UNIT-III

In linear data structure data is organized in sequential order and in non-linear data structure data is organized in random order. A tree is a very popular non-linear data structure used in a wide range of applications.

Tree

A tree is a non-linear hierarchical data structure that consists of nodes connected by edges.

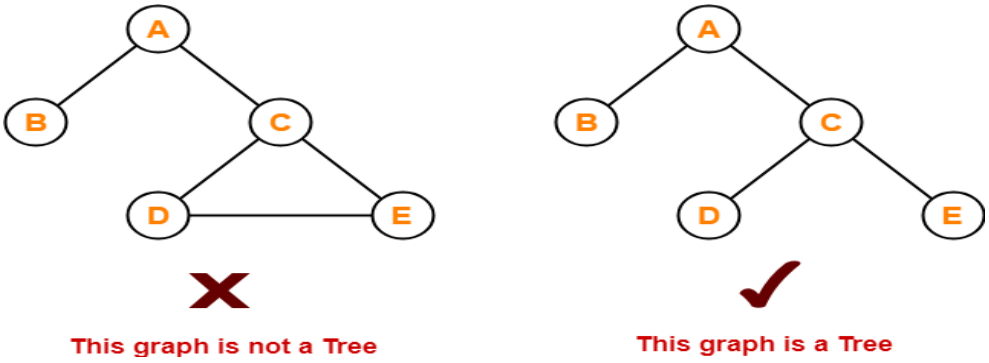
OR

- A tree is a connected graph without any circuits.
- If in a graph, there is one and only one path between every pair of vertices, then graph is called as a tree.

Why Tree Data Structure?

- Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially.
- In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But, it is not acceptable in today's computational world.
- Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

Example:



Properties:

The important properties of tree data structure are-

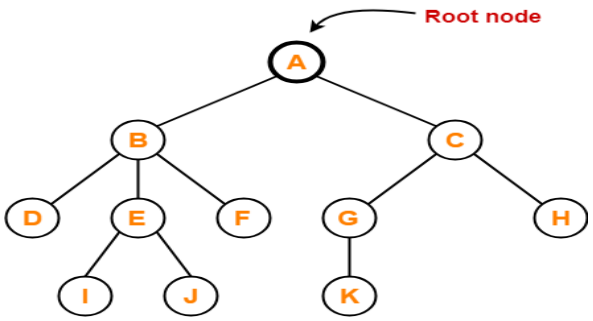
- There is one and only one path between every pair of vertices in a tree.
- A tree with n vertices has exactly (n-1) edges.
- A graph is a tree if and only if it is minimally connected.
- Any connected graph with n vertices and (n-1) edges is a tree.

Tree Terminology

1.Root

- The first node from where the tree originates is called as a **root node**.
- In any tree, there must be only one root node.
- We can never have multiple root nodes in a tree data structure.

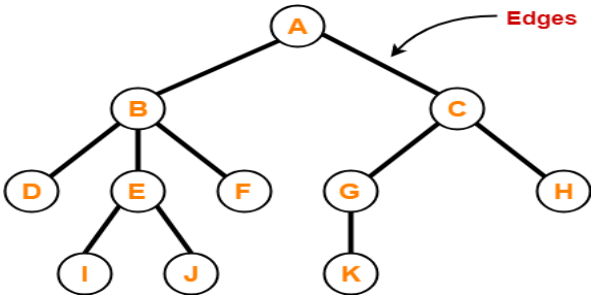
Example



2. Edge

- The connecting link between any two nodes is called as an **edge**.
- In a tree with n number of nodes, there are exactly (n-1) number of edges.

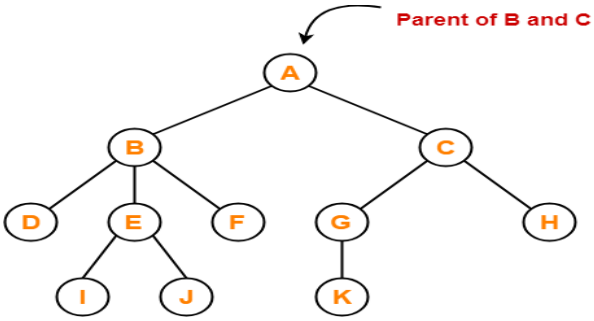
Example



3. Parent

- The node which has a branch from it to any other node is called as a **parent node**.
- In other words, the node which has one or more children is called as a parent node.
- In a tree, a parent node can have any number of child nodes.

Example



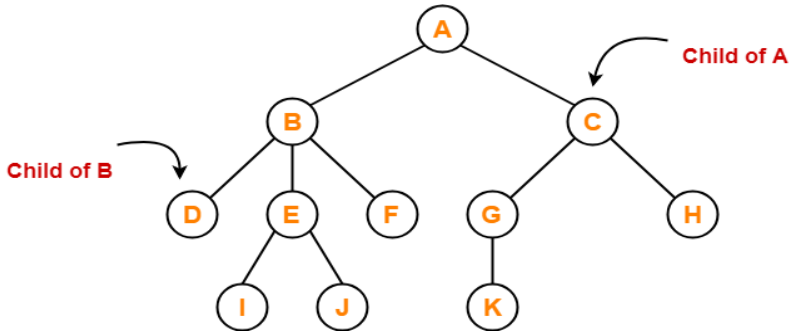
Here,

- Node A is the parent of nodes B and C
- Node B is the parent of nodes D, E and F
- Node C is the parent of nodes G and H
- Node E is the parent of nodes I and J
- Node G is the parent of node K

4. Child

- The node which is a descendant of some node is called as a **child node**.
- All the nodes except root node are child nodes.

Example



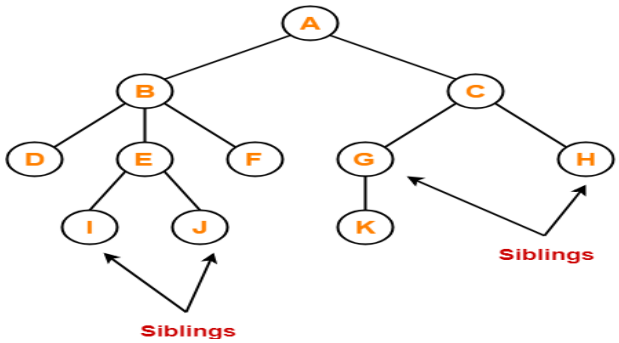
Here,

- Nodes B and C are the children of node A
- Nodes D, E and F are the children of node B
- Nodes G and H are the children of node C
- Nodes I and J are the children of node E
- Node K is the child of node G

5. Siblings

- Nodes which belong to the same parent are called as **siblings**.
- In other words, nodes with the same parent are sibling nodes.

Example



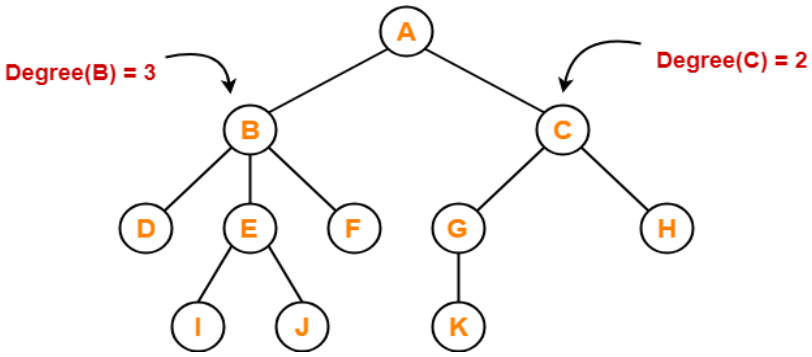
Here,

- Nodes B and C are siblings
- Nodes D, E and F are siblings
- Nodes G and H are siblings
- Nodes I and J are siblings

6. Degree

- **Degree of a node** is the total number of children of that node.
- **Degree of a tree** is the highest degree of a node among all the nodes in the tree.

Example



Here,

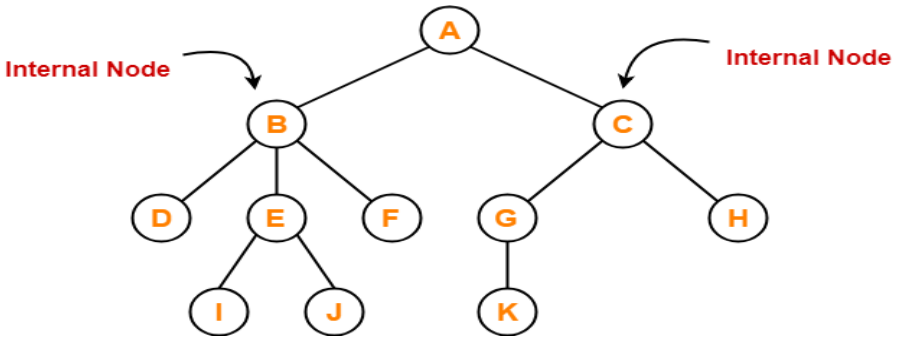
- Degree of node A = 2
- Degree of node B = 3
- Degree of node C = 2
- Degree of node D = 0
- Degree of node E = 2
- Degree of node F = 0
- Degree of node G = 1
- Degree of node H = 0
- Degree of node I = 0

- Degree of node J = 0
- Degree of node K = 0

7. Internal Node

- The node which has at least one child is called as an **internal node**.
- Internal nodes are also called as **non-terminal nodes**.
- Every non-leaf node is an internal node.

Example

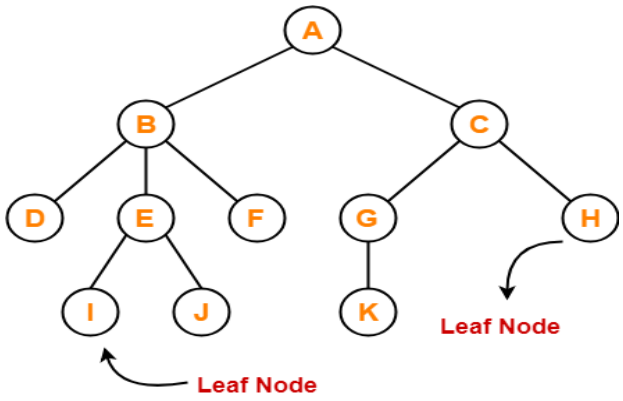


Here, nodes A, B, C, E and G are internal nodes.

8. Leaf Node

- The node which does not have any child is called as a **leaf node**.
- Leaf nodes are also called as **external nodes** or **terminal nodes**.

Example

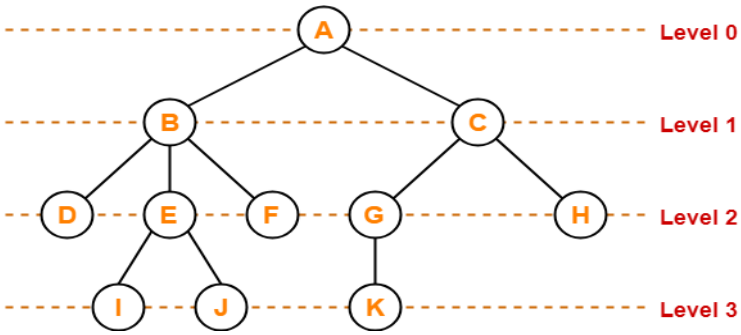


Here, nodes D, I, J, F, K and H are leaf nodes.

9. Level

- In a tree, each step from top to bottom is called as **level of a tree**.
- The level count starts with 0 and increments by 1 at each level or step.

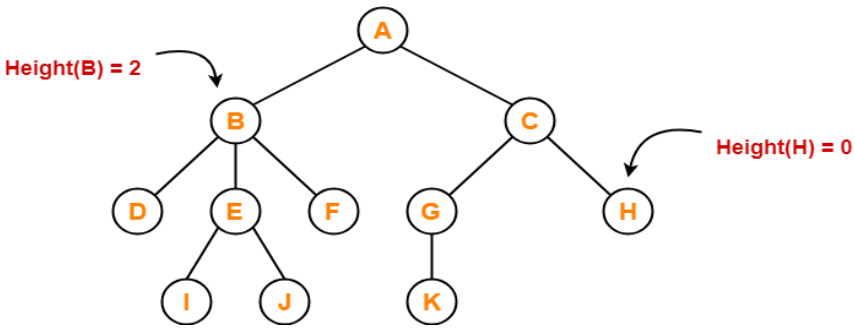
Example



10. Height

- Total number of edges that lies on the longest path from any leaf node to a particular node is called as **height of that node**.
- **Height of a tree** is the height of root node.
- Height of all leaf nodes = 0

Example



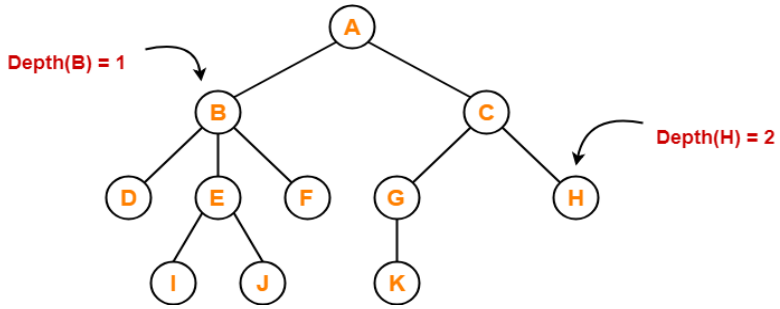
Here,

- Height of node A = 3
- Height of node B = 2
- Height of node C = 2
- Height of node D = 0
- Height of node E = 1
- Height of node F = 0
- Height of node G = 1
- Height of node H = 0
- Height of node I = 0
- Height of node J = 0
- Height of node K = 0

11. Depth

- Total number of edges from root node to a particular node is called as **depth of that node**.
- **Depth of a tree** is the total number of edges from root node to a leaf node in the longest path.
- Depth of the root node = 0
- The terms “level” and “depth” are used interchangeably.

Example



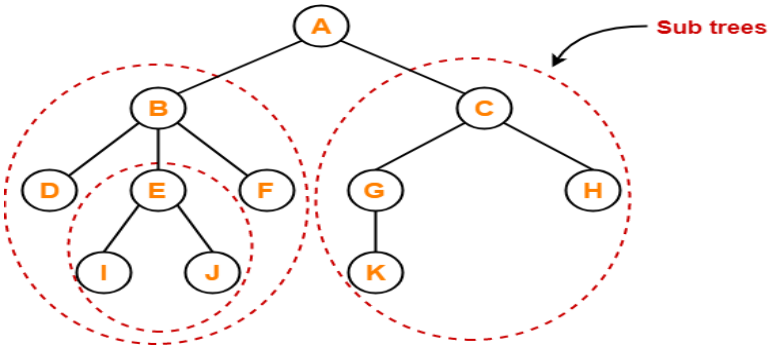
Here,

- Depth of node A = 0
- Depth of node B = 1
- Depth of node C = 1
- Depth of node D = 2
- Depth of node E = 2
- Depth of node F = 2
- Depth of node G = 2
- Depth of node H = 2
- Depth of node I = 3
- Depth of node J = 3
- Depth of node K = 3

12. Subtree

- In a tree, each child from a node forms a **subtree** recursively.
- Every child node forms a subtree on its parent node.

Example



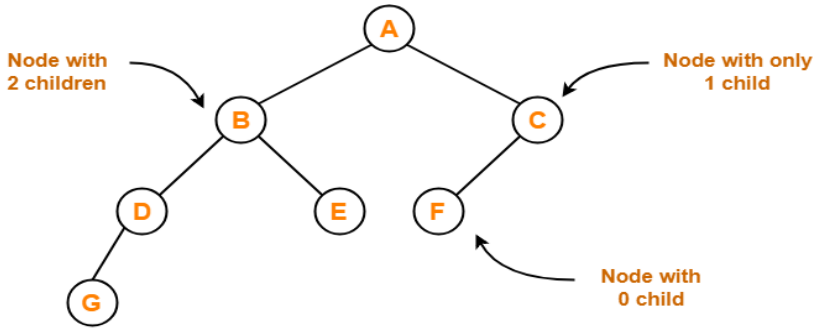
Types of Trees

1. Binary Tree
2. Binary Search Tree
3. AVL Tree
4. Red-Black Tress
5. Splay Tree

BINARY TREE

- Tree is a non-linear data structure.
- In a tree data structure, a node can have any number of child nodes.
- Binary tree is a special tree data structure in which each node can have at most 2 children.
- Thus, in a binary tree,Each node has either 0 child or 1 child or 2 children.
- One is known as a left child and the other is known as right child.

Example:



Binary Tree Example

Types of Binary Tree

1. Rooted Binary Tree
2. Full / Strictly Binary Tree

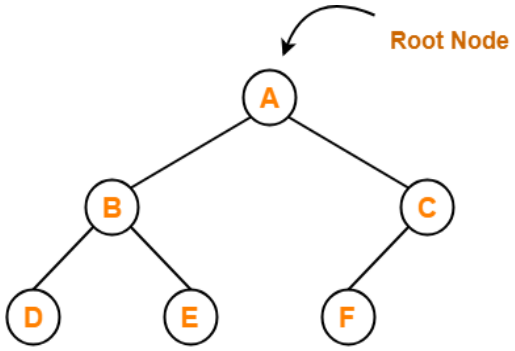
3. Complete / Perfect Binary Tree
4. Almost Complete Binary Tree
5. Skewed Binary Tree

1. Rooted Binary Tree

A **rooted binary tree** is a binary tree that satisfies the following 2 properties-

- It has a root node.
- Each node has at most 2 children.

Example

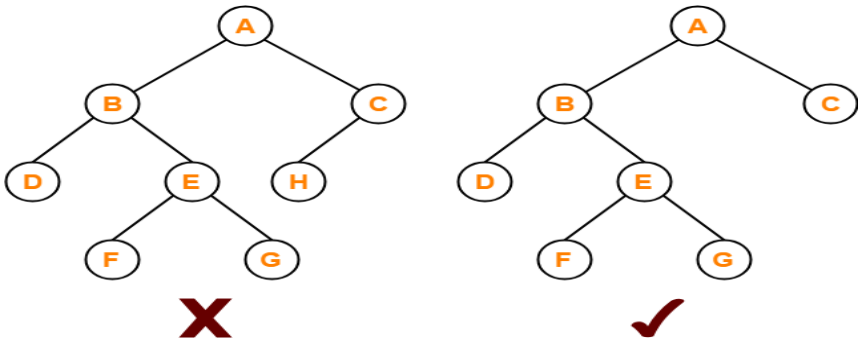


Rooted Binary Tree

2. Full / Strictly Binary Tree

- A binary tree in which every node has either 0 or 2 children is called as a **Full binary tree**.
- Full binary tree is also called as **Strictly binary tree**.

Example



Here,

- First binary tree is not a full binary tree.
- This is because node C has only 1 child.

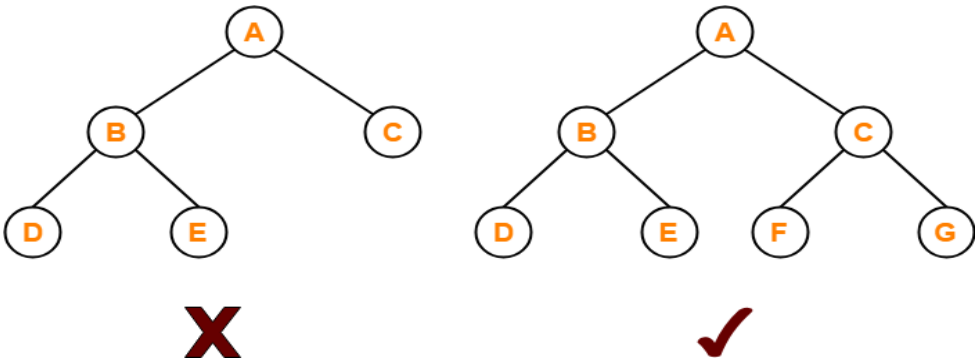
3. Complete / Perfect Binary Tree

A **complete binary tree** is a binary tree that satisfies the following 2 properties-

- Every internal node has exactly 2 children.
- All the leaf nodes are at the same level.

Complete binary tree is also called as **Perfect binary tree**.

Example



Here,

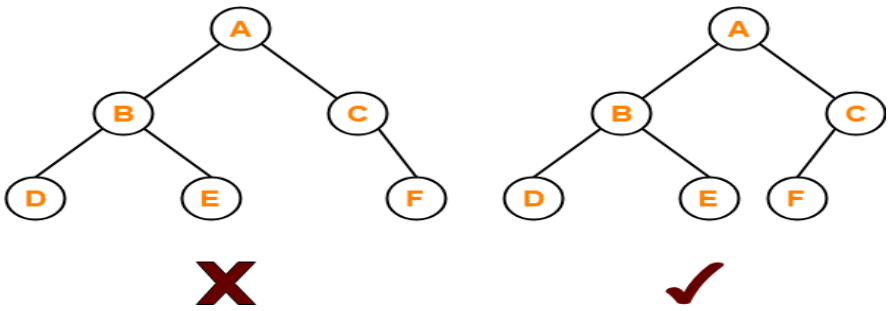
- First binary tree is not a complete binary tree.
- This is because all the leaf nodes are not at the same level.

4. Almost Complete Binary Tree

An **almost complete binary tree** is a binary tree that satisfies the following 2 properties-

- All the levels are completely filled except possibly the last level.
- The last level must be strictly filled from left to right.

Example



Here,

- First binary tree is not an almost complete binary tree.
- This is because the last level is not filled from left to right.

5. Skewed Binary Tree

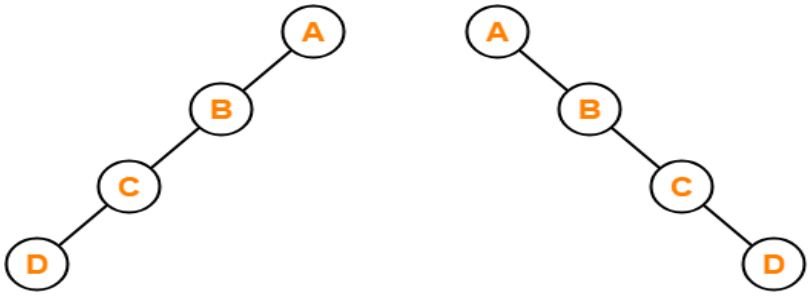
A **skewed binary tree** is a binary tree that satisfies the following 2 properties-

- All the nodes except one node has one and only one child.
- The remaining node has no child.

OR

A **skewed binary tree** is a binary tree of n nodes such that its depth is (n-1).

Example



Left Skewed Binary Tree Right Skewed Binary Tree

BINARY SEARCH TREE

- Binary tree is a special tree data structure.
- In a binary tree, each node can have at most 2 children.
- In a binary tree, nodes may be arranged in any random order.

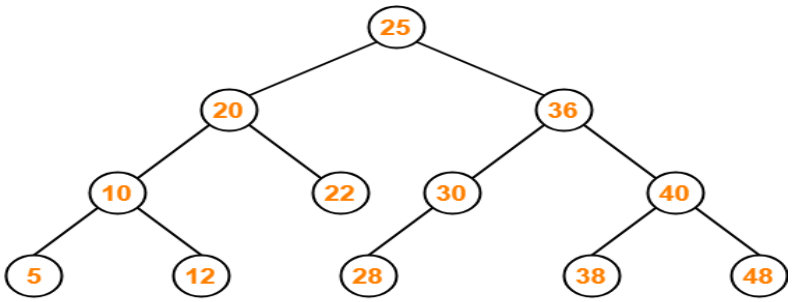
Binary Search Tree is a special kind of binary tree in which nodes are arranged in a specific order.

In a binary search tree (BST), each node contains-

- Only smaller values in its left sub tree
- Only larger values in its right sub tree

Example:

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.



Binary Search Tree

Note:

- Every binary search tree is a binary tree but every binary tree need not to be binary search tree.
- All the values in a BST must be distinct

Operations on a Binary Search Tree

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

1. **Insertion**
2. **Search**
3. **Deletion**

Insertion Operation in BST

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

Step 1 - Create a newNode with given value and set its **left** and **right** to **NULL**.

Step 2 - Check whether tree is Empty.

Step 3 - If the tree is **Empty**, then set **root** to **newNode**.

Step 4 - If the tree is **Not Empty**, then check whether the value of newNode is **smaller** or **larger** than the node (here it is root node).

Step 5 - If newNode is **smaller** than **or equal** to the node then move to its **left** child. If newNode is **larger** than the node then move to its **right** child.

Step 6- Repeat the above steps until we reach to the **leaf** node (i.e., reaches to NULL).

Step 7 - After reaching the leaf node, insert the newNode as **left child** if the newNode is **smaller or equal** to that leaf node or else insert it as **right child**.

Example:
Construct a Binary Search Tree (BST) for the following sequence of numbers-
50, 70, 60, 20, 90, 10, 40, 100

Solution:

When elements are given in a sequence,

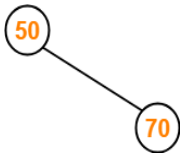
- Always consider the first element as the root node.
- Consider the given elements and insert them in the BST one by one.

The binary search tree will be constructed as explained below-

Insert 50

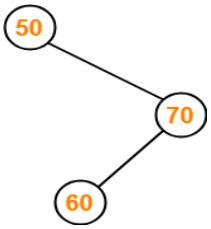
Insert 70

- As 70 > 50, so insert 70 to the right of 50.



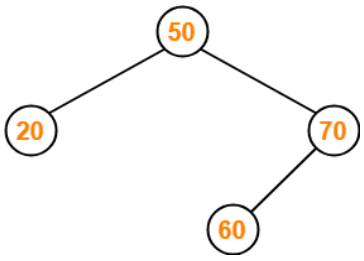
Insert 60

- As 60 > 50, so insert 60 to the right of 50.
- As 60 < 70, so insert 60 to the left of 70.



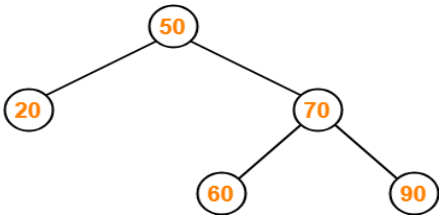
Insert 20

- As 20 < 50, so insert 20 to the left of 50.



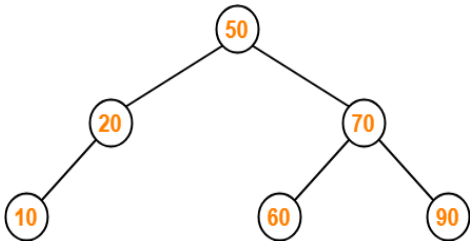
Insert 90

- As 90 > 50, so insert 90 to the right of 50.
- As 90 > 70, so insert 90 to the right of 70.



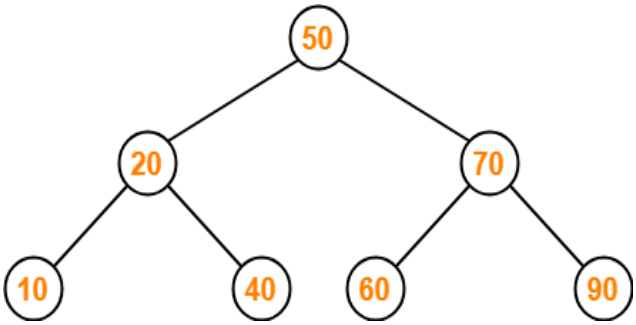
Insert 10

- As 10 < 50, so insert 10 to the left of 50.
- As 10 < 20, so insert 10 to the left of 20.



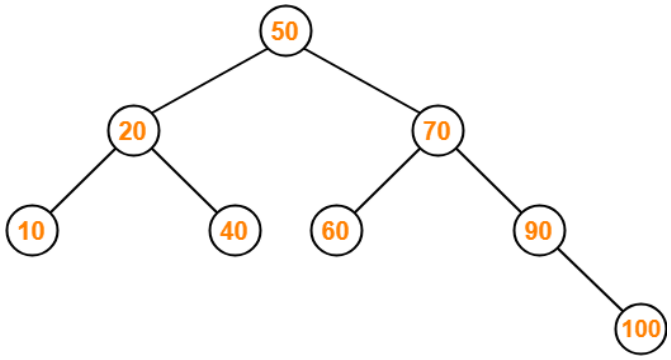
Insert 40

- As 40 < 50, so insert 40 to the left of 50.
- As 40 > 20, so insert 40 to the right of 20.



Insert 100

- As 100 > 50, so insert 100 to the right of 50.
- As 100 > 70, so insert 100 to the right of 70.
- As 100 > 90, so insert 100 to the right of 90.



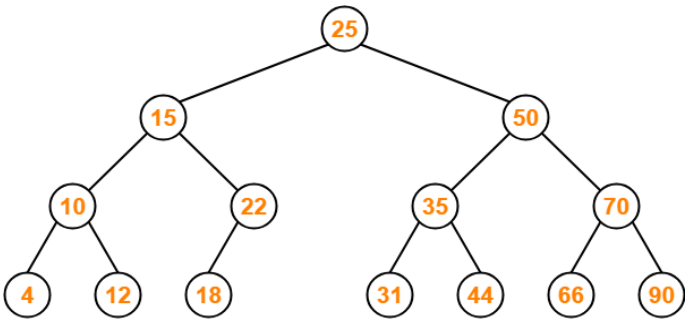
Binary Search Tree

Search Operation in BST

The search operation is performed as follows...

Step 1 - Read the search element from the user.
Step 2 - Compare the search element with the value of root node in the tree.
Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function
Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.
Step 5 - If search element is smaller, then continue the search process in left subtree.
Step 6- If search element is larger, then continue the search process in right subtree.
Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node
Step 8 - If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.
Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Example:
Consider key = 45 has to be searched in the given BST-



Binary Search Tree

- We start our search from the root node 25.
- As 45 > 25, so we search in 25's right subtree.
- As 45 < 50, so we search in 50's left subtree.
- As 45 > 35, so we search in 35's right subtree.
- As 45 > 44, so we search in 44's right subtree but 44 has no subtrees.
- So, we conclude that 45 is not present in the above BST.

Deletion Operation in BST

Deleting a node from Binary search tree includes following three cases...

Case 1: Deleting a Leaf node (A node with no children)

Case 2: Deleting a node with one child

Case 3: Deleting a node with two children

Case 1: Deleting a leaf node

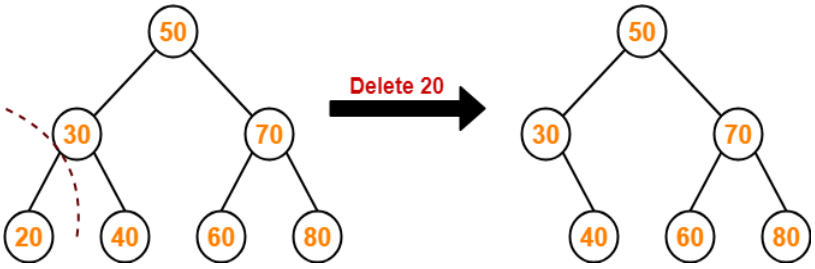
We use the following steps to delete a leaf node from BST...

Step 1 - Find the node to be deleted using **search operation**

Step 2 - Delete the node using **free** function (If it is a leaf) and terminate the function.

Example:

Consider the following example where node with value = 20 is deleted from the BST-



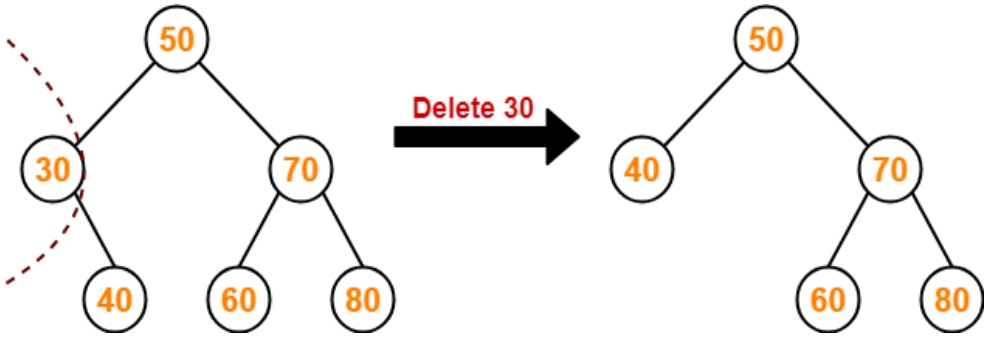
Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

Step 1 - Find the node to be deleted using **search operation**
Step 2 - If it has only one child then create a link between its parent node and child node.
Step 3 - Delete the node using **free** function and terminate the function.

Example:

Consider the following example where node with value = 30 is deleted from the BST



Case 3: Deleting a node with two children

We use the following steps to delete a node with two children from BST...

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

Example:

A node with two children may be deleted from the BST in the following two ways-

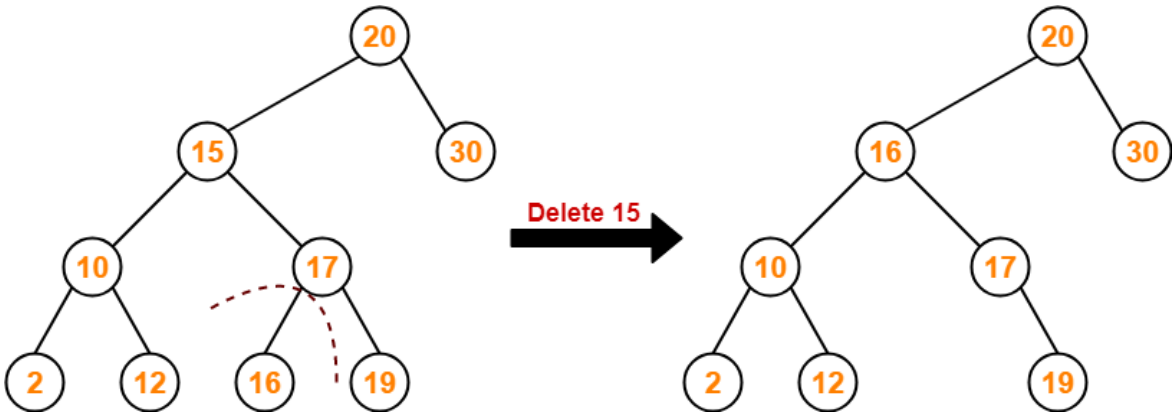
Method-1:

- Visit to the right subtree of the deleting node.
- Pluck the least value element called as inorder successor.

- Replace the deleting element with its inorder successor.

Example

Consider the following example where node with value = 15 is deleted from the BST-

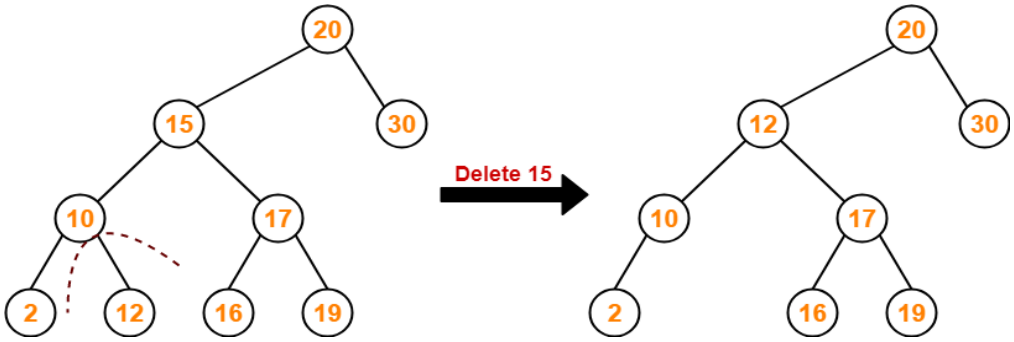


Method-2:

- Visit to the left subtree of the deleting node.
- Pluck the greatest value element called as inorder predecessor.
- Replace the deleting element with its inorder predecessor.

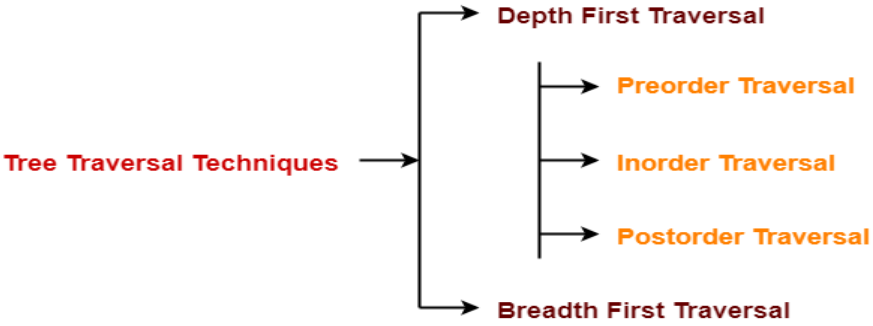
Example

Consider the following example where node with value = 15 is deleted from the BST



TREE TRAVERSAL

Tree Traversal refers to the process of visiting each node in a tree data structure exactly once.
Various tree traversal techniques are



DEPTH FIRST TRAVERSAL

Following three traversal techniques fall under Depth First Traversal-

1. Preorder Traversal
2. Inorder Traversal
3. Postorder Traversal

1. Preorder Traversal

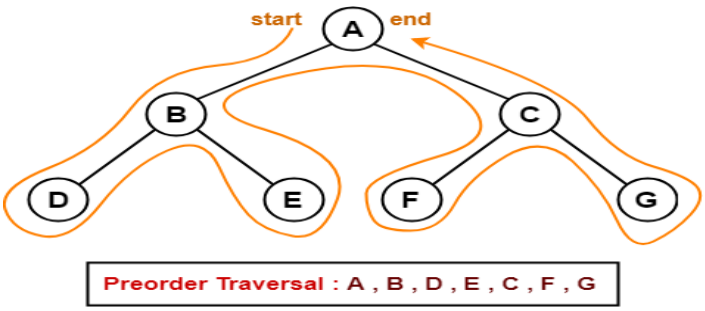
Algorithm

1. Visit the root
2. Traverse the left sub tree i.e. call Preorder (left sub tree)
3. Traverse the right sub tree i.e. call Preorder (right sub tree)

Root → Left → Right

Example

Consider the following example



2. Inorder Traversal

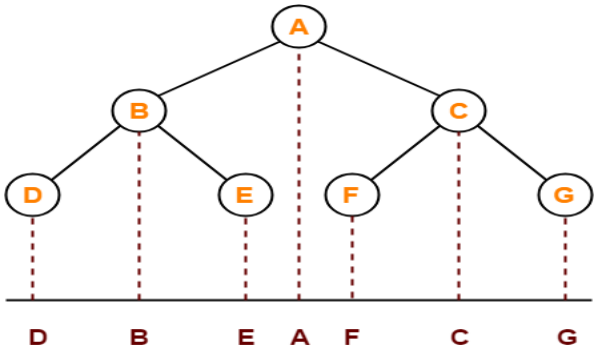
Algorithm

1. Traverse the left sub tree i.e. call Inorder (left sub tree)
2. Visit the root
3. Traverse the right sub tree i.e. call Inorder (right sub tree)

Left → Root → Right

Example

Consider the following example



Inorder Traversal : D , B , E , A , F , C , G

3. Postorder Traversal

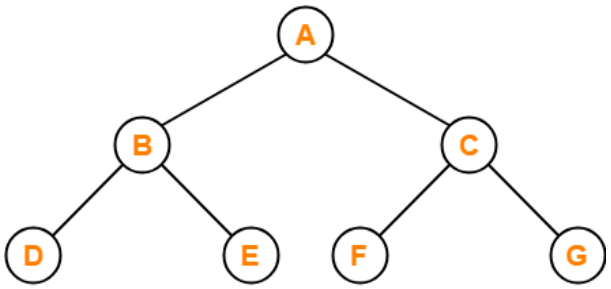
Algorithm

1. Traverse the left sub tree i.e. call Postorder (left sub tree)
2. Traverse the right sub tree i.e. call Postorder (right sub tree)
3. Visit the root

Left → Right → Root

Example

Consider the following example

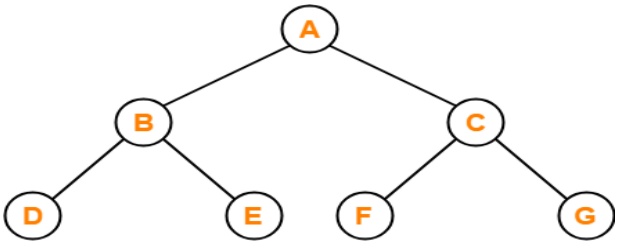


Postorder Traversal : D , E , B , F , G , C , A

BREADTH FIRST TRAVERSAL

- Breadth First Traversal of a tree prints all the nodes of a tree level by level.
- Breadth First Traversal is also called as **Level Order Traversal**.

Example



Level Order Traversal : A , B , C , D , E , F , G

AVL TREE (Adelson, Velski & Landis)

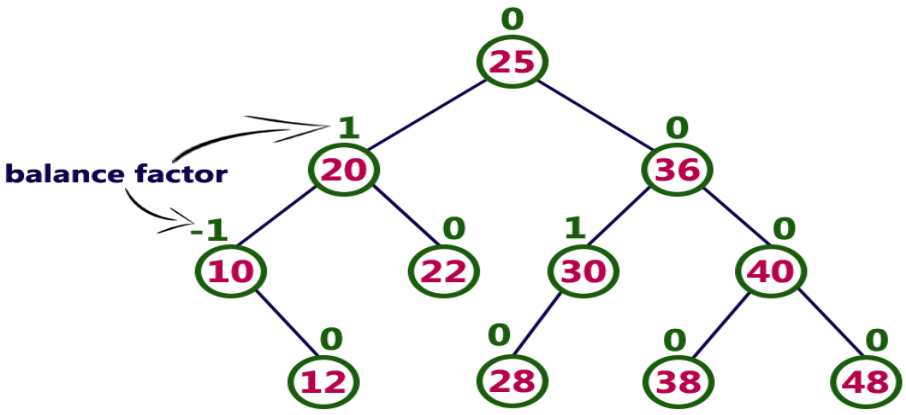
An AVL tree is a self balancing Binary Search Tree(BST). In an AVL tree, balance factor of every node is either -1, 0 or +1.

Balance Factor

Balance factor of a node in an AVL tree is the difference between the height of the left subtree and the height of the right subtree of that node.

Balance Factor = Height of Left Subtree - Height of Right Subtree

Example of AVL Tree

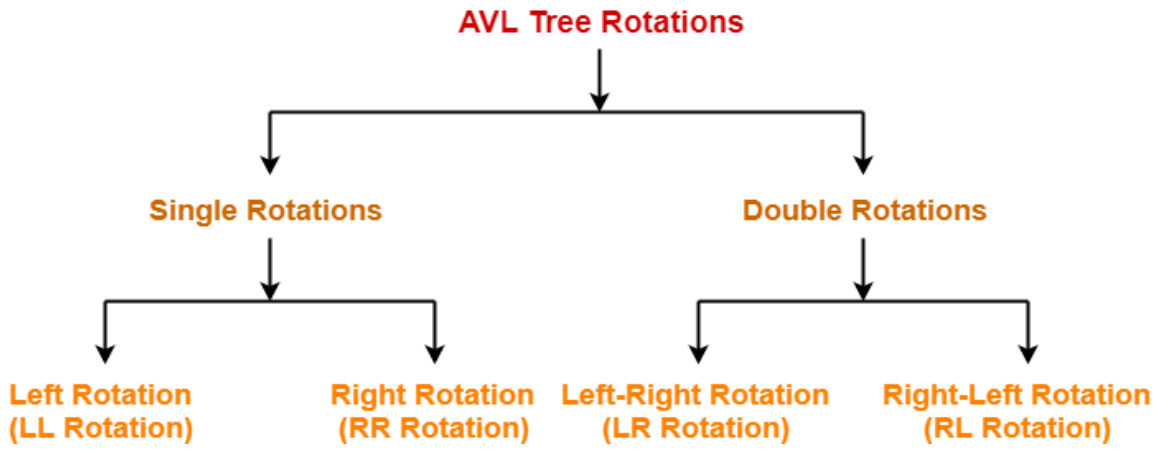


The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

Note: Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL tree.

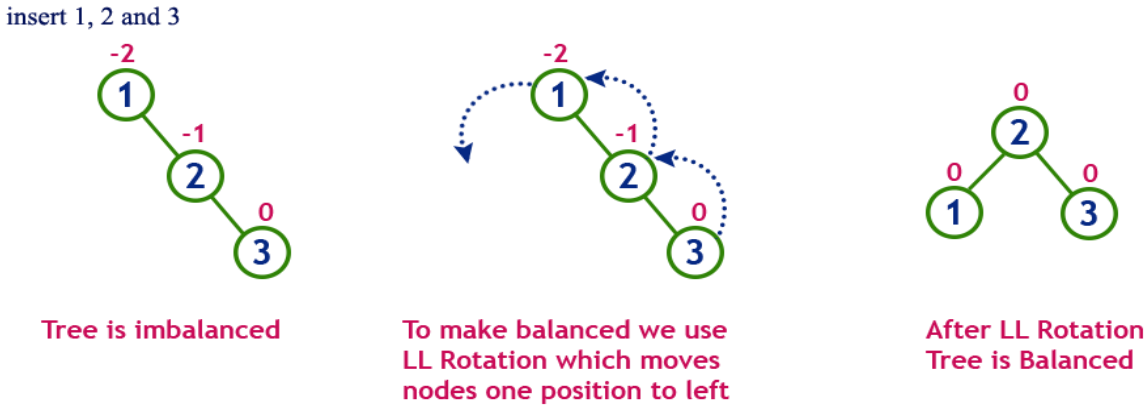
AVL Rotations

- If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques
- Rotation is the process of moving nodes either to left or to right to make the tree balanced.
- There are **four** rotations and they are classified into **two** types.



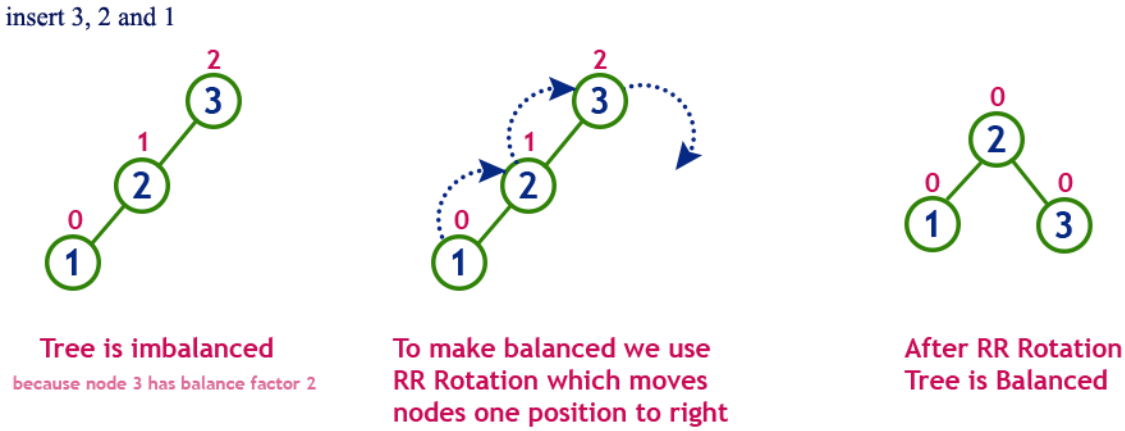
Left Rotation (LL Rotation)

In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree.



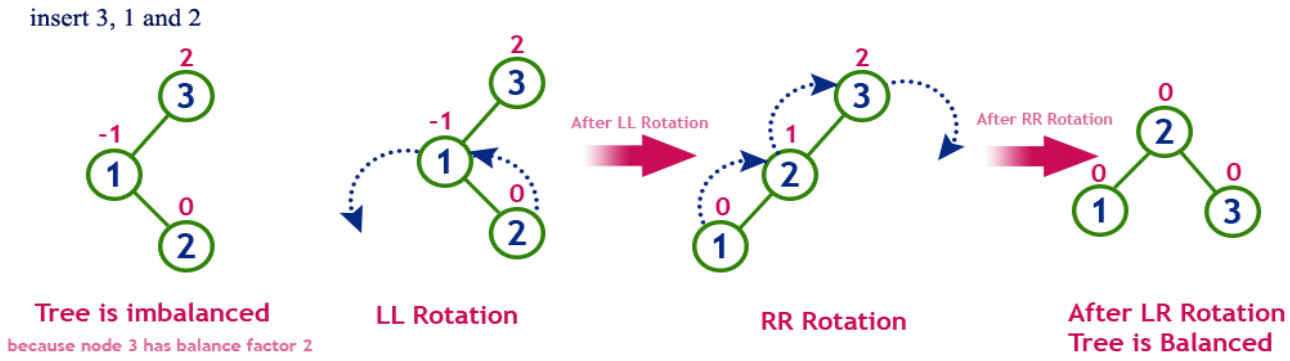
Single Right Rotation (RR Rotation)

In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree.



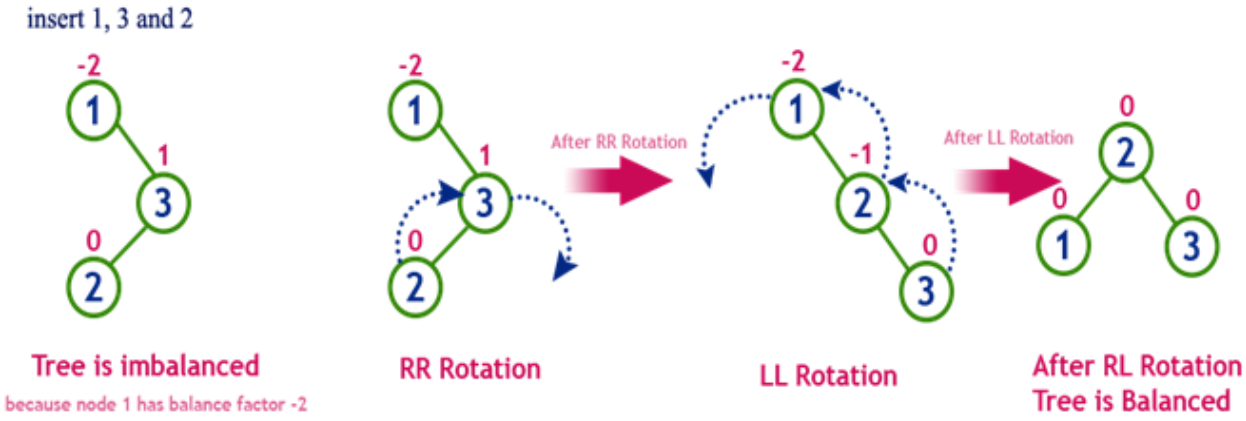
Left Right Rotation (LR Rotation)

The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree...



Right Left Rotation (RL Rotation)

The RL Rotation is sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree.



Operations on an AVL Tree

The following operations are performed on AVL tree...

1. Insertion
2. Search
3. Deletion

Insertion Operation in AVL Tree

In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

- **Step 1 -** Insert the new element into the tree using Binary Search Tree insertion logic.
- **Step 2 -** After insertion, check the **Balance Factor** of every node.
- **Step 3 -** If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
- **Step 4 -** If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation.

Example:

Construct AVL Tree for the following sequence of numbers-

50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48

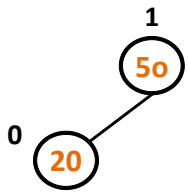
Solution:

Insert 50

Tree is Balanced

Insert 20

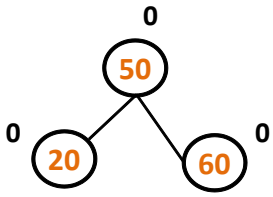
- As 20 < 50, so insert 20 in 50's left sub tree.



Tree is Balanced

Insert 60

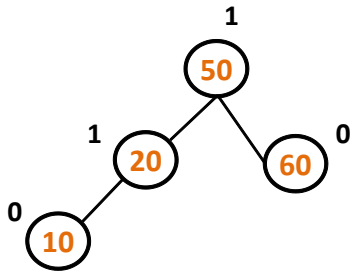
- As 60 > 50, so insert 60 in 50's right sub tree.



Tree is Balanced

Insert 10

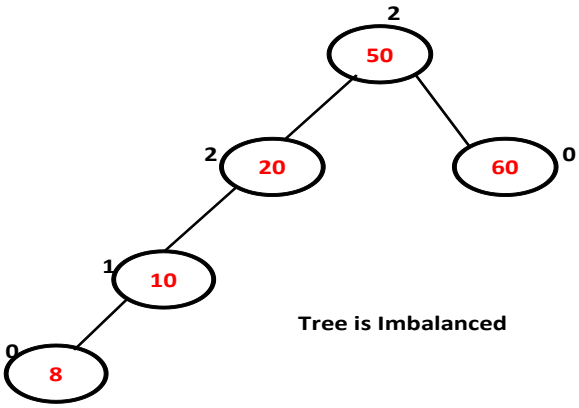
- As 10 < 50, so insert 10 in 50's left sub tree.
- As 10 < 20, so insert 10 in 20's left sub tree.



Tree is Balanced

Insert 8

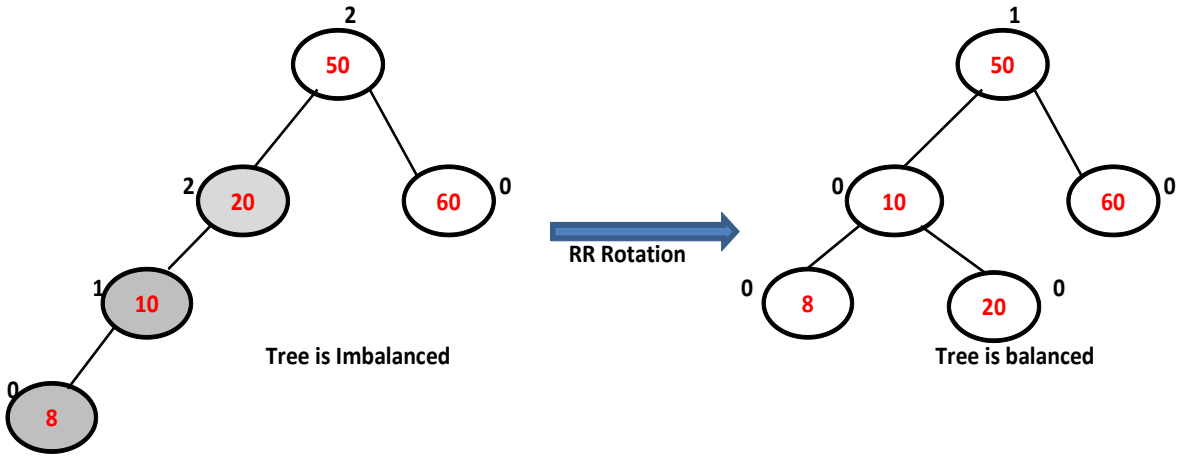
- As 8 < 50, so insert 8 in 50's left sub tree.
- As 8 < 20, so insert 8 in 20's left sub tree.
- As 8 < 10, so insert 8 in 10's left sub tree.



To balance the tree,

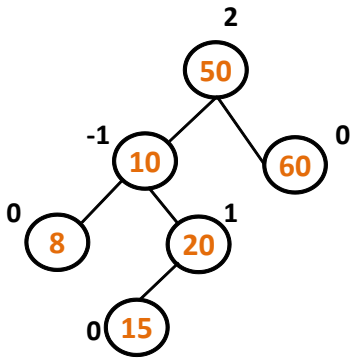
- Find the first imbalanced node on the path from the newly inserted node (node 8) to the root node.
- The first imbalanced node is node 20.
- Now, count three nodes from node 20 in the direction of leaf node.
- Then, use AVL tree rotation to balance the tree.

Following this, we have-



Step-06: Insert 15

- As 15 < 50, so insert 15 in 50's left sub tree.
- As 15 > 10, so insert 15 in 10's right sub tree.
- As 15 < 20, so insert 15 in 20's left sub tree.

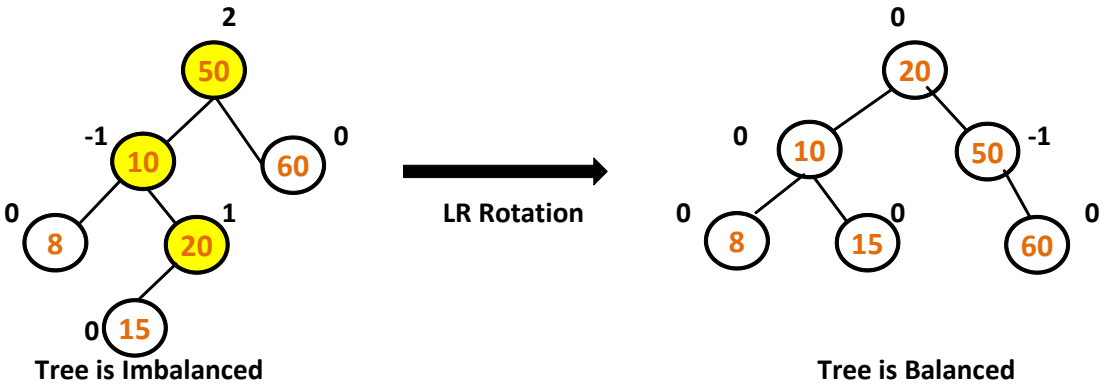


Tree is Imbalanced

To balance the tree,

- Find the first imbalanced node on the path from the newly inserted node (node 15) to the root node.
- The first imbalanced node is node 50.
- Now, count three nodes from node 50 in the direction of leaf node.
- Then, use AVL tree rotation to balance the tree.

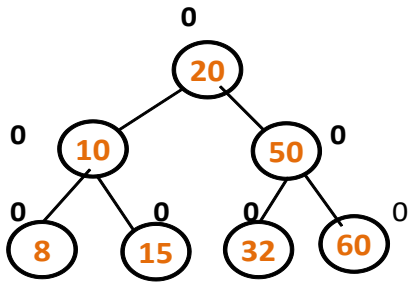
Following this, we have-



Tree is Imbalanced **Tree is Balanced**

Step-07: Insert 32

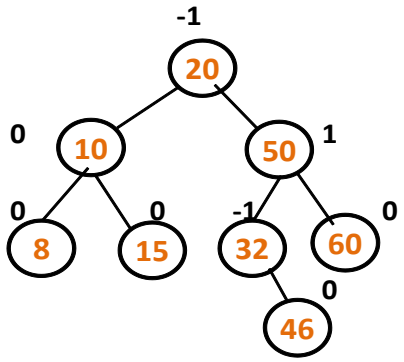
- As 32 > 20, so insert 32 in 20's right sub tree.
- As 32 < 50, so insert 32 in 50's left sub tree.



Tree is Balanced

Step-08: Insert 46

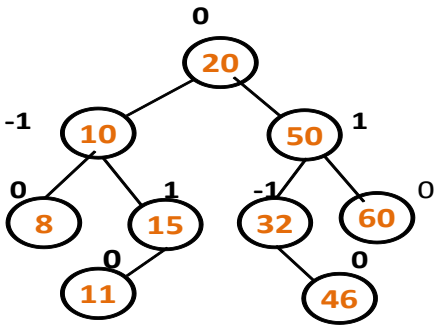
- As 46 > 20, so insert 46 in 20's right sub tree.
- As 46 < 50, so insert 46 in 50's left sub tree.
- As 46 > 32, so insert 46 in 32's right sub tree.



Tree is Balanced

Step-09: Insert 11

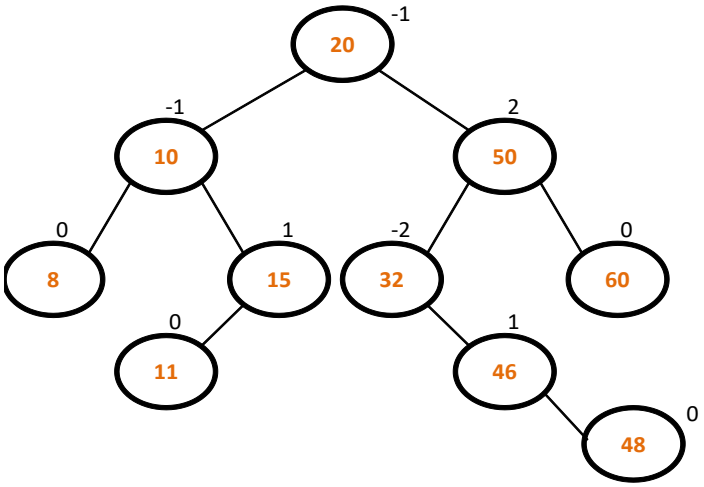
- As 11 < 20, so insert 11 in 20's left sub tree.
- As 11 > 10, so insert 11 in 10's right sub tree.
- As 11 < 15, so insert 11 in 15's left sub tree.



Tree is Balanced

Step-10: Insert 48

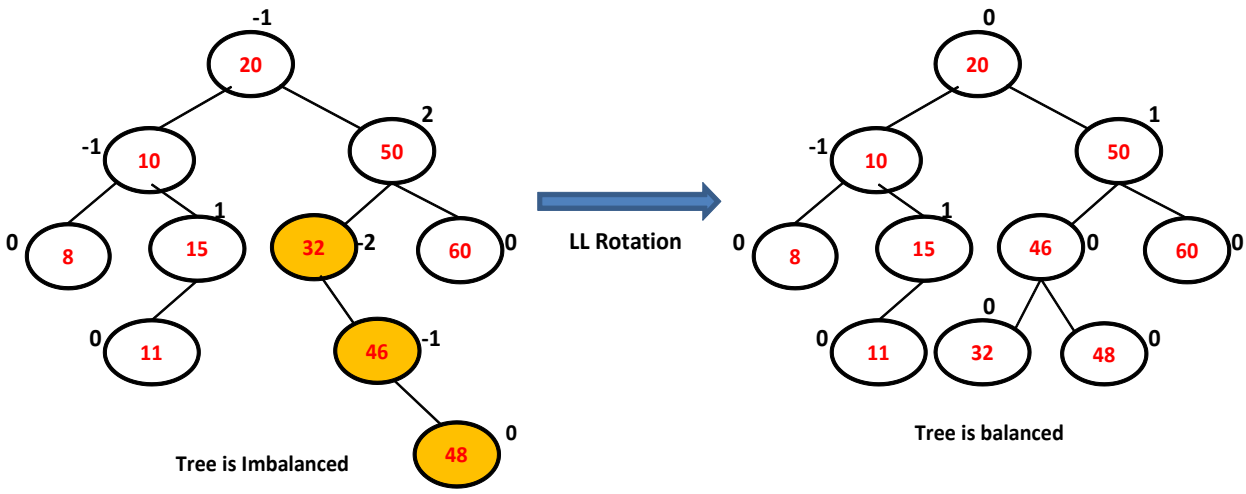
- As 48 > 20, so insert 48 in 20's right sub tree.
- As 48 < 50, so insert 48 in 50's left sub tree.
- As 48 > 32, so insert 48 in 32's right sub tree.
- As 48 > 46, so insert 48 in 46's right sub tree.



To balance the tree,

- Find the first imbalanced node on the path from the newly inserted node (node 48) to the root node.
- The first imbalanced node is node 32.
- Now, count three nodes from node 32 in the direction of leaf node.
- Then, use AVL tree rotation to balance the tree.

Following this, we have-



Deletion Operation in an AVL Tree

Deletion in an AVL Tree

- Deletion in an AVL tree is similar to that in a BST.
- Deletion of a node tends to disturb the *balance factor*. Thus to balance the tree, we again use the Rotation mechanism.

Deletion in AVL tree consists of two steps:

- **Removal of the node:** The given node is removed from the tree structure. The node to be removed can either be a leaf or an internal node.
- **Re-balancing of the tree:** The elimination of a node from the tree can cause disturbance to the balance factor of certain nodes. Thus it is important to re-balance *the nodes;* since the balance factor is the primary aspect that ensures the tree is an AVL Tree.

Deleting a node from Binary search tree includes following three cases...

Case 1: Deleting a Leaf node (A node with no children)

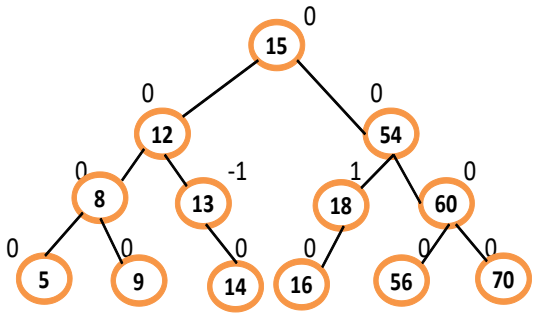
Case 2: Deleting a node with one child

Case 3: Deleting a node with two children

Note: There are certain points that must be kept in mind during a deletion process.

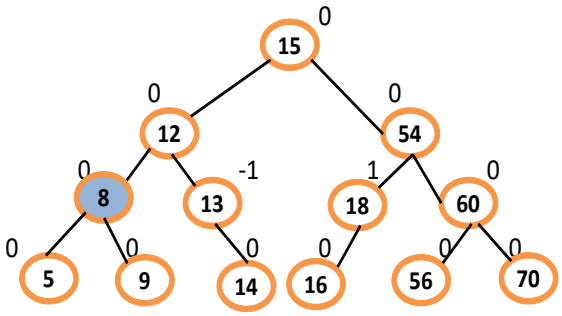
- If the node to be deleted is a leaf node, it is simply removed from the tree.
- If the node to be deleted has one child node, the child node is replaced with the node to be deleted simply.
- If the node to be deleted has two child nodes then,
  - Either replace the node with it's **inorder predecessor** , i.e, **the largest element of the left sub tree.**
  - Or replace the node with it's **inorder successor** , i.e, **the smallest element of the right sub tree.**

Example:



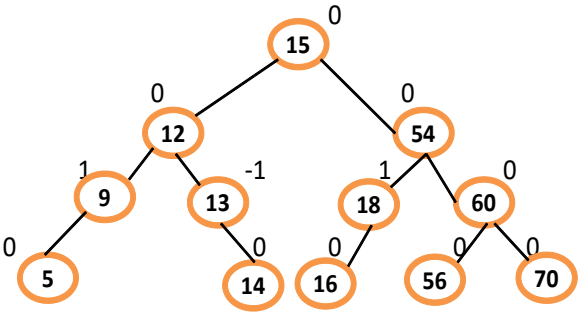
Step 1:

- The node to be deleted from the tree is **8**.
- If we observe it is the parent node of the node **5** and **9.**
- Since the node 8 has two children it can be replaced by either of it's child nodes.



Step 2:

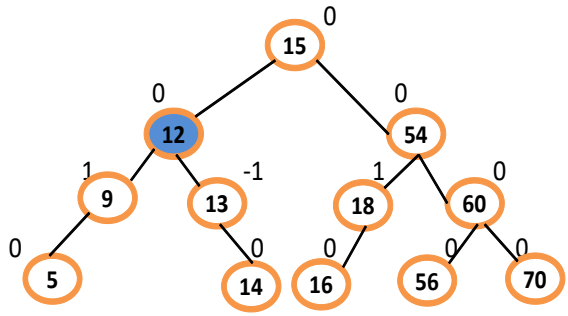
- The node **8** is deleted from the tree.
- As the node is deleted we replace it with either of it's children nodes.
- Here we **replaced** the node with the **inorder successor** , i.e, **9.**
- Again we check the balance factor for each node.



Step 3:

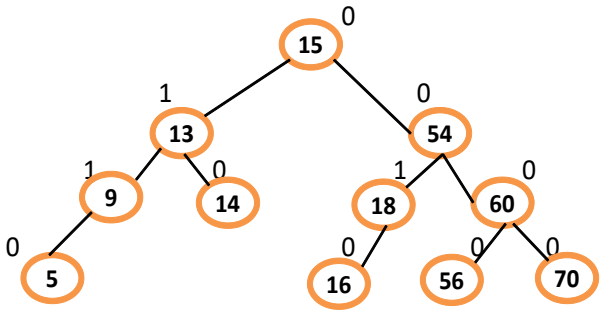
- Now The next element to be deleted is **12**.
- If we observe, we can see that the node 12 has a left subtree and a right subtree.
- We again can replace the node by either it's inorder successor or inorder predecessor.

- In this case we have replaced it by the inorder successor.



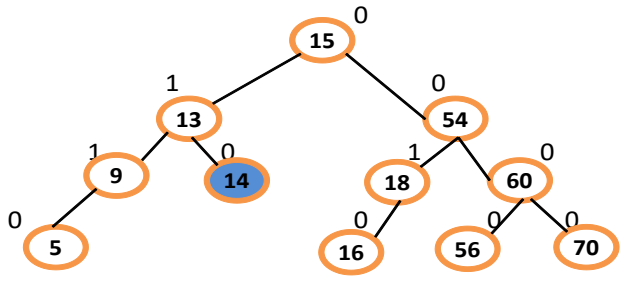
Step 4:

- The node **12** is deleted from the tree.
- Since we have replaced the node with the inorder successor, the tree structure looks like shown in the image.
- After removal and replacing check for the balance factor of each node of the tree.



Step 5:

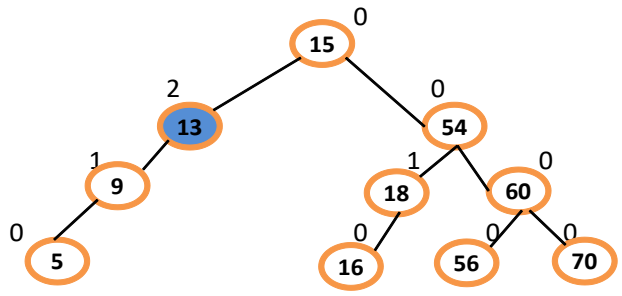
- The next node to be eliminated is **14**.
- It can be seen clearly in the image that 14 is a leaf node.
- Thus it can be eliminated easily from the tree.



Step 6:

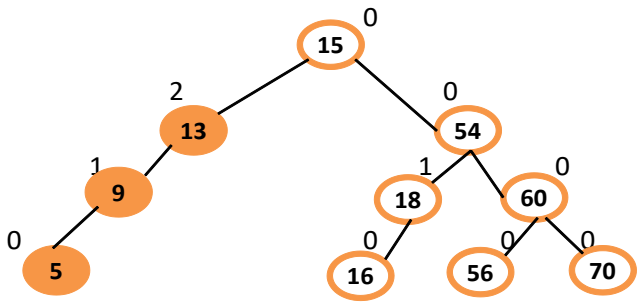
- As the node **14** is deleted, we check the balance factor of all the nodes.
- We can see the balance factor of the node **13** is **2**.

- This violates the terms of the AVL tree thus we need to balance it using the *rotation mechanism*.



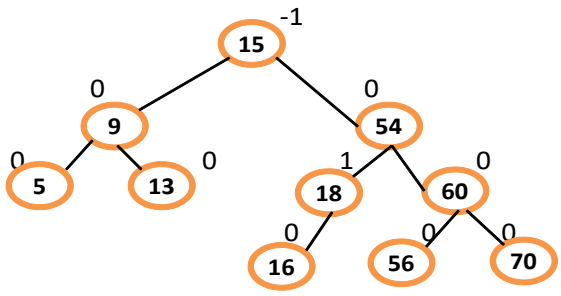
Step 7:

- In order to balance the tree, we identify the rotation mechanism to be applied.
- Here we need to use ***RR Rotation.***
- The nodes involved in the rotation is shown as follows.



Step 8:

- The nodes are rotated and the tree satisfies the conditions of an AVL tree.
- The final structure of the tree is shown as follows.
- We can see all the nodes have their balance factor as **'0' , '1'** and **'-1'.**

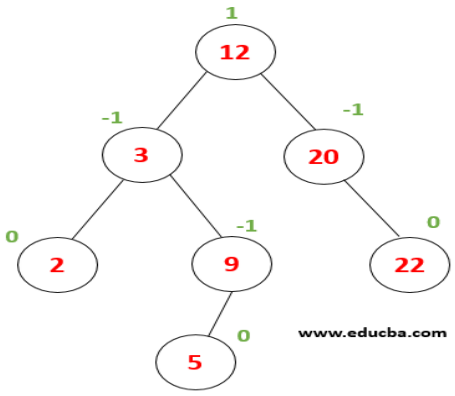
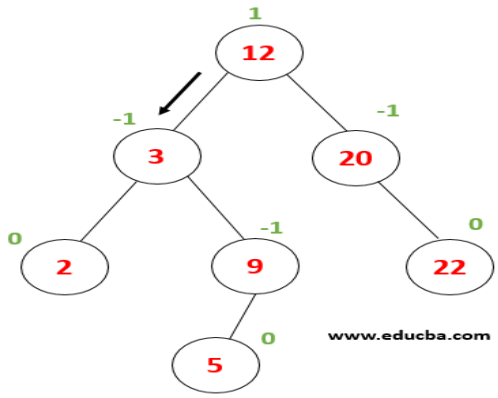
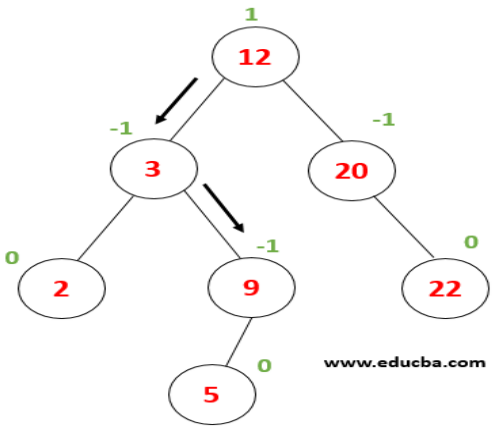


Search Opearation of AVL Tree

This operation is similar to performing a search in Binary Search Tree. Steps followed are as below:

1. Read the element provided by the user say x.
2. Compare the element from the root, if it is the same then exit otherwise go to the next step.
3. If x<root element: go to left child, and compare again.
4. Else go to the right child and compare again.
5. Follow processes B and C until you find the element and exit.

Example:

	Consider this Tree, where we need to perform a search for node value 9. First- let x=9, x< root value (12) then, the value must be in the left subtree of the root element.
	Now x is compared with node value 3 x> 3 thus we must proceed towards the right subtree.
	Now x is compared with node (9) , where 9 == 9 returns true. Thus, element searching completes in the tree.

RED – BLACK TREE DATASTRUCTURE

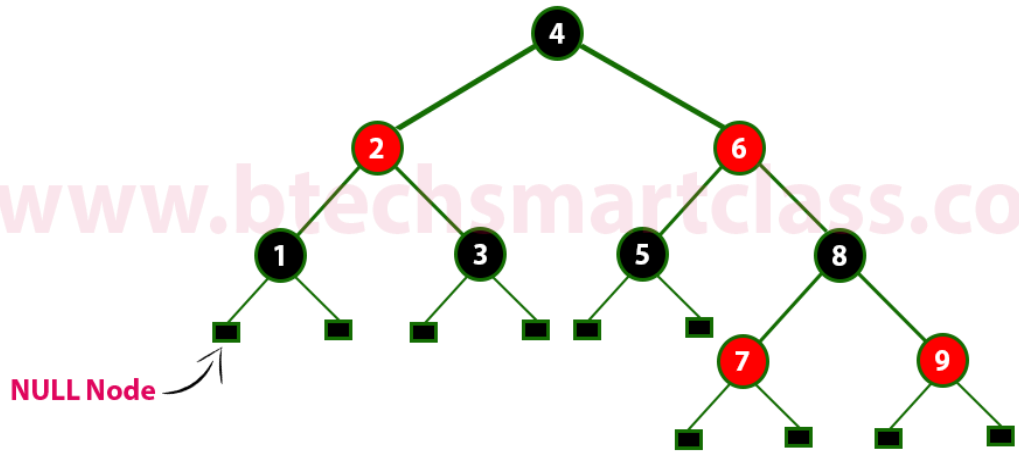
Red Black Tree is a Binary Search Tree in which every node is colored either RED or BLACK.

In Red Black Tree, the color of a node is decided based on the properties of Red-Black Tree. Every Red Black Tree has the following properties.

Properties of Red Black Tree

1. Red - Black Tree must be a Binary Search Tree.
2. The ROOT node must be colored BLACK.
3. The children of Red colored node must be colored BLACK. (There should not be two consecutive RED nodes).
4. In all the paths of the tree, there should be same number of BLACK colored nodes.
5. Every new node must be inserted with RED color.
6. Every leaf (e.i. NULL node) must be colored BLACK.

Example



The above tree is a Red-Black tree where every node is satisfying all the properties of Red-Black Tree.

Note: Every Red Black Tree is a binary search tree but every Binary Search Tree need not be Red Black tree.

Insertion into RED BLACK Tree

- In a Red-Black Tree, every new node must be inserted with the color RED.
- The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree. But it is inserted with a color property.
- After every insertion operation, we need to check all the properties of Red-Black Tree.

- If all the properties are satisfied then we go to next operation otherwise we perform the following operation to make it Red Black Tree.

    1. **Recolor(Change Grand parent as Red, Siblings as Black)**
    2. **Rotation(Same side)**
    3. **Rotation followed by Recolor(opposite side)**

The insertion operation in Red Black tree is performed using the following steps...

- **Step 1 -** Check whether tree is Empty.
- **Step 2 -** If tree is empty then insert the **newNode** as Root node with color **RED**
- **Step 3 -**If Root node is Red,color(Root) is changed to black and exit from the operation.
- **Step 4 -** If tree is not Empty then insert the newNode as leaf node with color Red.
- **Step 5 -** If the parent of newNode is Black then exit from the operation.
- **Step 6 -** If the parent of newNode is Red then check the color of parentnode's sibling of newNode.
- **Step 7 -** If it is colored Black or NULL then

  Case 1: (Same Side)

  Recolor

  Opposite side rotation

  Case 2: (Opposite Side)

  One Side Rotate

  Perform Case 1

- **Step 8 -** If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

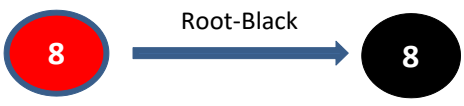
Example:

Create a RED BLACK Tree by inserting following sequence of elements

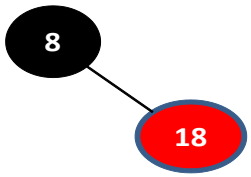
8,18,5,15,17,25,40 & 80

Insert 8

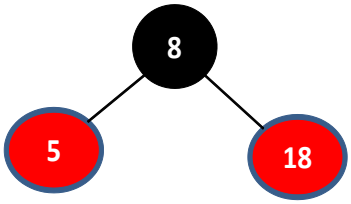
- Tree is empty.So insert newNode as Root node with Red color.
- If Root node is Red,color(Root) is changed to black and exit from the operation.



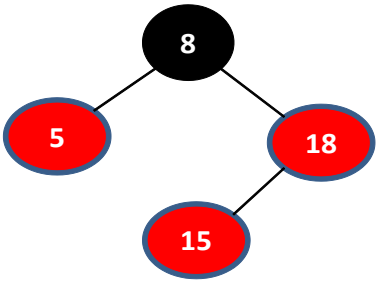
Insert 18
Tree is not empty.So insert newNode with red color.



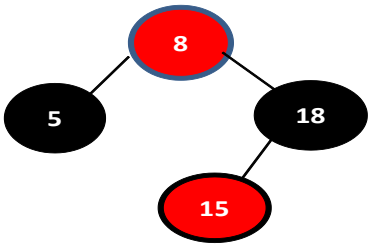
Insert 5
Tree is not empty.So insert newNode with red color.

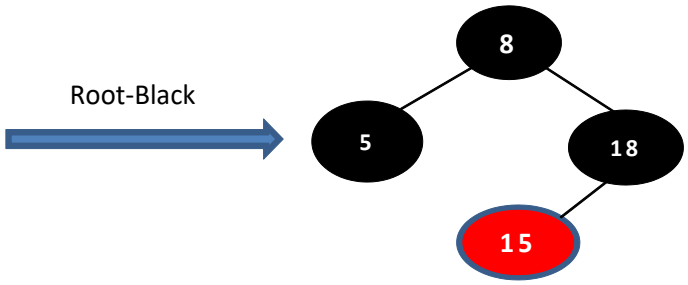


Insert 15
Tree is not empty.So insert newNode with red color.



- Here there are two consecutive Red nodes(18 & 15).
- The newNode's parent sibling color is Red
- So we use RECOLOR to make it RED BLACK Tree.



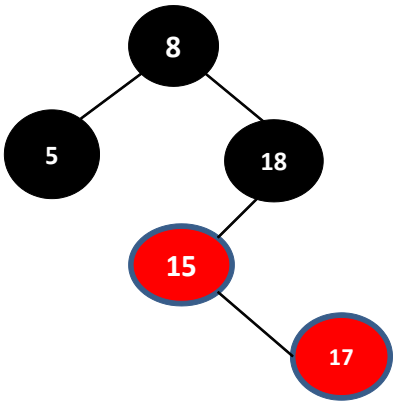


After RECOLOR operation,the tree is satisfying all RED BLACK Tree properties.

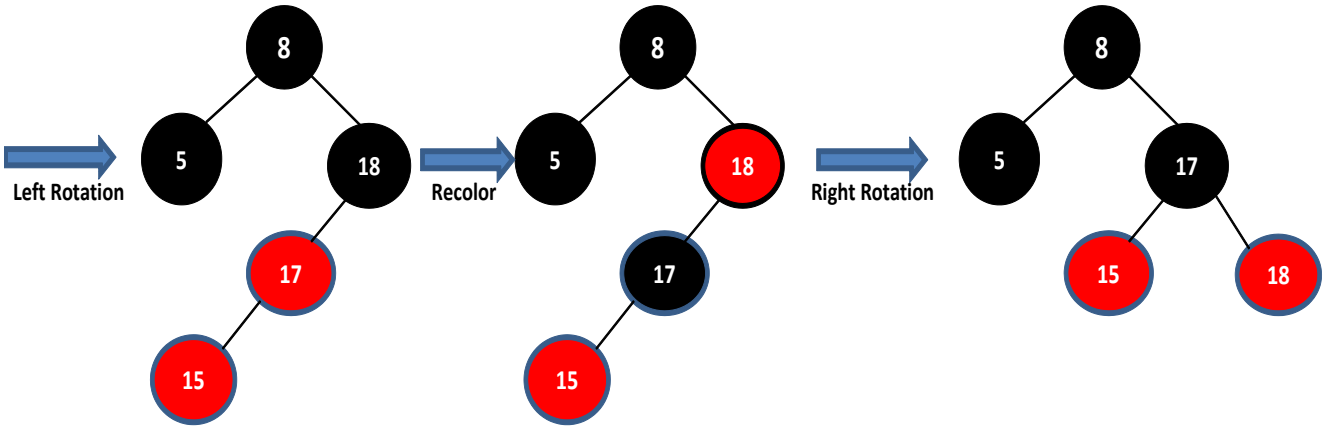
Insert 17

Tree is not empty.So insert newNode with red color.

Here there are two consecutive Red nodes(18 & 15).

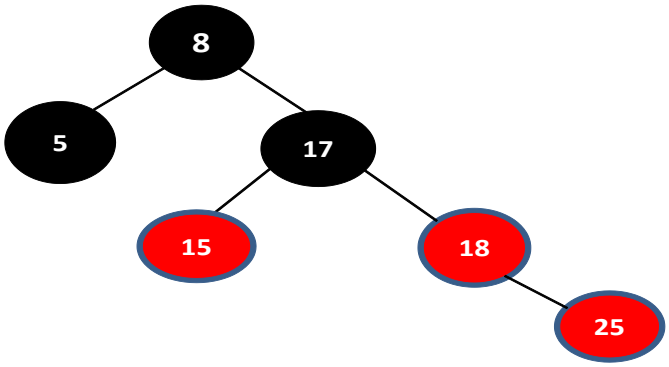


- Here there are two consecutive Red nodes(15 & 17).
- The newNode's parent sibling color is NULL.So we need rotation.
- Here,we need LR Rotaton and RECOLOR.

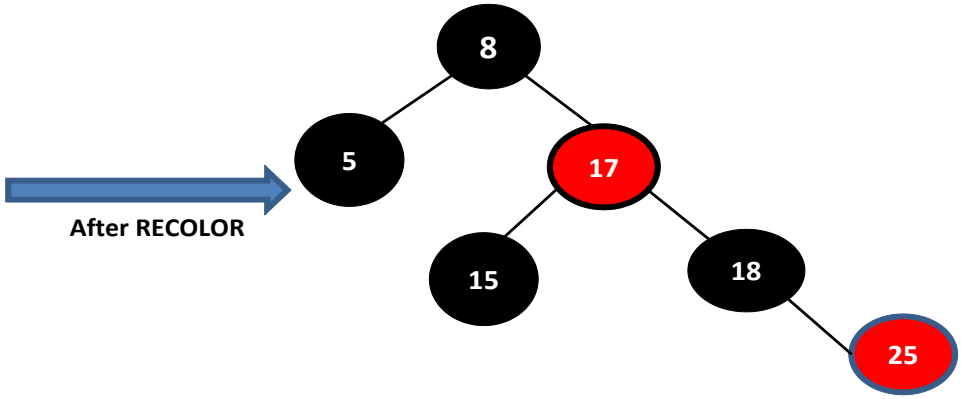


Insert 25

Tree is not empty.So insert newNode with red color.



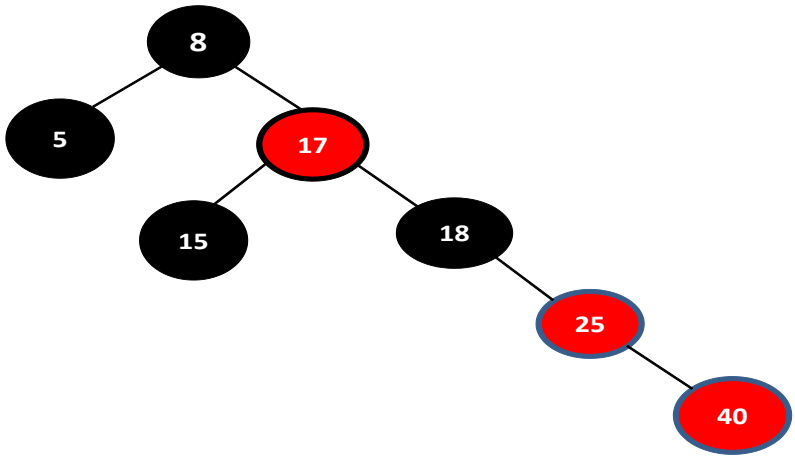
- Here there are two consecutive Red nodes(18 & 25).
- The newNode's parent sibling color is Red and parent's parent is not root node.
- So we use RECOLOR and Recheck.



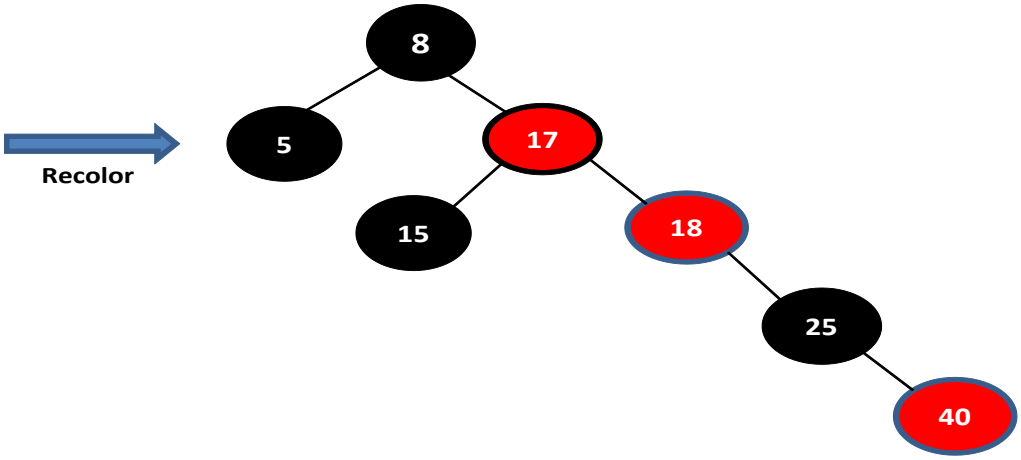
After RECOLOR operation,the tree is satisfying all RED BLACK Tree properties.

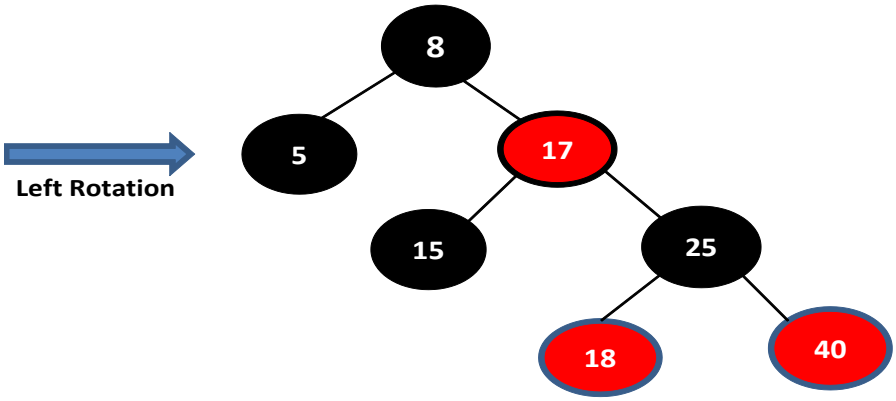
Insert 40

Tree is not empty.So insert newNode with red color.



- Here there are two consecutive Red nodes(25 & 40).
- The newNode's parent sibling is NULL.
- Here,we use LL Rotaton and Recolor then Recheck.

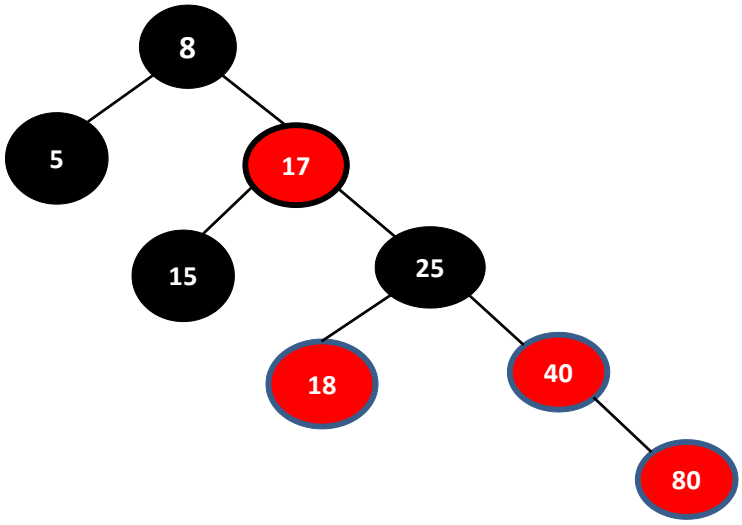




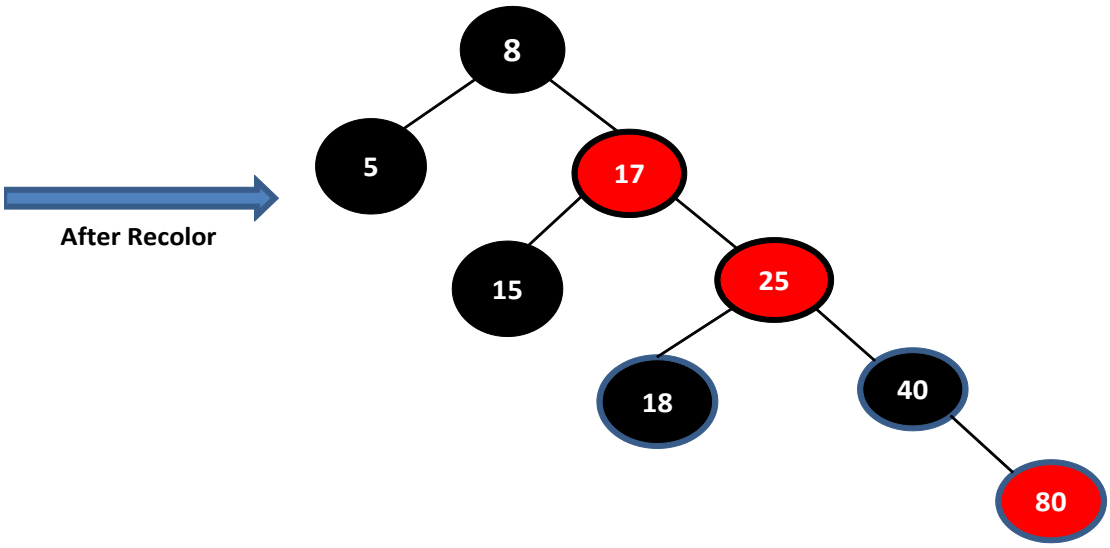
After LL Rotation & Recolor operation, the tree is satisfying all RED BLACK Tree properties.

Insert 80

Tree is not empty.So insert newNode with red color.

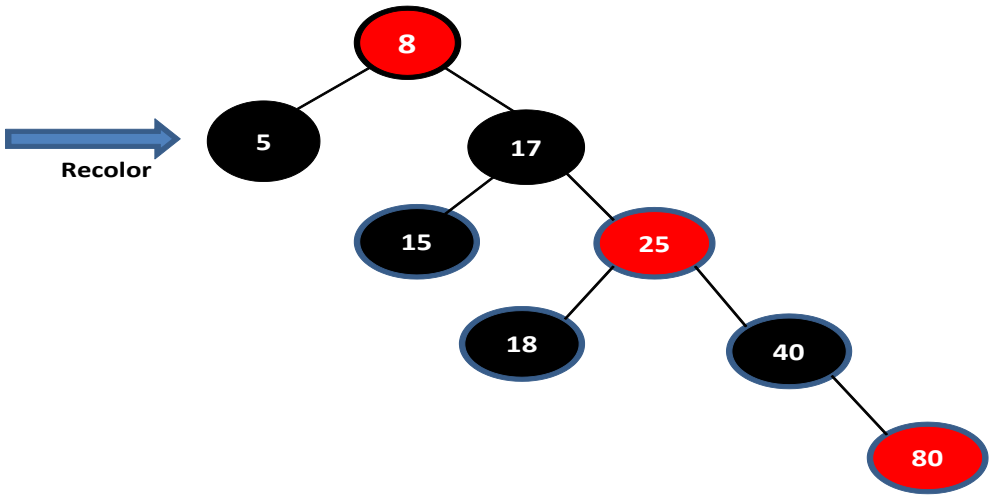


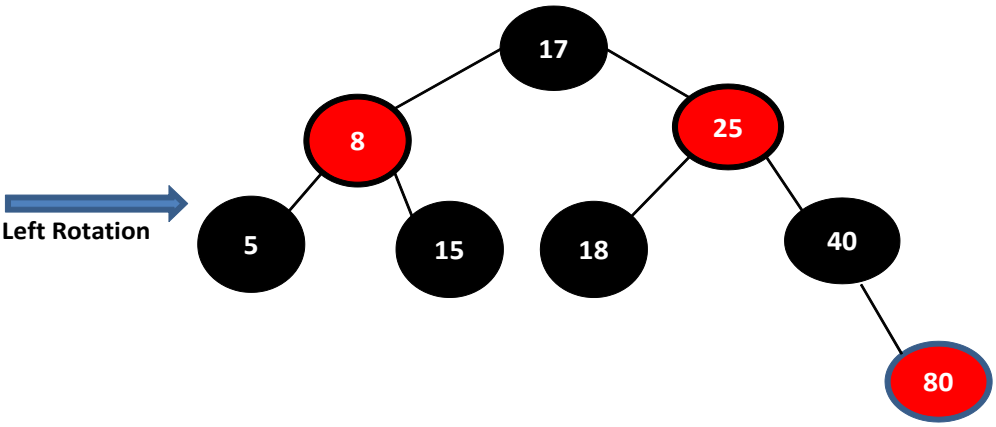
- Here there are two consecutive Red nodes(40 & 80).
- The newNode's parent sibling color is Red
- So we use RECOLOR and Recheck.



- After Recolor again there are two consecutive Red nodes(17 & 25).

- The newNode's parent sibling color is black.And nodes are same side.
- We use Recolor and Left Rotation then Recheck.





Finally above Tree is satisfying all the Properties of RED BLACK Tree and it is perfect RED BLACK Tree.

SPLAY TREE DATASTRUCTURE

- Splay tree is another variant of a binary search tree. In a splay tree, recently accessed element is placed at the root of the tree.
- In a splay tree, every operation is performed at the root of the tree. All the operations in splay tree are involved with a common operation called **"Splaying"**
- **Splaying** an element, is the process of bringing it to the root position by performing suitable rotation operations.

- Every operation on splay tree performs the splaying operation.
- For example, the insertion operation first inserts the new element using the binary search tree insertion process, then the newly inserted element is splayed so that it is placed at the root of the tree.
- The search operation in a splay tree is nothing but searching the element using binary search process and then splaying that searched element so that it is placed at the root of the tree.

Rotations in Splay Tree

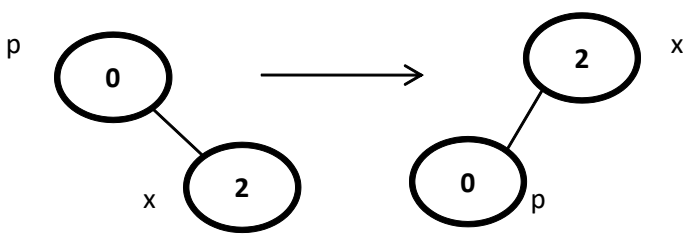
1. Zig Rotation
2. Zag Rotation
3. Zig - Zig Rotation
4. Zag - Zag Rotation
5. Zig - Zag Rotation
6. Zag - Zig Rotation

Example:

Insert 0,2,1,6,4,5 into Splay trees

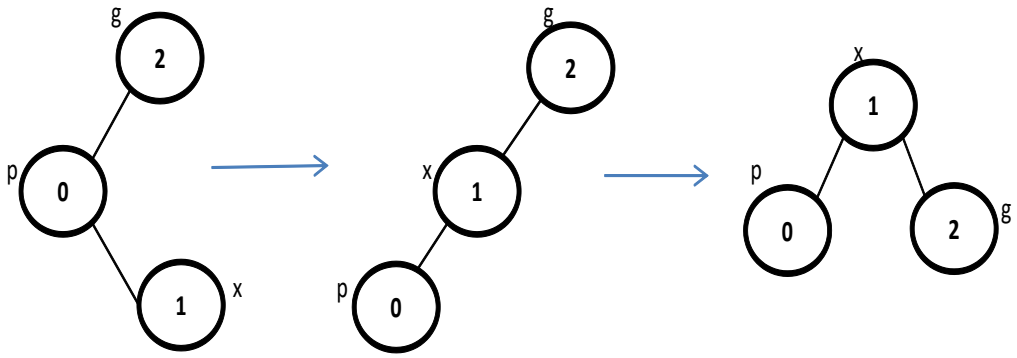
1. Zag Rotation

1 **Insert 0,2**



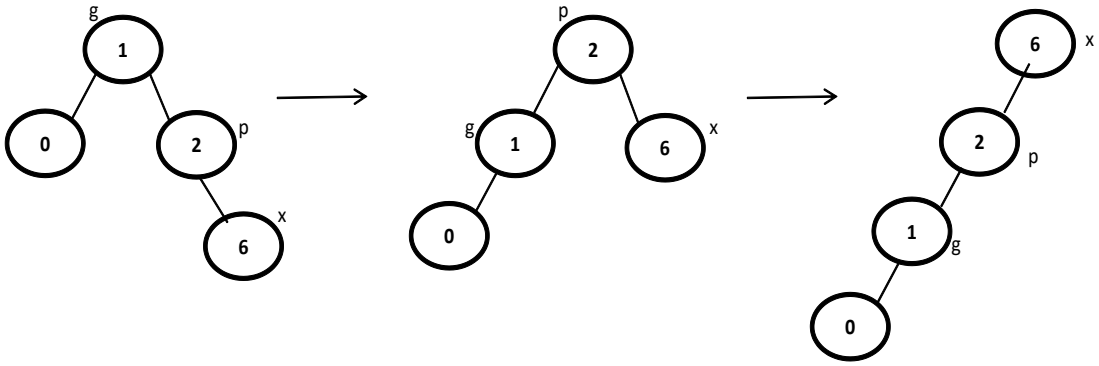
2. Zig-Zag Rotation

2 Insert 1



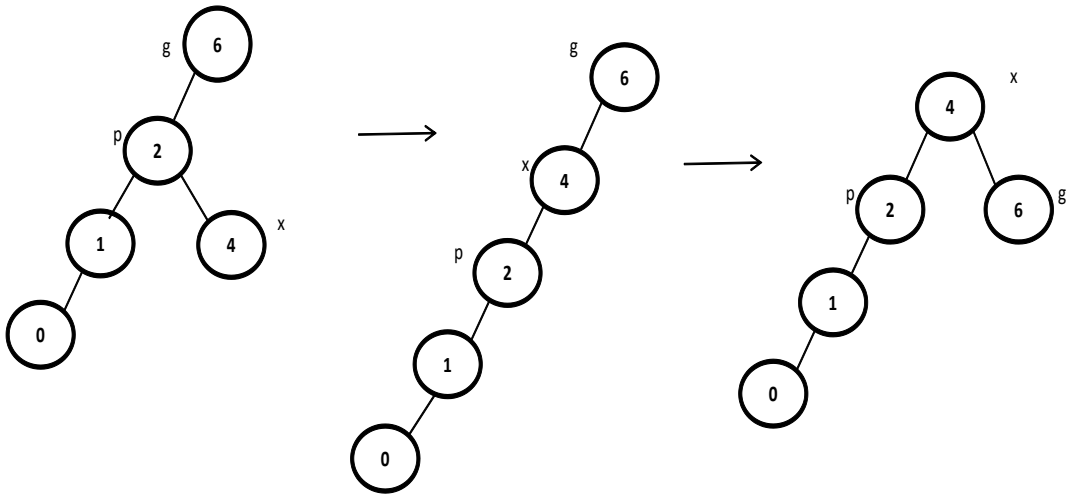
3.Zag-Zag Rotation

3 Insert 6



4.Zig-Zag Rotation

4 Insert 4



5.Zag-Zig Rotation

5 Insert 5