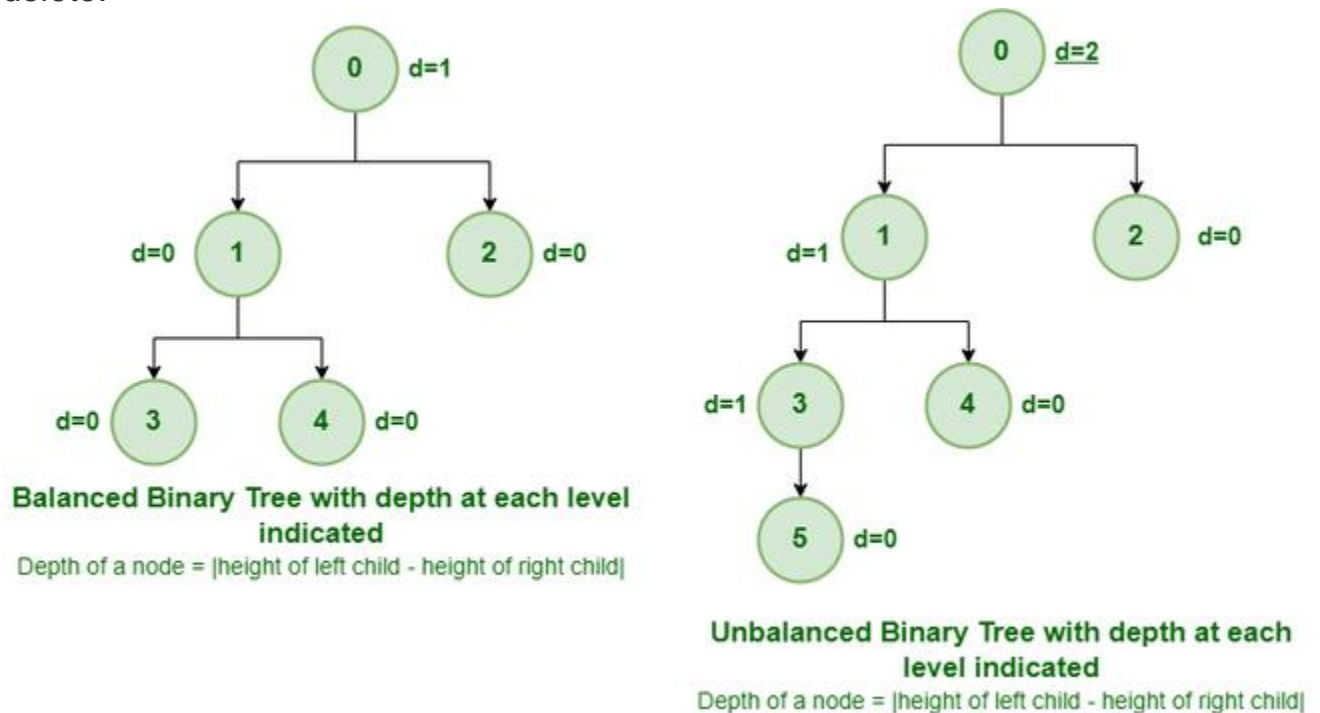
In a Perfect Binary Tree, the number of leaf nodes is the number of internal nodes plus 1

$L = I + 1$ Where L = Number of leaf nodes, I = Number of internal nodes.

3. Balanced Binary Tree

A binary tree is balanced if the height of the tree is $O(\log n)$ where n is the number of nodes. For Example, the AVL tree maintains $O(\log n)$ height by making sure that the difference between the heights of the left and right subtrees is at most 1. Red-Black trees maintain $O(\log n)$ height by making sure that the number of Black nodes on every root to leaf paths is the same and that

there are no adjacent red nodes. Balanced Binary Search trees are performance-wise good as they provide $O(\log n)$ time for search, insert and delete.

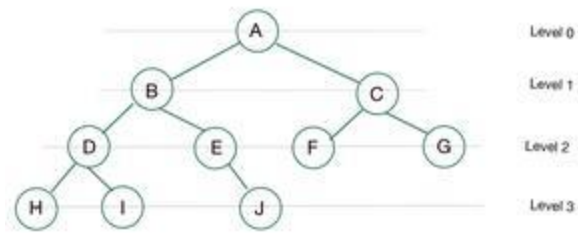


Example of Balanced and Unbalanced Binary Tree

It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1. In the figure above, the root node having a value 0 is unbalanced with a depth of 2 units.

Difference between Full and Complete Binary Tree

A [binary tree](#) is a type of data structure where each node can only have **two offspring** at most named as “**left**” and “**right**” child.

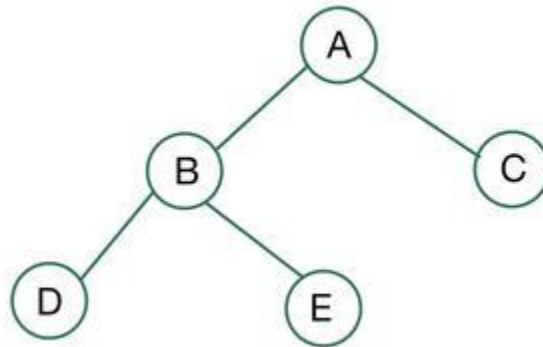


A Binary Tree

There are different [types of binary tree](#) but here we are going to discuss about the difference of **Complete binary tree** and **Full binary tree**.

Full Binary Tree:

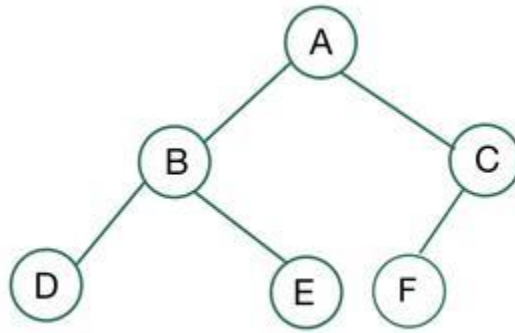
A full binary tree is a binary tree in which **all of the nodes have either 0 or 2 offspring**. In other terms, a full binary tree is a binary tree in which all nodes, except the leaf nodes, have two offspring.



A Full Binary Tree

Complete Binary Tree:

A binary tree is said to be a **complete binary tree** if all its levels, except possibly the last level, have the maximum number of possible nodes, and all the nodes in the **last level appear as far left as possible**.



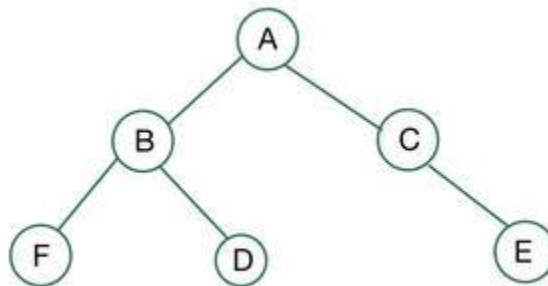
A Complete Binary Tree

There are 2 points that you can recognize from here,

- 1. The leftmost side of the leaf node must always be filled first.*
- 2. It isn't necessary for the last leaf node to have a right sibling.*

Check the following examples to understand the full and complete binary tree in a better way.

Example 1:

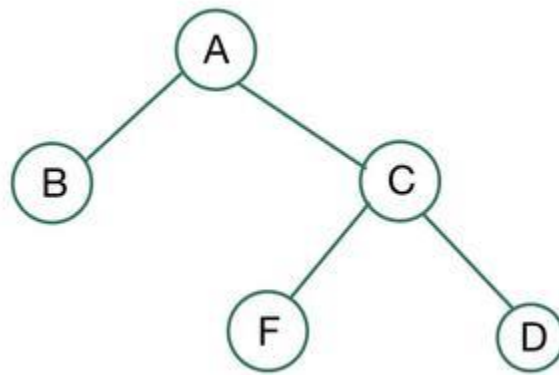


Neither complete nor full

- *Node C has just one child therefore, it is **not a Full binary tree**.*
- *Node C also has a right child but no left child, therefore it is **also not a Complete binary tree**.*

*Hence, the binary tree shown above is **neither complete nor full binary tree**.*

Example 2:

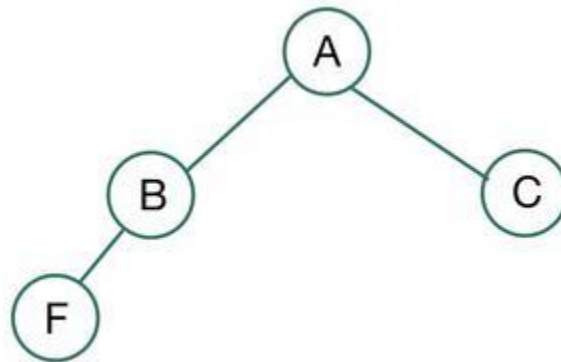


Full but not complete

- All of the nodes have either **0** or **2** offspring, therefore, **it is a Full binary tree**.
- **It is not a Complete binary tree** because node **B** has no children whereas node **C** has children, and according to a complete binary tree, nodes should be filled from the **left side**.

Hence, the binary tree shown above is a **Full binary tree** and it is **not a Complete binary tree**.

Example 3:

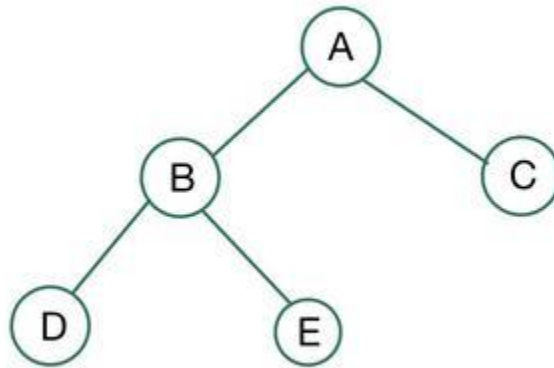


Complete but not full

- **It is a complete binary tree** as all the nodes are left filled.
- Node **B** has just one child, therefore, **it is not a full binary tree**.

Hence, the binary tree shown above is a **Complete binary tree** and it is **not a Full binary tree**.

Example 4:



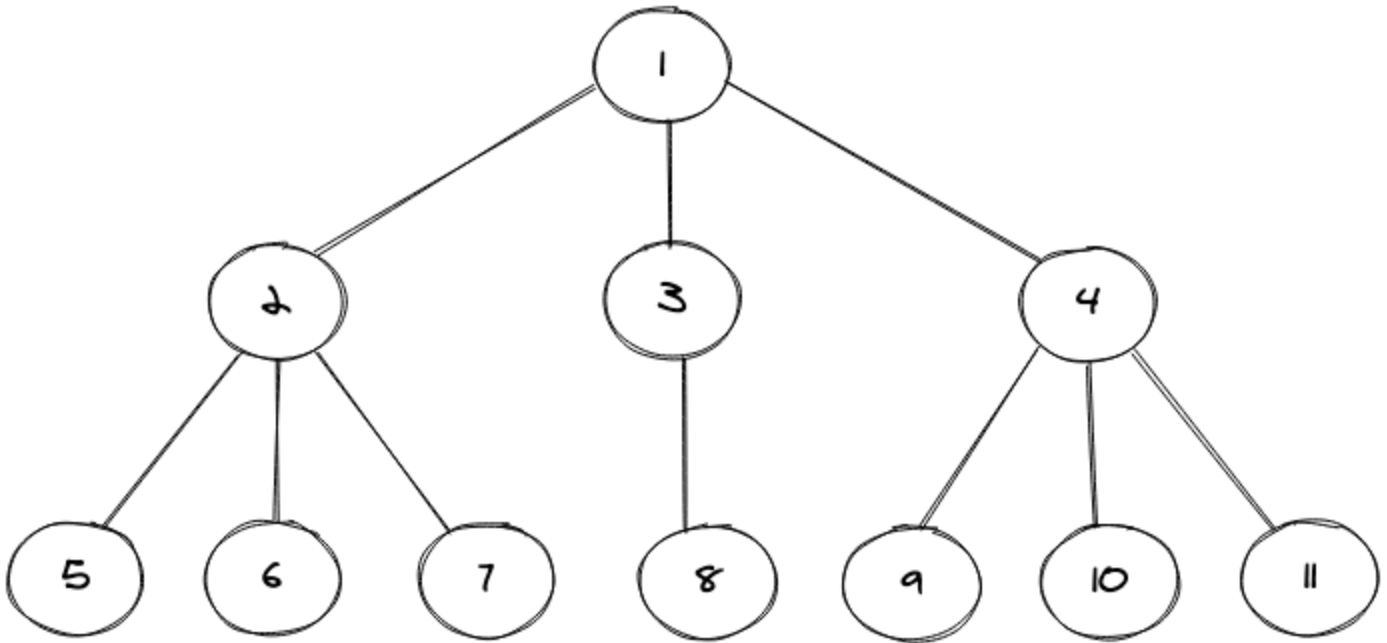
Complete and full

- It is a **Complete binary** tree because all the nodes are **left filled**.
- All of the nodes have either **0** or **2 offspring**, therefore, it is a **full binary tree**.

N-ary Tree Data Structure

The N-ary tree is a tree that allows us to have **n** number of children of a particular node, hence the name **N-ary**, making it **slightly complex than the very common binary trees** that allow us to have at most 2 children of a particular node.

A pictorial representation of an N-ary tree is shown below:



In the N-ary tree shown above, we can notice that there are **11 nodes** in total and **some nodes have three children** and some have had one only. In the case of a binary tree, it was easier to store these children nodes as we can assign two nodes (i.e. left and right) to a particular node to mark its children, but here it's not that simple.

Implementation of N-ary Tree

The first thing that we do when we are dealing with non-linear data structures is to create our own structure (constructors in Java) for them. Like in the case of a binary tree, we make use of a class `TreeNode` and inside that class, we create our constructors and have our class-level variables.

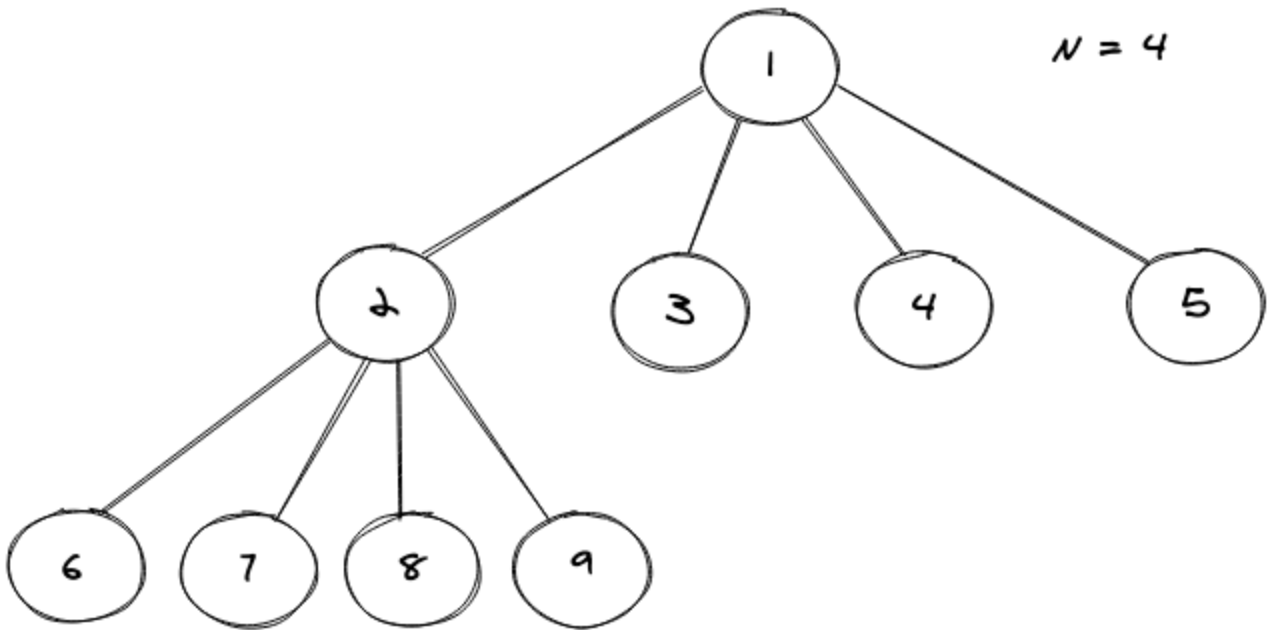
Types of N-ary Tree

Following are the types of N-ary tree:

1. Full N-ary Tree

A Full N-ary Tree is an N-ary tree that allows each node to have either 0 or N children.

Consider the pictorial representation of a full N-ary tree shown below:

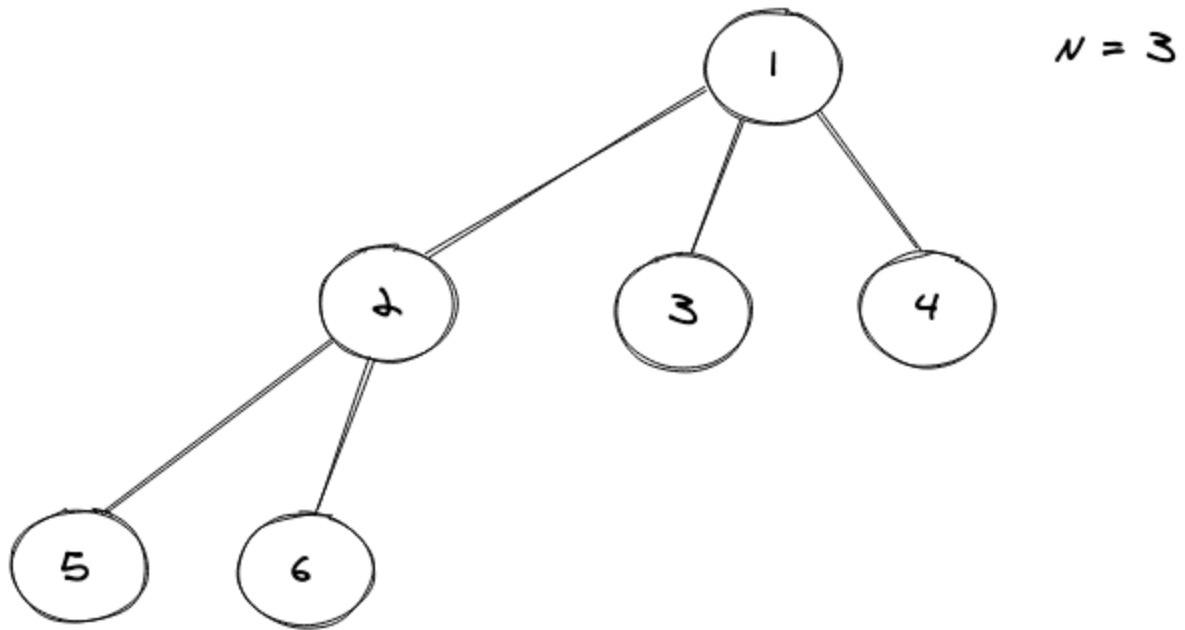


Notice that all the nodes of the above N-ary have either 4 children or 0, hence satisfying the property.

2. Complete N-ary Tree

A complete N-ary tree is an N-ary tree in which the nodes at each level of the tree should be complete (should have exactly **N children**) except the last level nodes and if the last level nodes aren't complete, the nodes must be "as left as possible".

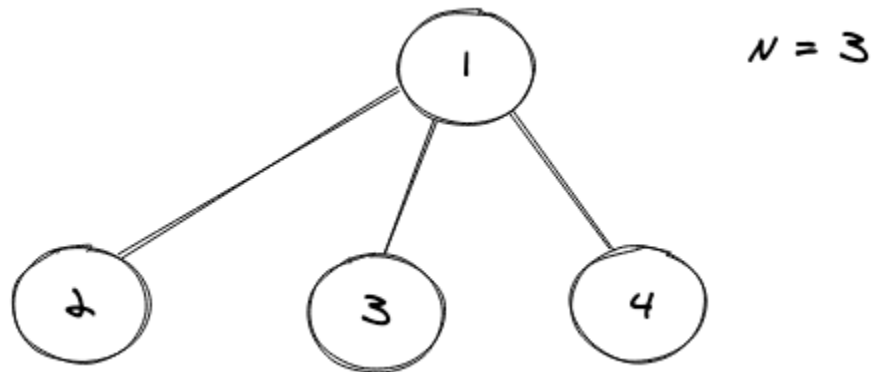
Consider the pictorial representation of a complete N-ary tree shown below:



3. Perfect N-ary Tree

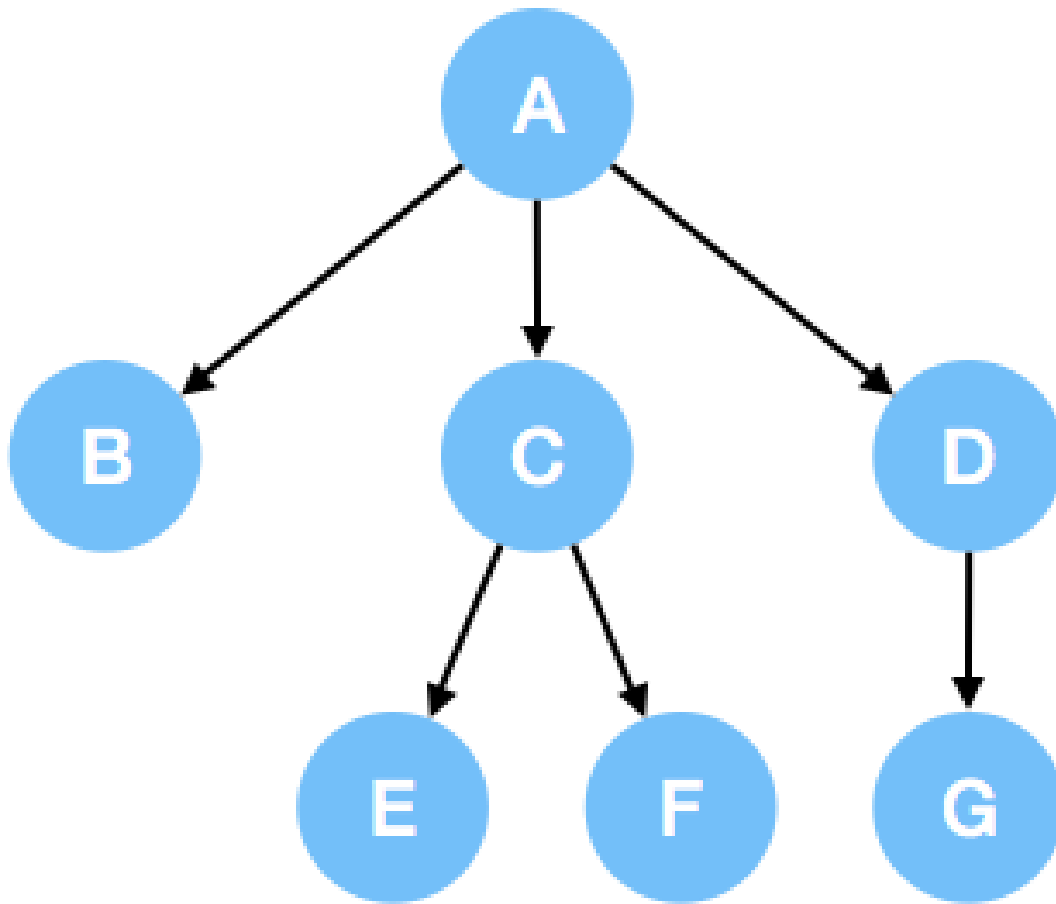
A perfect N-ary tree is a full N-ary tree but the level of the leaf nodes must be the same.

Consider the pictorial representation of a perfect N-ary tree shown below:



N-ary Tree Traversal Examples

We will use the following 3-ary tree as example:



1. Preorder Traversal

In an N-ary tree, preorder means visit the root node first and then traverse the subtree rooted at its children one by one. For instance, the preorder of the 3-ary tree above is: A->B->C->E->F->D->G.

2. Postorder Traversal

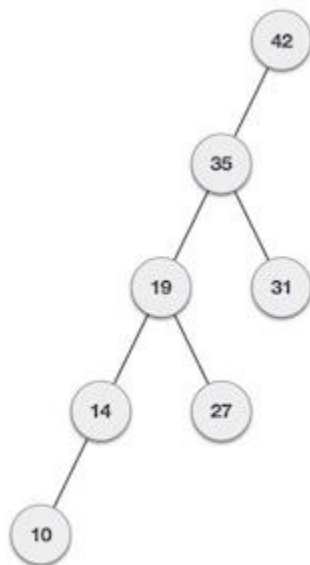
In an N-ary tree, postorder means traverse the subtree rooted at its children first and then visit the root node itself. For instance, the postorder of the 3-ary tree above is: B->E->F->C->G->D->A.

3. Level-order Traversal

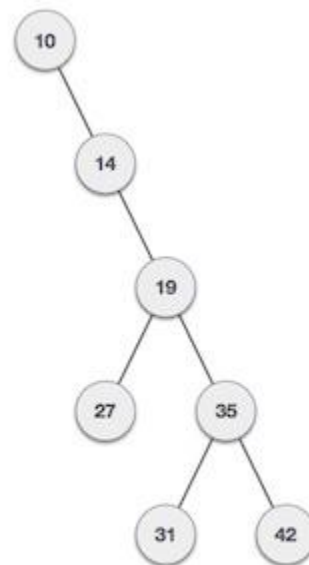
Level-order traversal in an N-ary tree is the same with a binary tree. Typically, when we do breadth-first search in a tree, we will traverse the tree in level order. For instance, the level-order of the 3-ary tree above is: A->B->C->D->E->F->G.

Data Structure and Algorithms - AVL Trees

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



If input 'appears' non-increasing manner

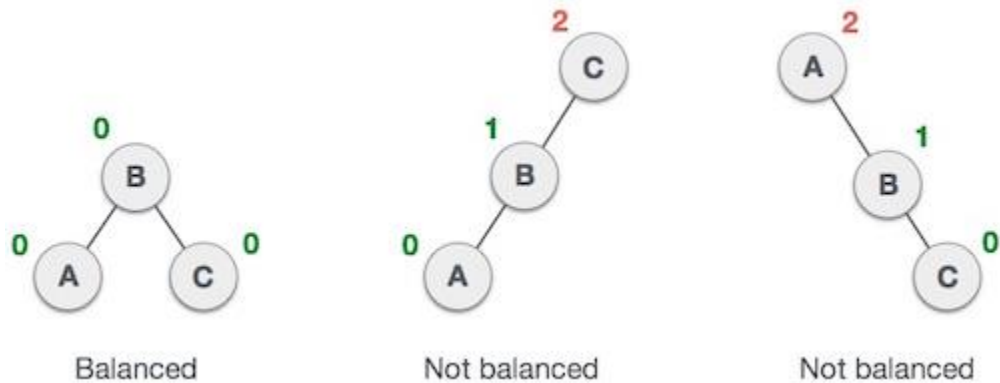


If input 'appears' in non-decreasing manner

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.

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

AVL Rotations

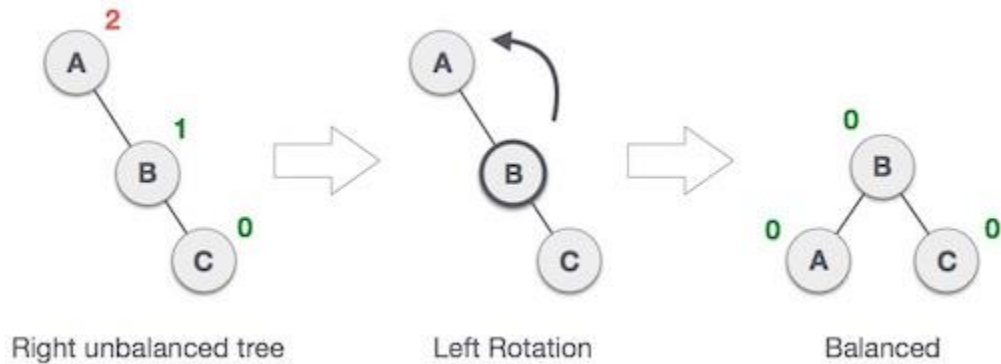
To balance itself, an AVL tree may perform the following four kinds of rotations –

- 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

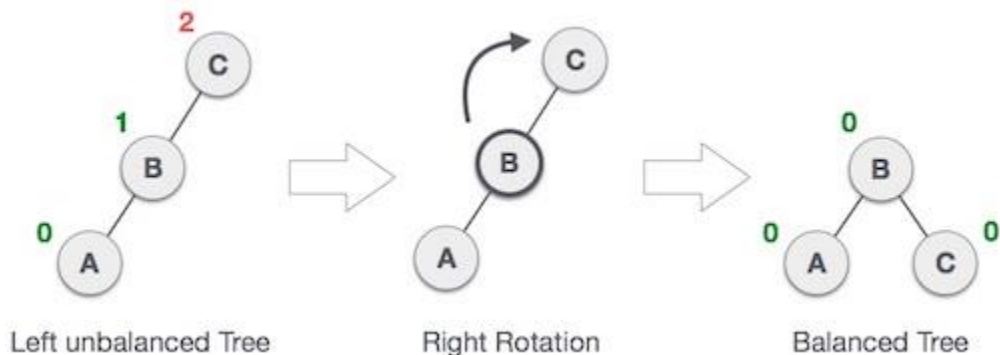
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



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.

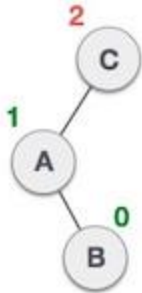


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

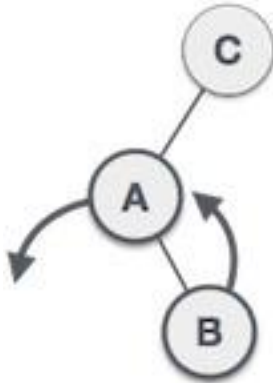
Left-Right Rotation

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.

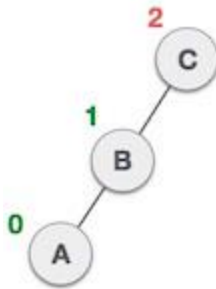
State	Action
-------	--------



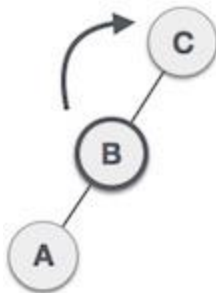
A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.



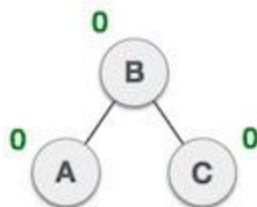
We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.



Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.



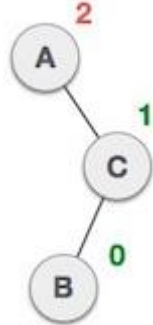
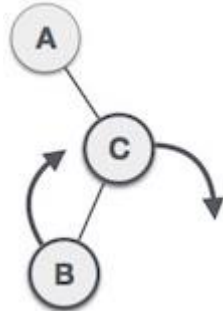
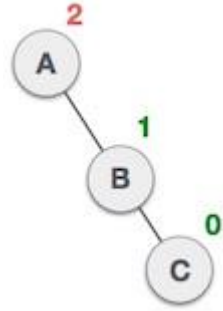
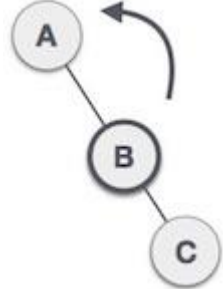
We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.

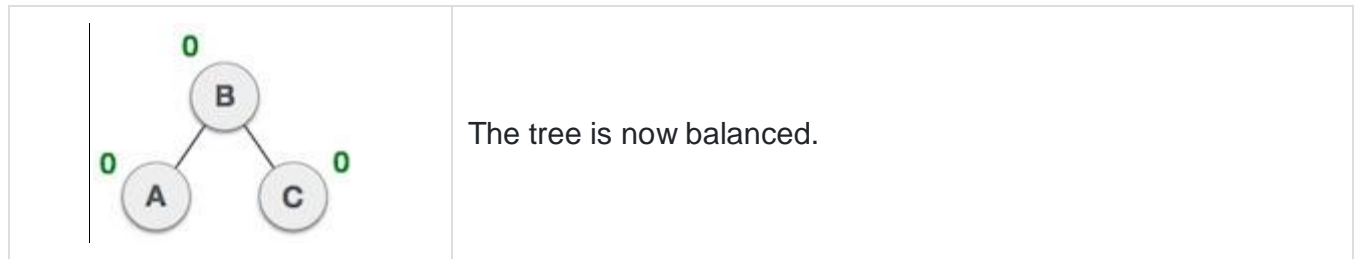


The tree is now balanced.

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
	A node has been inserted into the left subtree of the right subtree. This makes A , an unbalanced node with balance factor 2.
	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 .



Basic Operations of AVL Trees

The basic operations performed on the AVL Tree structures include all the operations performed on a binary search tree, since the AVL Tree at its core is actually just a binary search tree holding all its properties. Therefore, basic operations performed on an AVL Tree are – **Insertion** and **Deletion**.

Insertion operation

The data is inserted into the AVL Tree by following the Binary Search Tree property of insertion, i.e. the left subtree must contain elements less than the root value and right subtree must contain all the greater elements.

However, in AVL Trees, after the insertion of each element, the balance factor of the tree is checked; if it does not exceed 1, the tree is left as it is. But if the balance factor exceeds 1, a balancing algorithm is applied to readjust the tree such that balance factor becomes less than or equal to 1 again.

Algorithm

The following steps are involved in performing the insertion operation of an AVL Tree –

- Step 1 – Create a node
- Step 2 – Check if the tree is empty

Step 3 – If the tree is empty, the new node created will become the root node of the AVL Tree.

Step 4 – If the tree is not empty, we perform the Binary Search Tree insertion operation and check the balancing factor of the node in the tree.

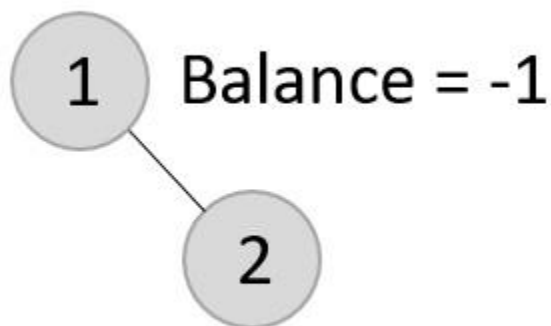
Step 5 – Suppose the balancing factor exceeds ± 1 , we apply suitable rotations on the said node and resume the insertion from Step 4.

Let us understand the insertion operation by constructing an example AVL tree with 1 to 7 integers.

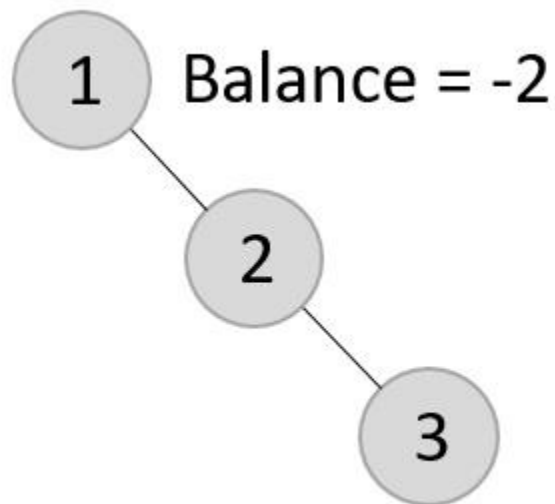
Starting with the first element 1, we create a node and measure the balance, i.e., 0.



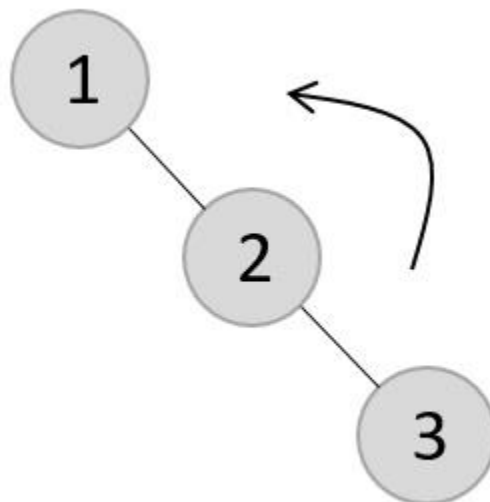
Since both the binary search property and the balance factor are satisfied, we insert another element into the tree.



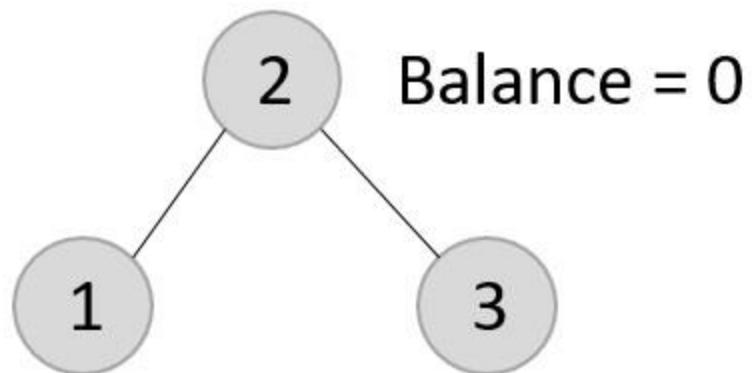
The balance factor for the two nodes are calculated and is found to be -1 (Height of left subtree is 0 and height of the right subtree is 1). Since it does not exceed 1, we add another element to the tree.



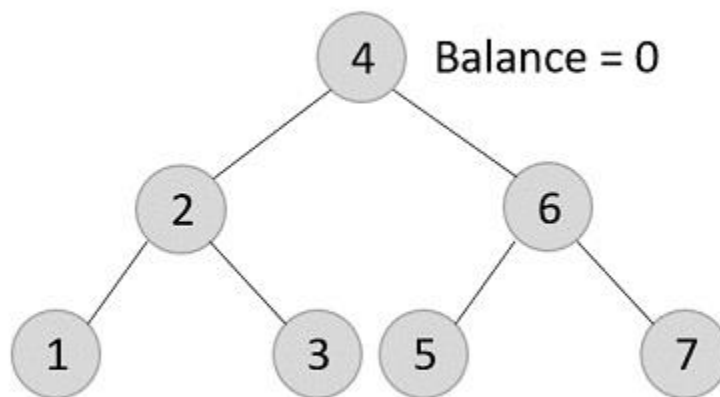
Now, after adding the third element, the balance factor exceeds 1 and becomes 2. Therefore, rotations are applied. In this case, the RR rotation is applied since the imbalance occurs at two right nodes.



The tree is rearranged as –



Similarly, the next elements are inserted and rearranged using these rotations. After rearrangement, we achieve the tree as –

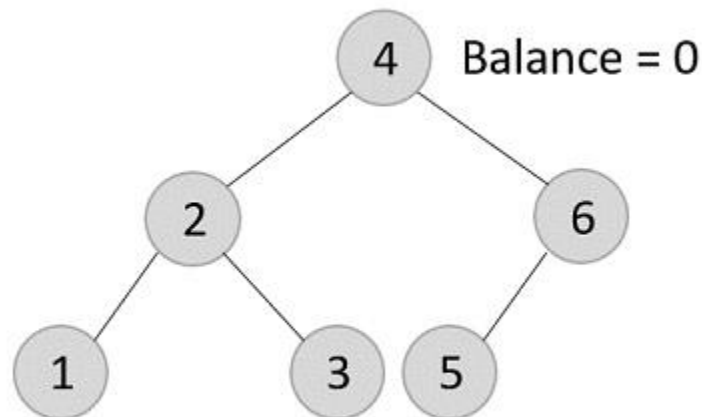


Deletion operation

Deletion in the AVL Trees take place in three different scenarios –

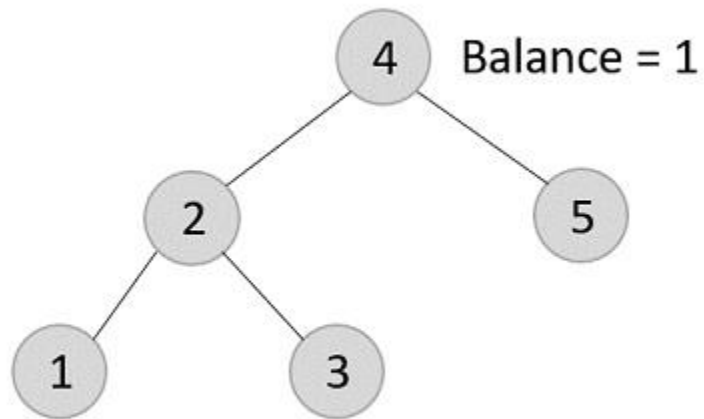
- **Scenario 1 (Deletion of a leaf node)** – If the node to be deleted is a leaf node, then it is deleted without any replacement as it does not disturb the binary search tree property. However, the balance factor may get disturbed, so rotations are applied to restore it.
- **Scenario 2 (Deletion of a node with one child)** – If the node to be deleted has one child, replace the value in that node with the value in its child node. Then delete the child node. If the balance factor is disturbed, rotations are applied.
- **Scenario 3 (Deletion of a node with two child nodes)** – If the node to be deleted has two child nodes, find the inorder successor of that node and replace its value with the inorder successor value. Then try to delete the inorder successor node. If the balance factor exceeds 1 after deletion, apply balance algorithms.

Using the same tree given above, let us perform deletion in three scenarios –



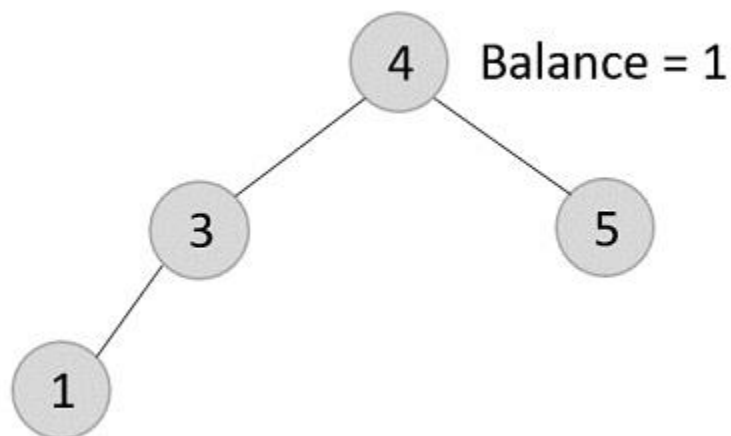
- Deleting element 7 from the tree above –

Since the element 7 is a leaf, we normally remove the element without disturbing any other node in the tree

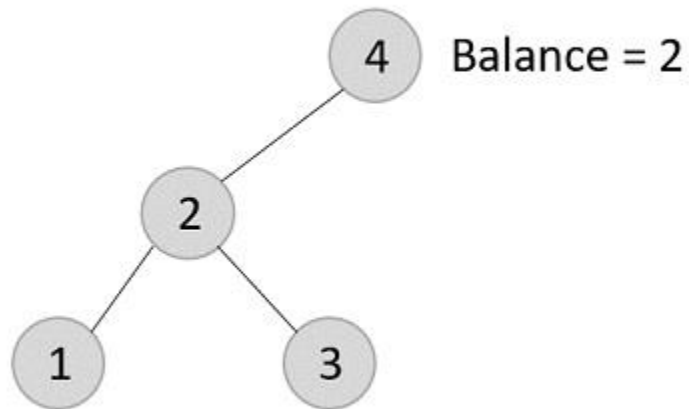


- Deleting element 6 from the output tree achieved –

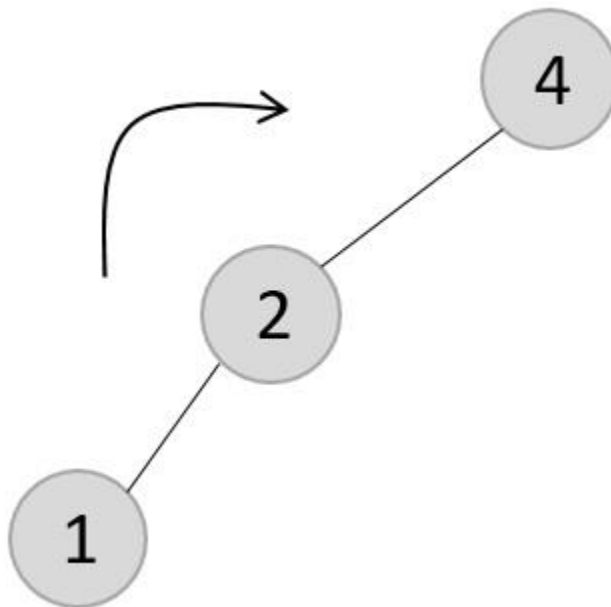
However, element 6 is not a leaf node and has one child node attached to it. In this case, we replace node 6 with its child node: node 5.



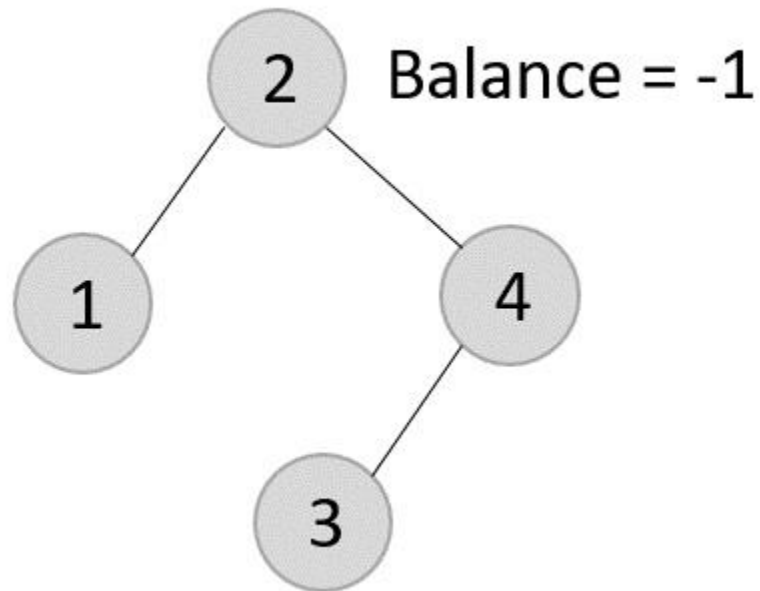
The balance of the tree becomes 1, and since it does not exceed 1 the tree is left as it is. If we delete the element 5 further, we would have to apply the left rotations; either LL or LR since the imbalance occurs at both 1-2-4 and 3-2-4.



The balance factor is disturbed after deleting the element 5, therefore we apply LL rotation (we can also apply the LR rotation here).

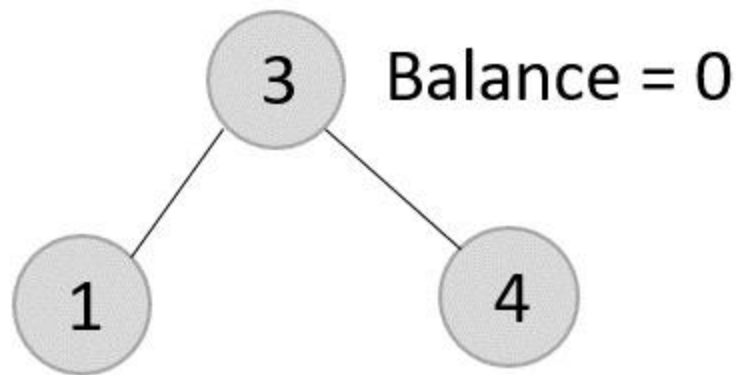


Once the LL rotation is applied on path 1-2-4, the node 3 remains as it was supposed to be the right child of node 2 (which is now occupied by node 4). Hence, the node is added to the right subtree of the node 2 and as the left child of the node 4.



- Deleting element 2 from the remaining tree –

As mentioned in scenario 3, this node has two children. Therefore, we find its inorder successor that is a leaf node (say, 3) and replace its value with the inorder successor.

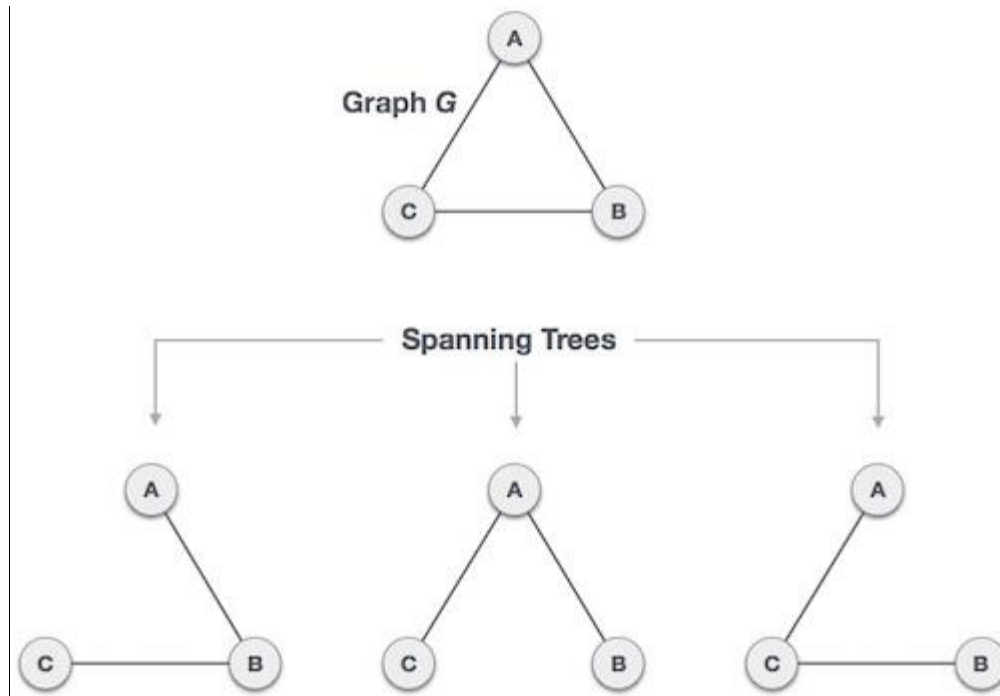


The balance of the tree still remains 1, therefore we leave the tree as it is without performing any rotations.

Data Structure & Algorithms - 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 Minimal Spanning Tree

-
-

Kruskal's minimal spanning tree algorithm is one of the efficient methods to find the minimum spanning tree of a graph. A minimum spanning tree is a subgraph that connects all the vertices present in

the main graph with the least possible edges and minimum cost (sum of the weights assigned to each edge).

The algorithm first starts from the forest – which is defined as a subgraph containing only vertices of the main graph – of the graph, adding the least cost edges later until the minimum spanning tree is created without forming cycles in the graph.

Kruskal's algorithm has easier implementation than prim's algorithm, but has higher complexity.

Kruskal's Algorithm

The inputs taken by the kruskal's algorithm are the graph $G \{V, E\}$, where V is the set of vertices and E is the set of edges, and the source vertex S and the minimum spanning tree of graph G is obtained as an output.

Algorithm

- Sort all the edges in the graph in an ascending order and store it in an array *edge[]*.

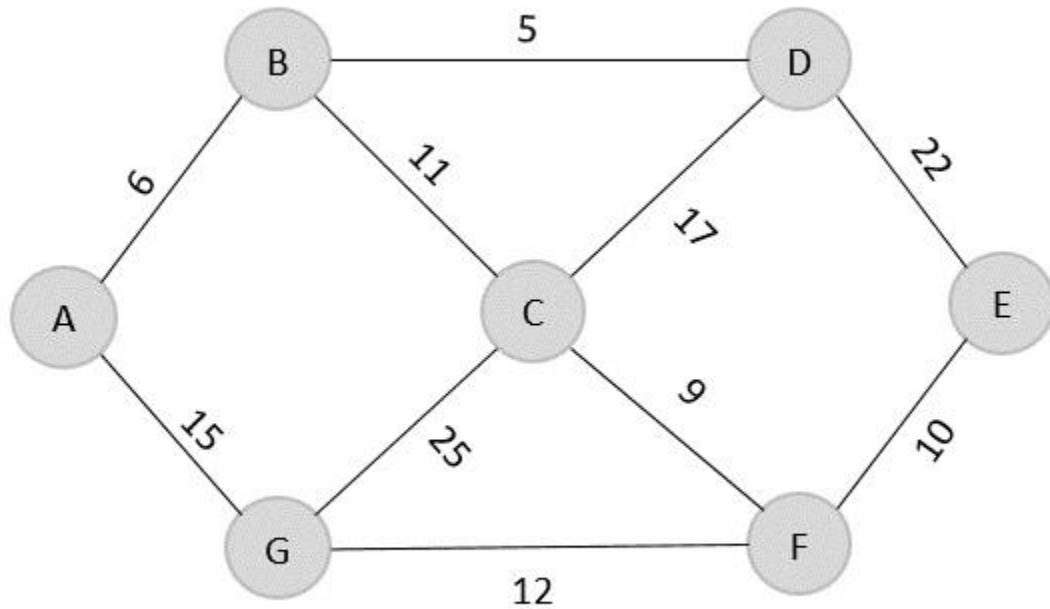
Edge

Cost

- Construct the forest of the graph on a plane with all the vertices in it.
- Select the least cost edge from the *edge[]* array and add it into the forest of the graph. Mark the vertices visited by adding them into the *visited[]* array.
- Repeat the steps 2 and 3 until all the vertices are visited without having any cycles forming in the graph
- When all the vertices are visited, the minimum spanning tree is formed.
- Calculate the minimum cost of the output spanning tree formed.

Examples

Construct a minimum spanning tree using kruskal's algorithm for the graph given below –

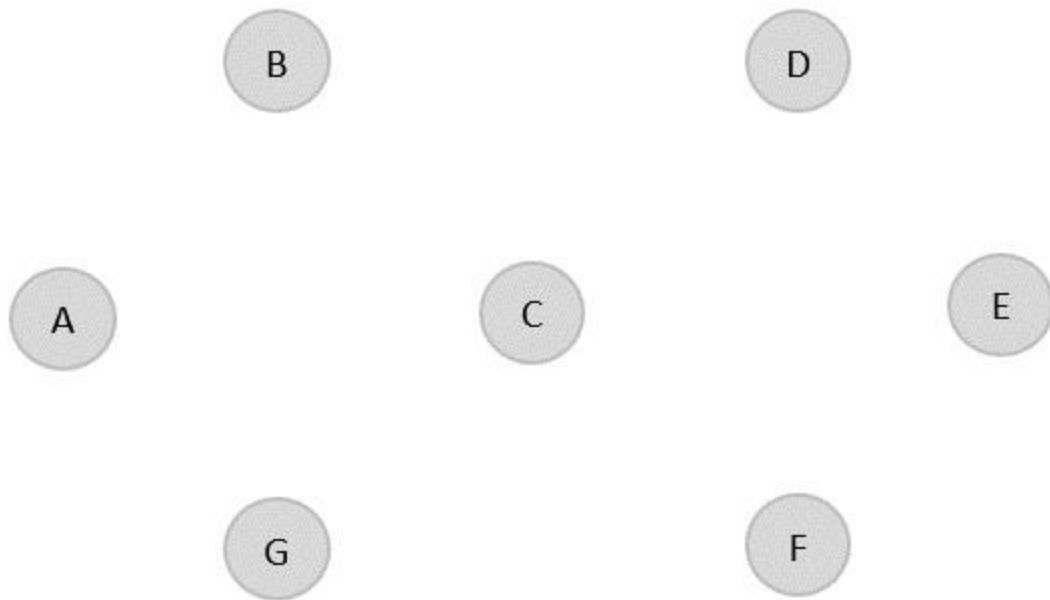


Solution

As the first step, sort all the edges in the given graph in an ascending order and store the values in an array.

Edge	B→D	A→B	C→F	F→E	B→C	G→F	A→G	C→D	D→E	C→G
Cost	5	6	9	10	11	12	15	17	22	25

Then, construct a forest of the given graph on a single plane.

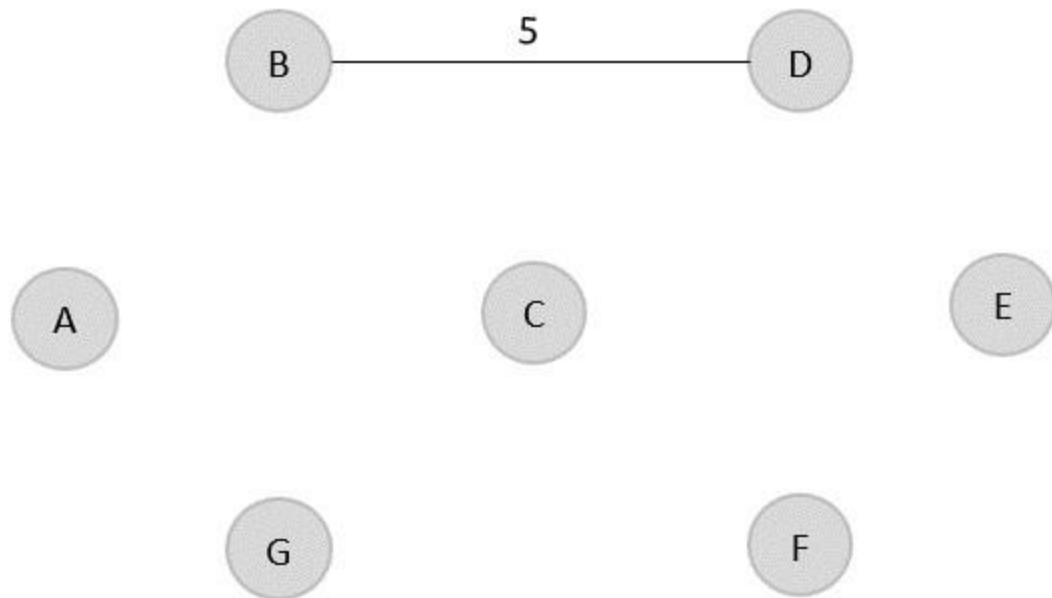


From the list of sorted edge costs, select the least cost edge and add it onto the forest in output graph.

$B \rightarrow D = 5$

Minimum cost = 5

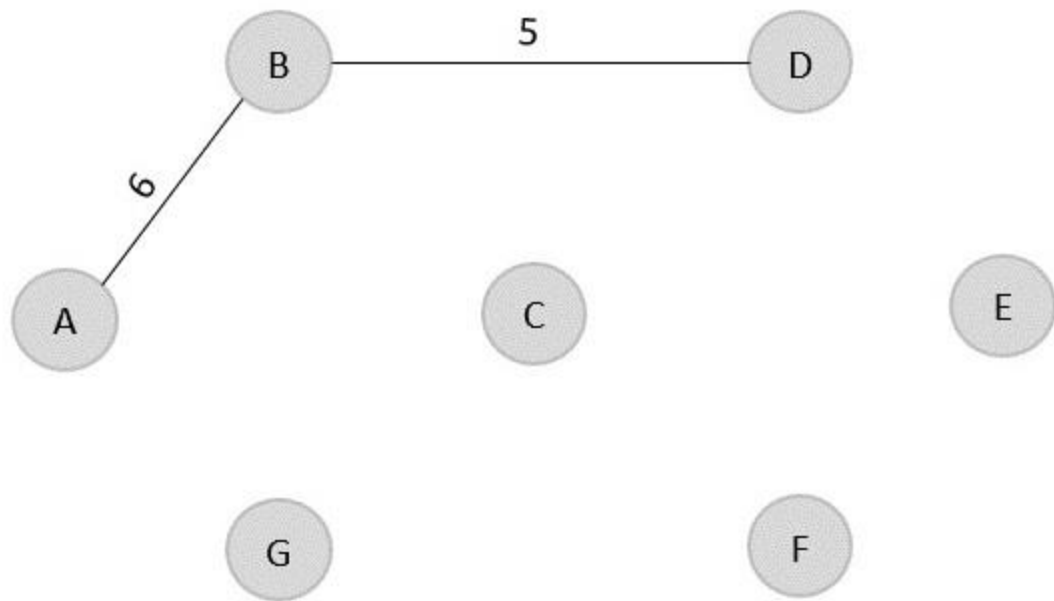
Visited array, $v = \{B, D\}$



Similarly, the next least cost edge is $B \rightarrow A = 6$; so we add it onto the output graph.

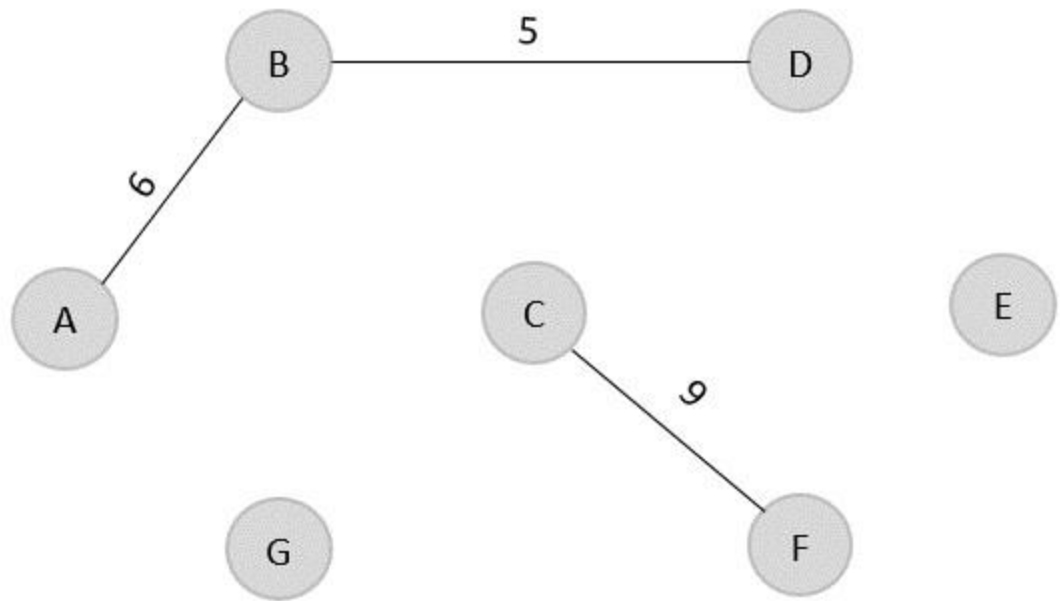
Minimum cost = $5 + 6 = 11$

Visited array, $v = \{B, D, A\}$



The next least cost edge is $C \rightarrow F = 9$; add it onto the output graph.

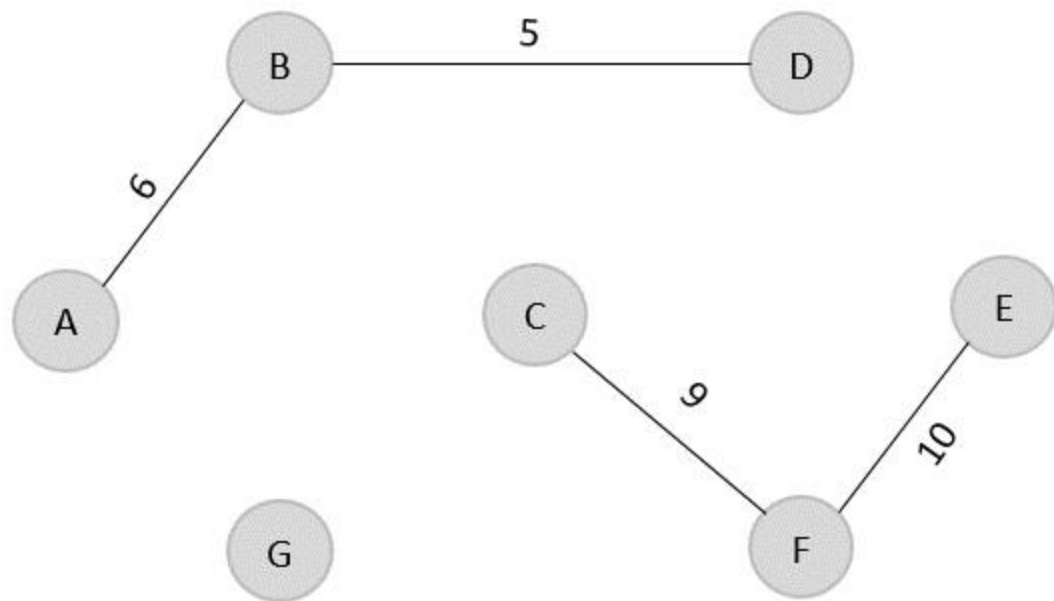
Minimum Cost = $5 + 6 + 9 = 20$
Visited array, $v = \{B, D, A, C, F\}$



The next edge to be added onto the output graph is $F \rightarrow E = 10$.

Minimum Cost = $5 + 6 + 9 + 10 = 30$

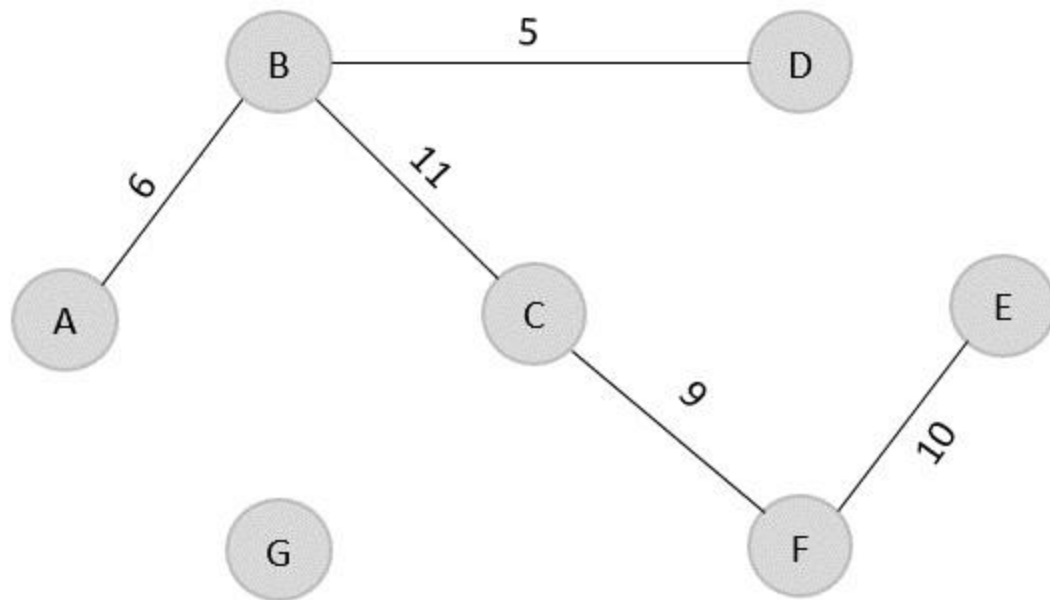
Visited array, $v = \{B, D, A, C, F, E\}$



The next edge from the least cost array is $B \rightarrow C = 11$, hence we add it in the output graph.

Minimum cost = $5 + 6 + 9 + 10 + 11 = 41$

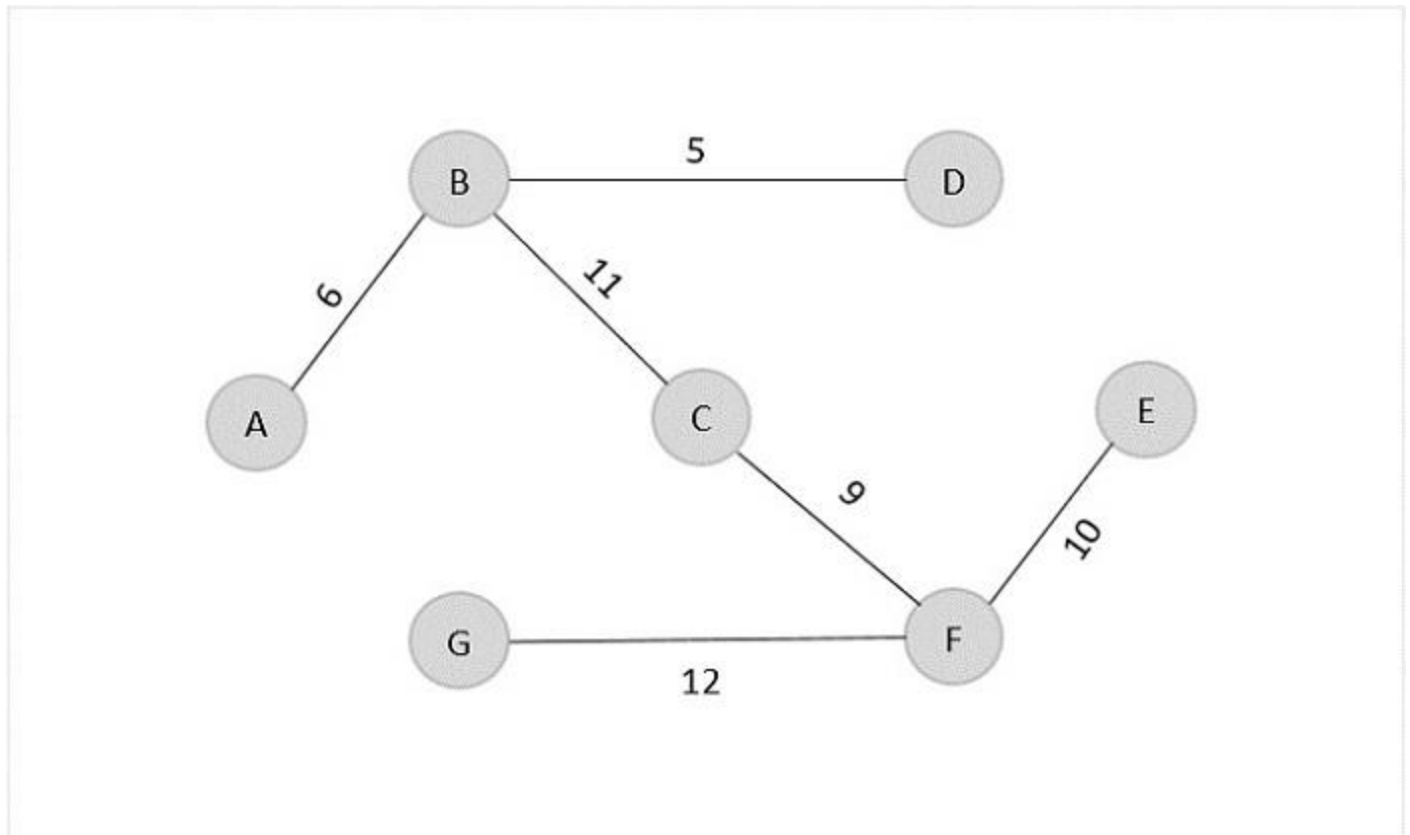
Visited array, $v = \{B, D, A, C, F, E\}$



The last edge from the least cost array to be added in the output graph is $F \rightarrow G = 12$.

Minimum cost = $5 + 6 + 9 + 10 + 11 + 12 = 53$

Visited array, $v = \{B, D, A, C, F, E, G\}$



The obtained result is the minimum spanning tree of the given graph with cost = 53.

Prim's Minimal Spanning Tree

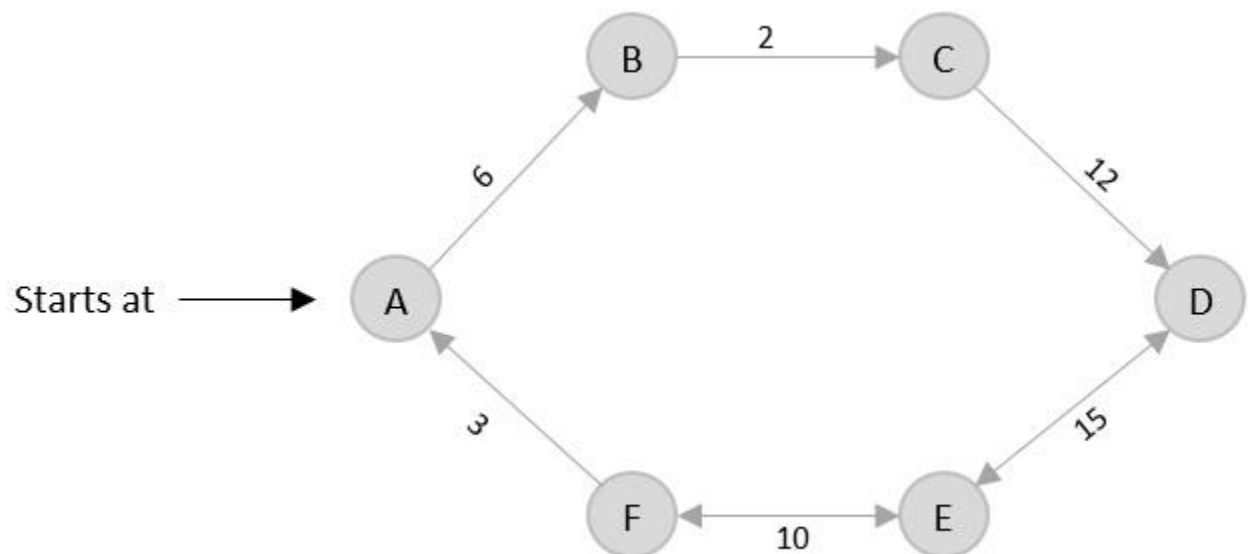
-
-

[Previous Page](#)

[Next Page](#)

Prim's minimal spanning tree algorithm is one of the efficient methods to find the minimum spanning tree of a graph. A minimum spanning tree is a subgraph that connects all the vertices present in the main graph with the least possible edges and minimum cost (sum of the weights assigned to each edge).

The algorithm, similar to any shortest path algorithm, begins from a vertex that is set as a root and walks through all the vertices in the graph by determining the least cost adjacent edges.



Prim's Algorithm

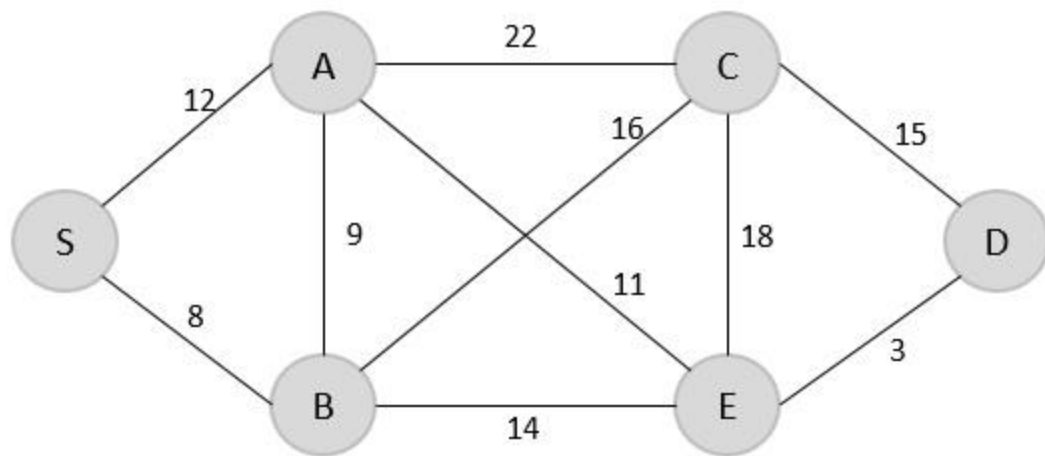
To execute the prim's algorithm, the inputs taken by the algorithm are the graph $G \{V, E\}$, where V is the set of vertices and E is the set of edges, and the source vertex S . A minimum spanning tree of graph G is obtained as an output.

Algorithm

- Declare an array *visited*[] to store the visited vertices and firstly, add the arbitrary root, say S , to the visited array.
- Check whether the adjacent vertices of the last visited vertex are present in the *visited*[] array or not.
- If the vertices are not in the *visited*[] array, compare the cost of edges and add the least cost edge to the output spanning tree.
- The adjacent unvisited vertex with the least cost edge is added into the *visited*[] array and the least cost edge is added to the minimum spanning tree output.
- Steps 2 and 4 are repeated for all the unvisited vertices in the graph to obtain the full minimum spanning tree output for the given graph.
- Calculate the cost of the minimum spanning tree obtained.

Examples

- Find the minimum spanning tree using prim's method (greedy approach) for the graph given below with S as the arbitrary root.



Solution

Step 1

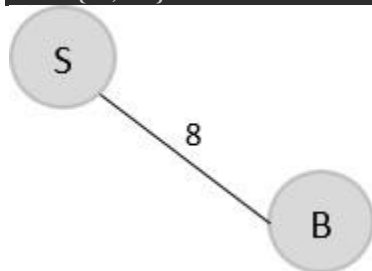
Create a visited array to store all the visited vertices into it.

```
V = { }
```

The arbitrary root is mentioned to be S, so among all the edges that are connected to S we need to find the least cost edge.

```
S → B = 8
```

```
V = {S, B}
```



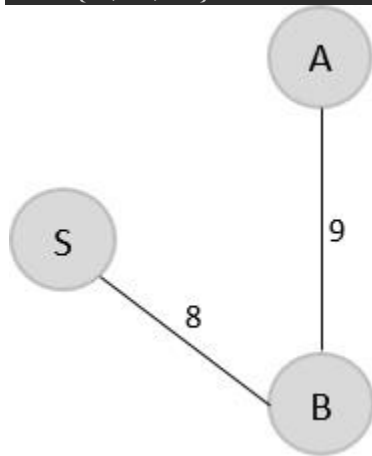
Step 2

Since B is the last visited, check for the least cost edge that is connected to the vertex B.

$B \rightarrow A = 9$
 $B \rightarrow C = 16$
 $B \rightarrow E = 14$

Hence, $B \rightarrow A$ is the edge added to the spanning tree.

$V = \{S, B, A\}$



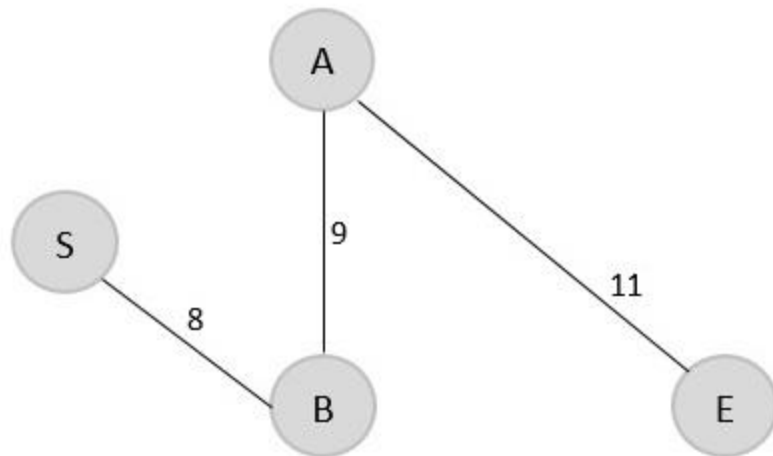
Step 3

Since A is the last visited, check for the least cost edge that is connected to the vertex A.

$A \rightarrow C = 22$
 $A \rightarrow B = 9$
 $A \rightarrow E = 11$

But $A \rightarrow B$ is already in the spanning tree, check for the next least cost edge. Hence, $A \rightarrow E$ is added to the spanning tree.

$V = \{S, B, A, E\}$



Step 4

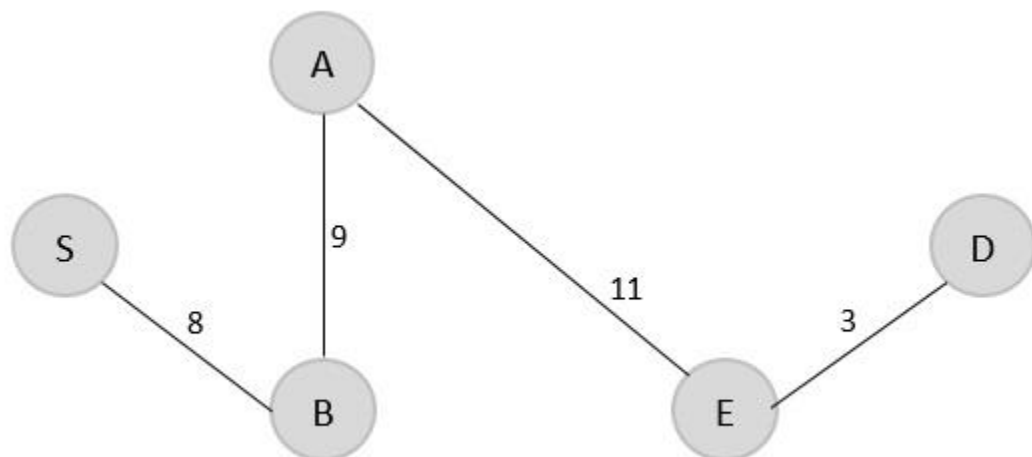
Since E is the last visited, check for the least cost edge that is connected to the vertex E.

$E \rightarrow C = 18$

$E \rightarrow D = 3$

Therefore, $E \rightarrow D$ is added to the spanning tree.

$V = \{S, B, A, E, D\}$



Step 5

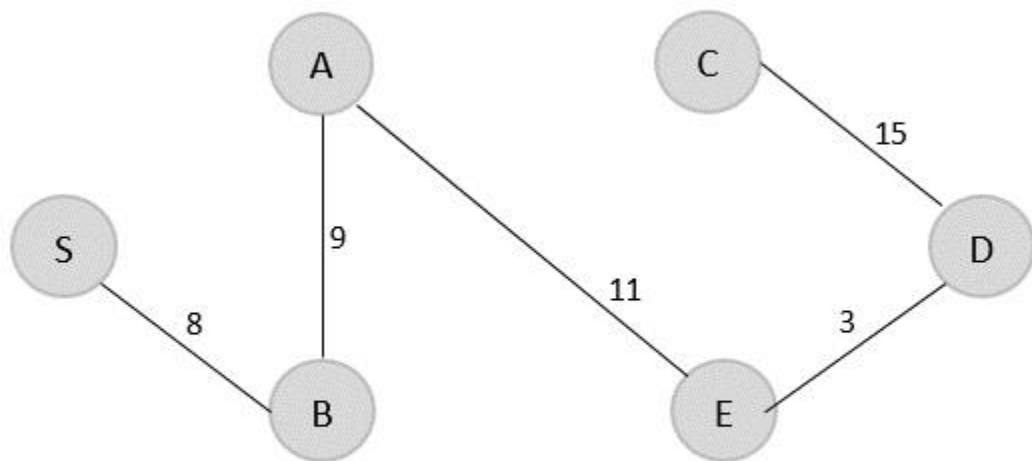
Since D is the last visited, check for the least cost edge that is connected to the vertex D.

$D \rightarrow C = 15$

$E \rightarrow D = 3$

Therefore, $D \rightarrow C$ is added to the spanning tree.

$V = \{S, B, A, E, D, C\}$



The minimum spanning tree is obtained with the minimum cost = 46

Both **Prim's and Kruskal's algorithm** finds the Minimum Spanning Tree and follow the Greedy approach of problem-solving, but there are few **major differences between them.**

Prim's Algorithm	Kruskal's Algorithm
It starts to build the Minimum Spanning Tree from any vertex in the graph.	It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph.
It traverses one node more than one time to get the minimum distance.	It traverses one node only once.
Prim's algorithm gives connected component as well as it works only on connected graph.	Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components
It generates the minimum spanning tree starting from the root vertex.	It generates the minimum spanning tree starting from the least weighted edge.
Applications of prim's algorithm are Travelling Salesman Problem, Network for roads and Rail tracks connecting all the cities etc.	Applications of Kruskal algorithm are LAN connection, TV Network etc.

Red Black Trees

Red-Black Trees are another type of the Balanced Binary Search Trees with two coloured nodes: Red and Black. It is a self-balancing binary search tree that makes use of these colours to maintain the balance factor during the insertion and deletion operations. Hence, during the Red-Black Tree operations, the memory uses 1 bit of storage to accommodate the colour information of each node

In Red-Black trees, also known as RB trees, there are different conditions to follow while assigning the colours to the nodes.

- The root node is always black in colour.
- No two adjacent nodes must be red in colour.
- Every path in the tree (from the root node to the leaf node) must have the same amount of black coloured nodes.

Even though AVL trees are more balanced than RB trees, with the balancing algorithm in AVL trees being stricter than that of RB trees, multiple and faster insertion and deletion operations are made more efficient through RB trees.

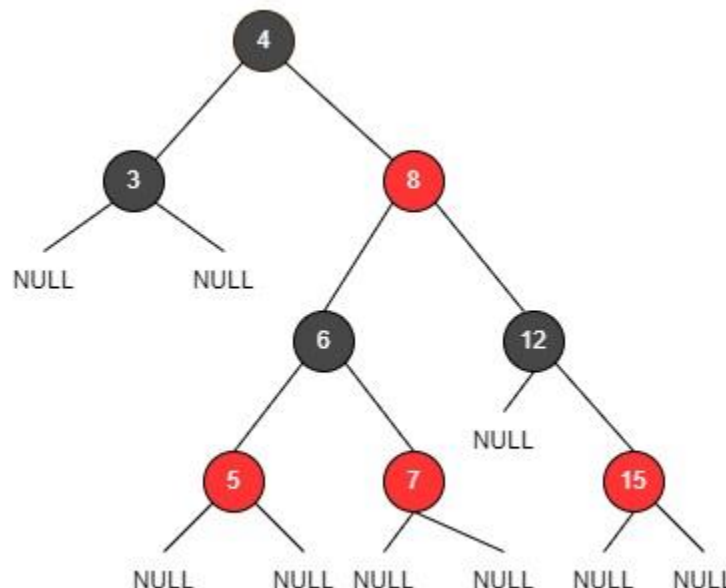


Fig: RB trees

Basic Operations of Red-Black Trees

The operations on Red-Black Trees include all the basic operations usually performed on a Binary Search Tree. Some of the basic operations of an RB Tree include –

- Insertion
- Deletion
- Search

Insertion operation

Insertion operation of a Red-Black tree follows the same insertion algorithm of a binary search tree. The elements are inserted following the binary search property and as an addition, the nodes are color coded as red and black to balance the tree according to the red-black tree properties.

Follow the procedure given below to insert an element into a red-black tree by maintaining both binary search tree and red black tree properties.

Case 1 – Check whether the tree is empty; make the current node as the root and color the node black if it is empty.

Case 2 – But if the tree is not empty, we create a new node and color it red. Here we face two different cases –

- If the parent of the new node is a black colored node, we exit the operation and tree is left as it is.
- If the parent of this new node is red and the color of the parent's sibling is either black or if it does not exist, we apply a suitable rotation and recolor accordingly.
- If the parent of this new node is red and color of the parent's sibling is red, recolor the parent, the sibling and grandparent nodes to black. The grandparent is recolored only if it is **not** the root node; if it is the root node recolor only the parent and the sibling.

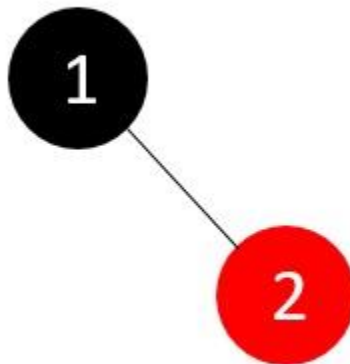
Example

Let us construct an RB Tree for the first 7 integer numbers to understand the insertion operation in detail –

The tree is checked to be empty so the first node added is a root and is colored black.

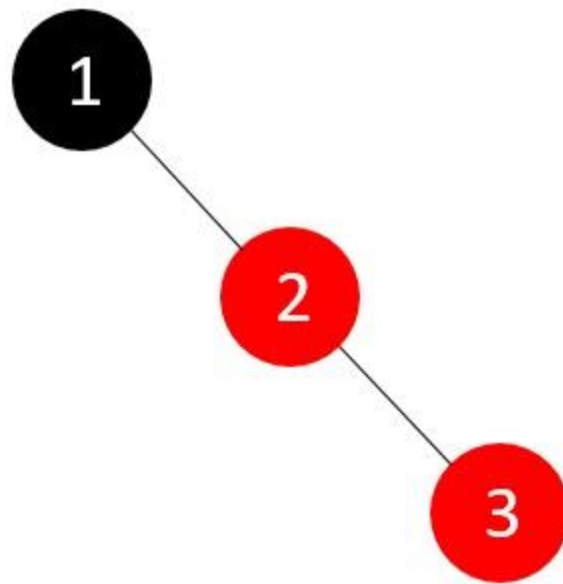


Now, the tree is not empty so we create a new node and add the next integer with color red,

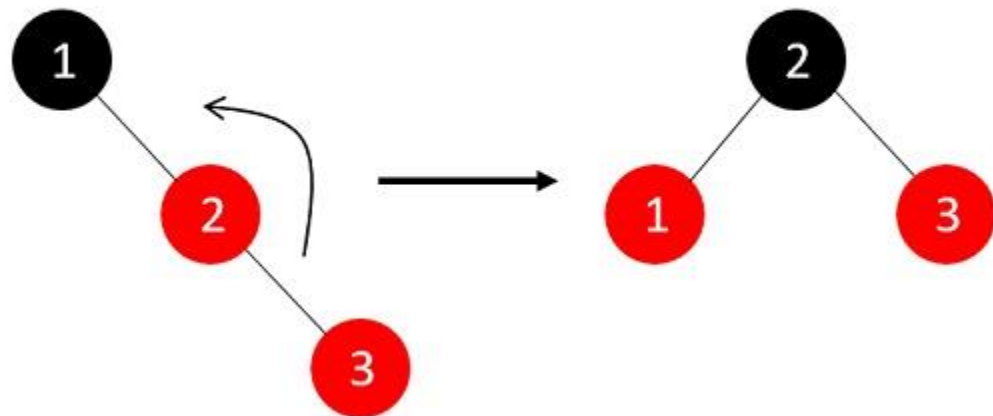


The nodes do not violate the binary search tree and RB tree properties, hence we move ahead to add another node.

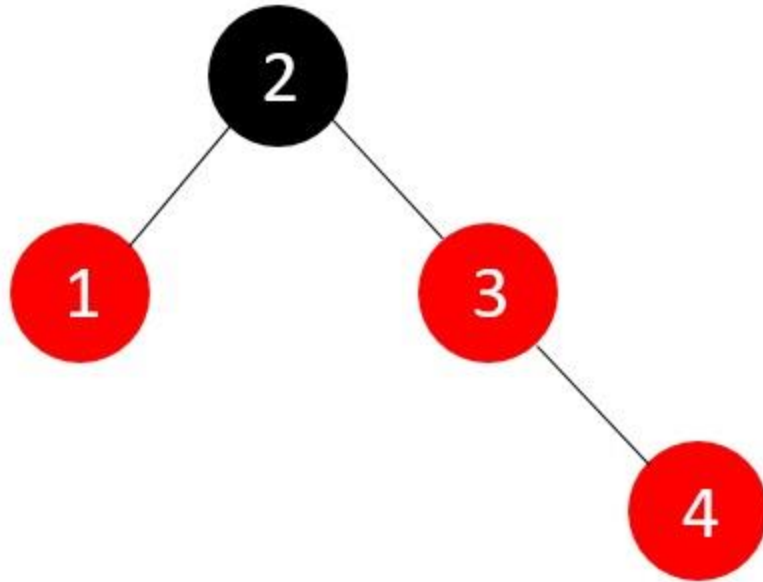
The tree is not empty; we create a new red node with the next integer to it. But the parent of the new node is not a black colored node,



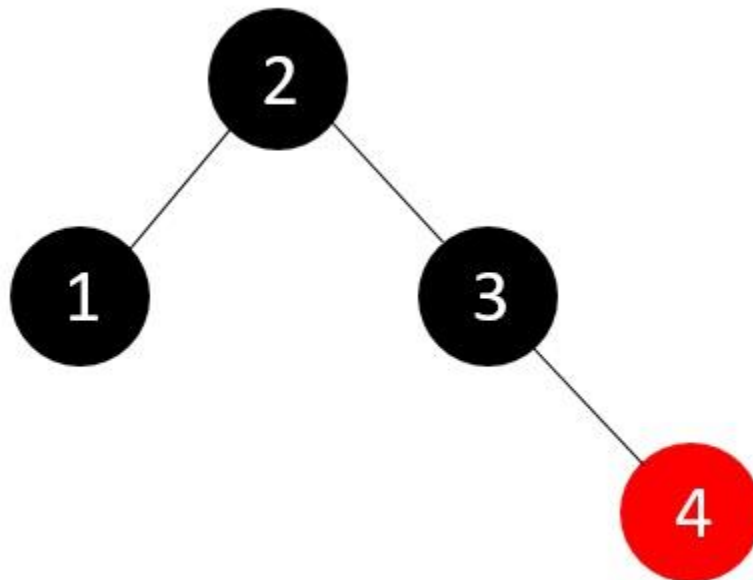
The tree right now violates both the binary search tree and RB tree properties; since parent's sibling is NULL, we apply a suitable rotation and recolor the nodes.



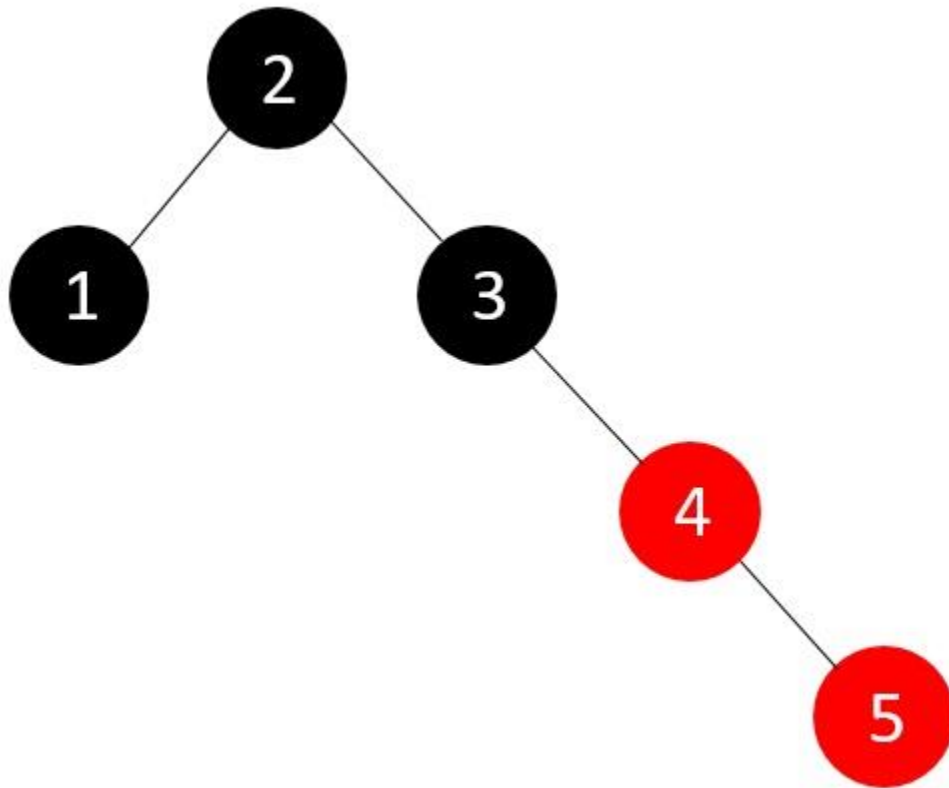
Now that the RB Tree property is restored, we add another node to the tree –



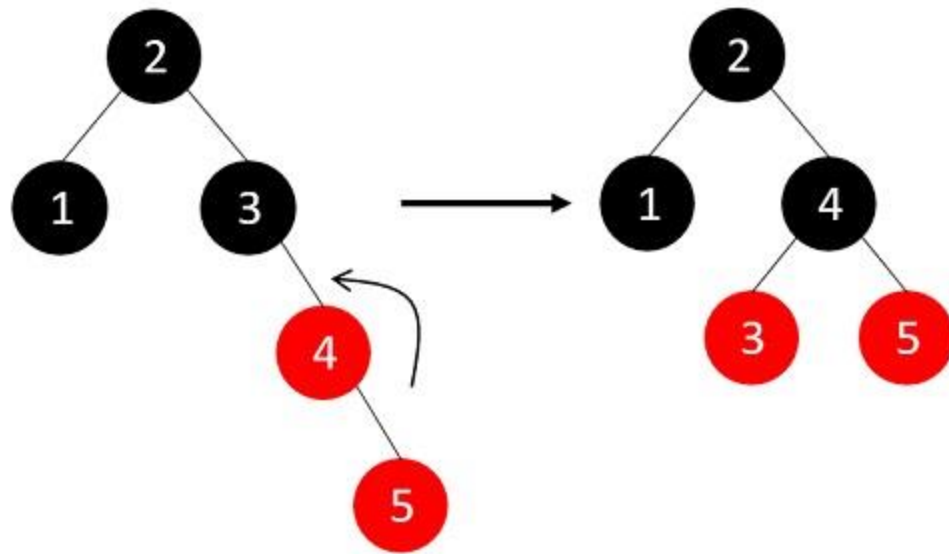
The tree once again violates the RB Tree balance property, so we check for the parent's sibling node color, red in this case, so we just recolor the parent and the sibling.



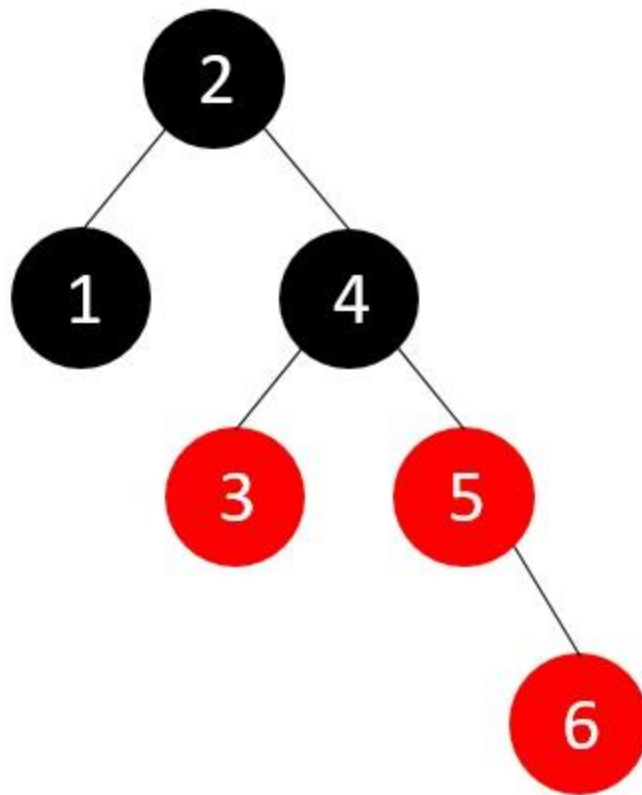
We next insert the element 5, which makes the tree violate the RB Tree balance property once again.



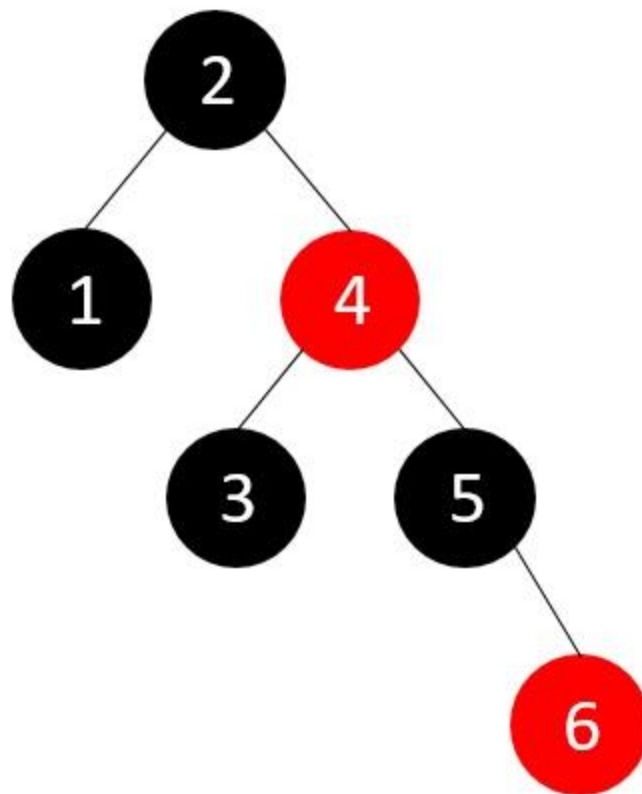
And since the sibling is NULL, we apply suitable rotation and recolor.



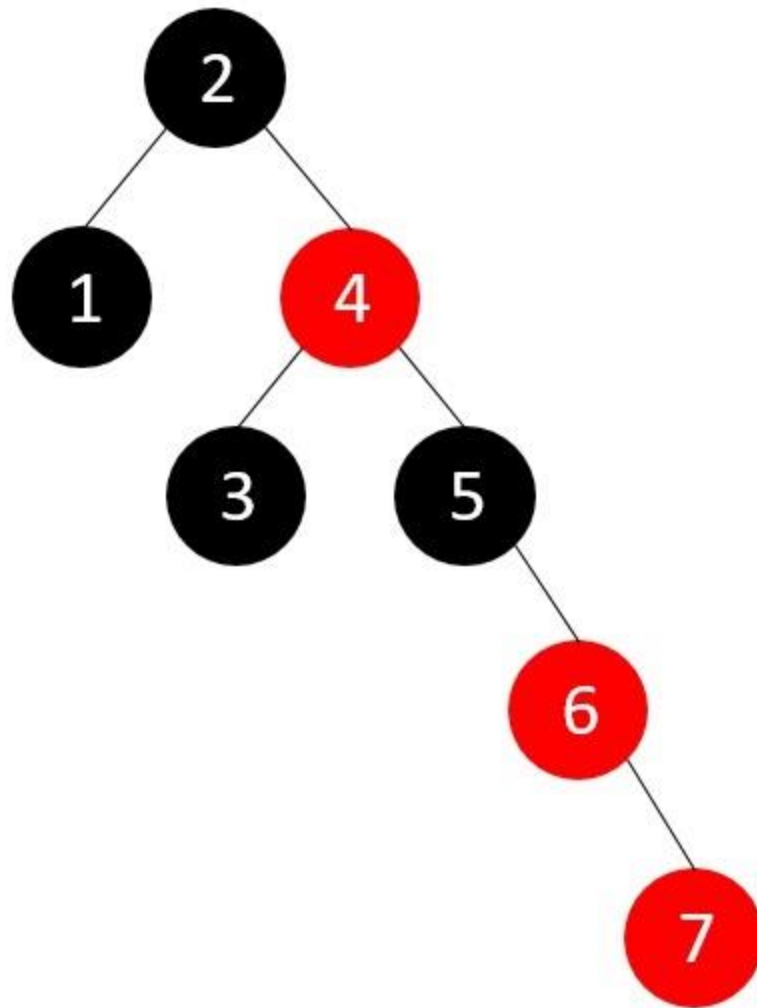
Now, we insert element 6, but the RB Tree property is violated and one of the insertion cases need to be applied –



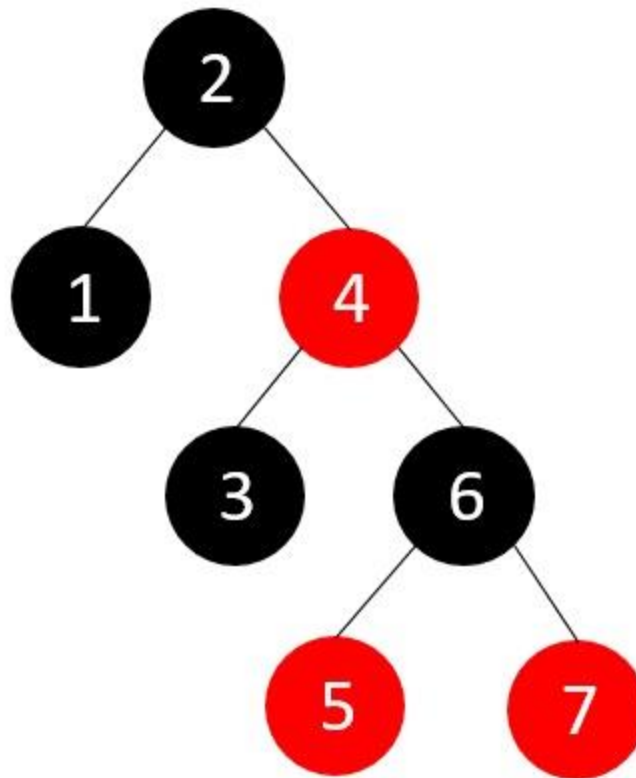
The parent's sibling is red, so we recolor the parent, parent's sibling and the grandparent nodes since the grandparent is not the root node.



Now, we add the last element, 7, but the parent node of this new node is red.



Since the parent's sibling is NULL, we apply suitable rotations (RR rotation)



The final RB Tree is achieved.

Deletion operation

The deletion operation on red black tree must be performed in such a way that it must restore all the properties of a binary search tree and a red black tree. Follow the steps below to perform the deletion operation on the red black tree –

Firstly, we perform deletion based on the binary search tree properties.

Case 1 – If either the node to be deleted or the node's parent is red, just delete it.

Case 2 – If the node is a double black, just remove the double black (double black occurs when the node to be deleted is a black colored

leaf node, as it adds up the NULL nodes which are considered black colored nodes too)

Case 3 – If the double black's sibling node is also a black node and its child nodes are also black in color, follow the steps below –

- Remove double black
- Recolor its parent to black (if the parent is a red node, it becomes black; if the parent is already a black node, it becomes double black)
- Recolor the parent's sibling with red
- If double black node still exists, we apply other cases.

Case 4 – If the double black node's sibling is red, we perform the following steps –

- Swap the colors of the parent node and the parent's sibling node.
- Rotate parent node in the double black's direction
- Reapply other cases that are suitable.

Case 5 – If the double black's sibling is a black node but the sibling's child node that is closest to the double black is red, follows the steps below –

- Swap the colors of double black's sibling and the sibling's child in question
- Rotate the sibling node in the opposite direction of double black (i.e. if the double black is a right child apply left rotations and vice versa)
- Apply case 6.

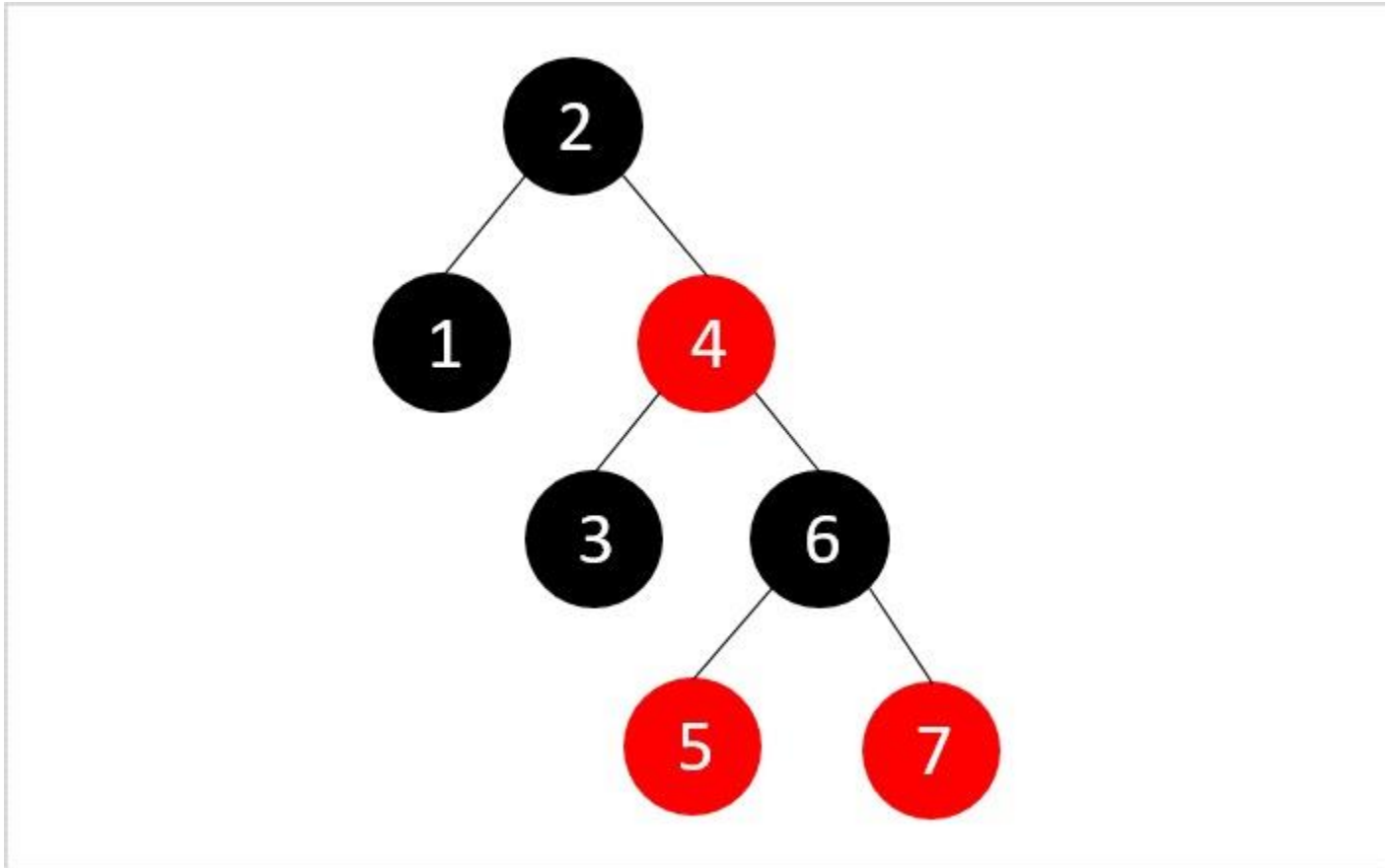
Case 6 – If the double black's sibling is a black node but the sibling's child node that is farther to the double black is red, follows the steps below –

- Swap the colors of double black's parent and sibling nodes
- Rotate the parent in double black's direction (i.e. if the double black is a right child apply right rotations and vice versa)
- Remove double black

- Change the color of red child node to black.

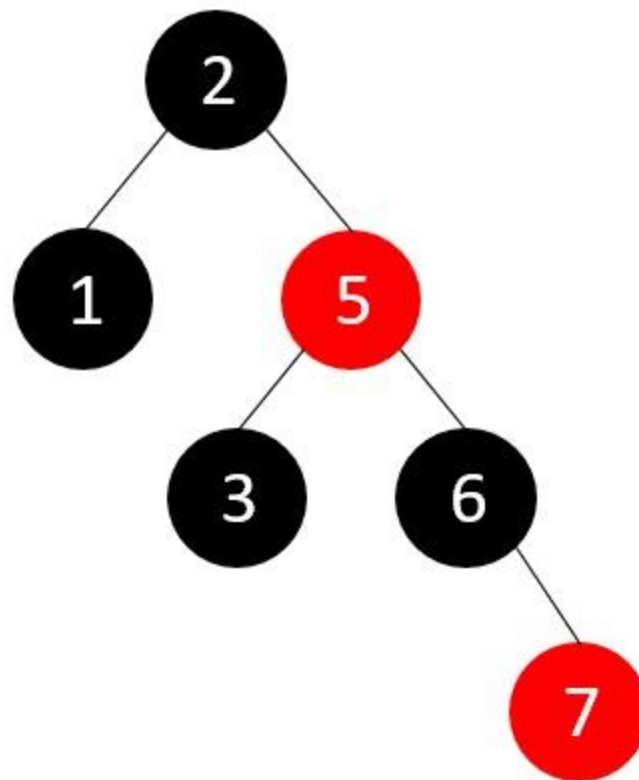
Example

Considering the same constructed Red-Black Tree above, let us delete few elements from the tree.



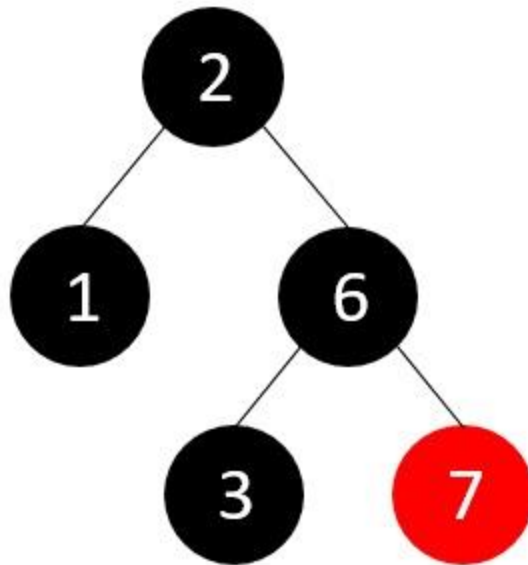
Delete elements 4, 5, 3 from the tree.

To delete the element 4, let us perform the binary search deletion first.

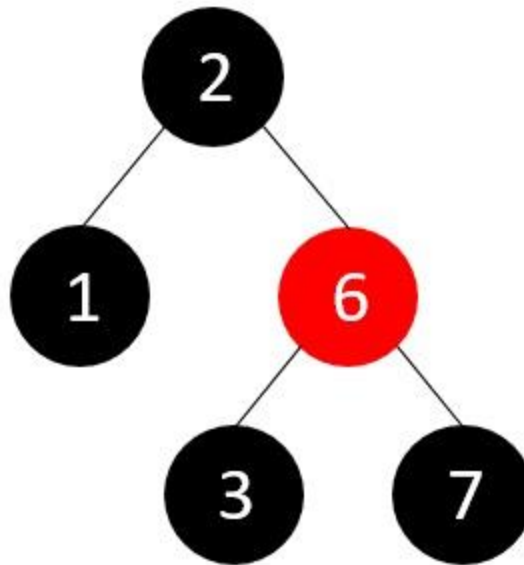


After performing the binary search deletion, the RB Tree property is not disturbed, therefore the tree is left as it is.

Then, we delete the element 5 using the binary search deletion

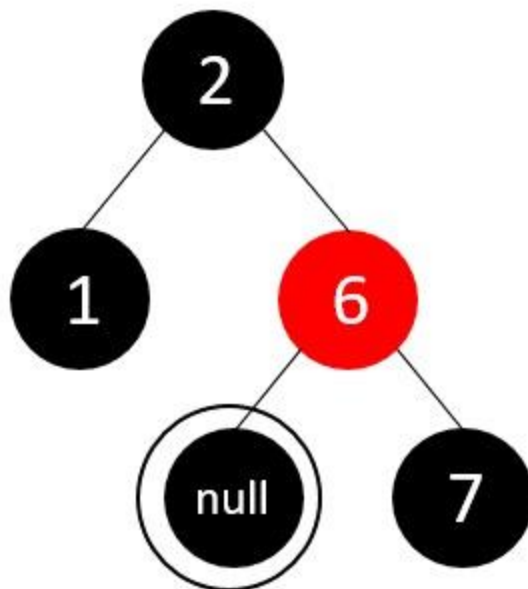


But the RB property is violated after performing the binary search deletion, i.e., all the paths in the tree do not hold same number of black nodes; so we swap the colors to balance the tree.

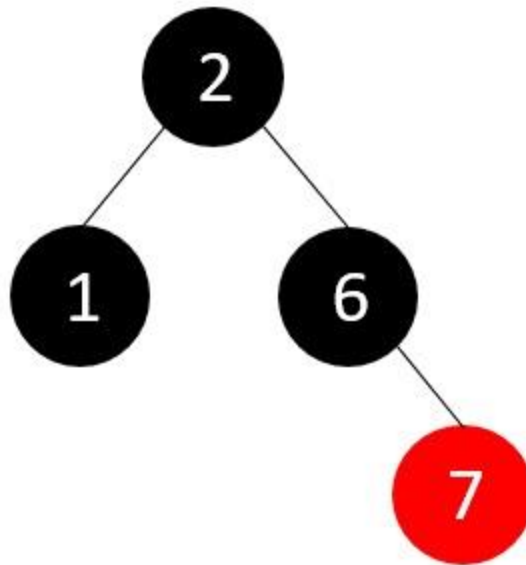


Then, we delete the node 3 from the tree obtained –

Applying binary search deletion, we delete node 3 normally as it is a leaf node. And we get a double node as 3 is a black colored node.



We apply case 3 deletion as double black's sibling node is black and its child nodes are also black. Here, we remove the double black, recolor the double black's parent and sibling.



All the desired nodes are deleted and the RB Tree property is maintained.

Search operation

The search operation in red-black tree follows the same algorithm as that of a binary search tree. The tree is traversed and each node is compared with the key element to be searched; if found it returns a successful search. Otherwise, it returns an unsuccessful search.