UNIT III

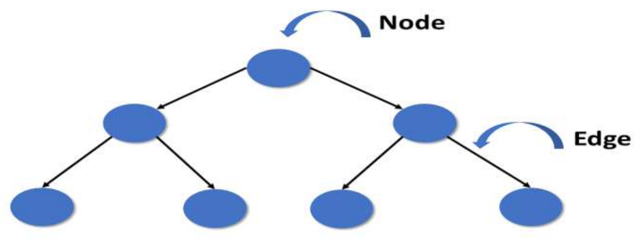
Search Trees: Binary Search Trees, Definition, Implementation, Operations- Searching, Insertion and Deletion, AVL Trees, Definition, Height of an AVL Tree, Operations – Insertion, Deletion and Searching, Red –Black, Splay Trees.

TREES INTRODUCTION

The tree is a nonlinear hierarchical data structure and comprises a collection of entities known as nodes. It connects each node in the tree data structure using "edges", both directed and undirected.

The image below represents the tree data structure. The blue-colored circles depict the nodes of the tree and the black lines connecting each node with another are called edges.

You will understand the parts of trees better, in the terminologies section.



The Necessity for a Tree in Data Structures

Other data structures like arrays, linked-list, stacks, and queues are linear data structures, and all these data structures store data in sequential order. Time complexity increases with increasing data size to perform operations like insertion and deletion on these linear data structures. But it is not acceptable for today's world of computation.

The non-linear structure of trees enhances the data storing, data accessing, and manipulation processes by employing advanced control methods traversal through it. You will learn about tree traversal in the upcoming section.

Tree Terminologies

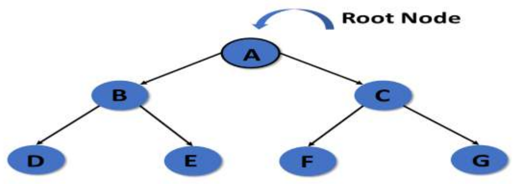
The following are some of the basic tree terms

- Root Node
- Edge
- Parent node
- Child node

- Siblings
- Leaf nodes or external nodes
- Internal nodes
- Degree
- Level
- Height
- Depth
- Path
- Subtree

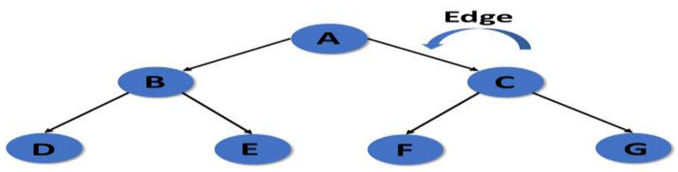
Root

- In a tree data structure, the root is the first node of the tree. The root node is the initial node of the tree in data structures.
- In the tree data structure, there must be only one root node.



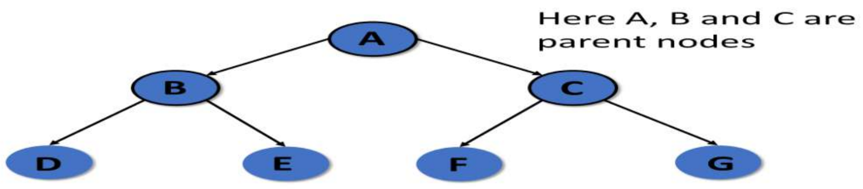
Edge

- In a tree in data structures, the connecting link of any two nodes is called the edge of the tree data structure.
- In the tree data structure, N number of nodes connecting with N -1 number of edges.



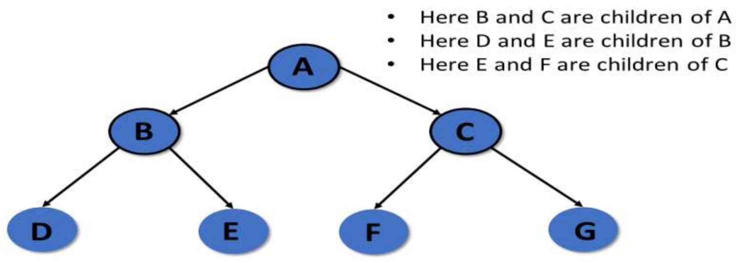
Parent

In the tree in data structures, the node that is the predecessor of any node is known as a parent node, or a node with a branch from itself to any other successive node is called the parent node.



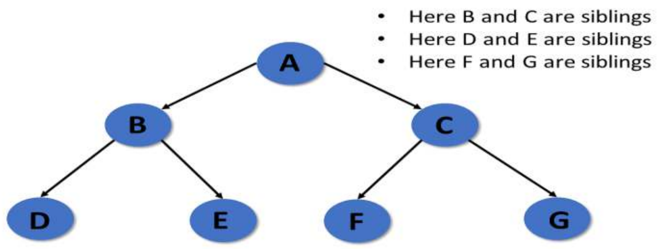
Child

- The node, a descendant of any node, is known as child nodes in data structures.
- In a tree, any number of parent nodes can have any number of child nodes.
- In a tree, every node except the root node is a child node.


Here B and C are children of A
Here D and E are children of B
Here E and F are children of C
A
B
C
D
E
F
G

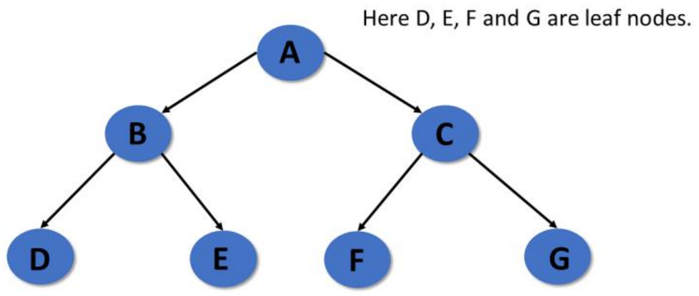
Siblings

In trees in the data structure, nodes that belong to the same parent are called siblings.


Here B and C are siblings
Here D and E are siblings
Here F and G are siblings
A
B
C
D
E
F
G

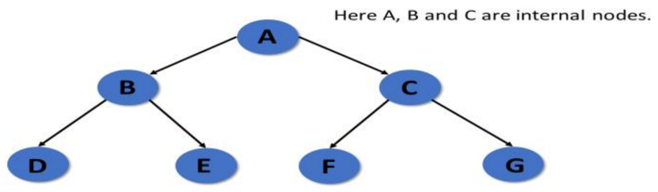
Leaf

- Trees in the data structure, the node with no child, is known as a leaf node.
- In trees, leaf nodes are also called external nodes or terminal nodes.



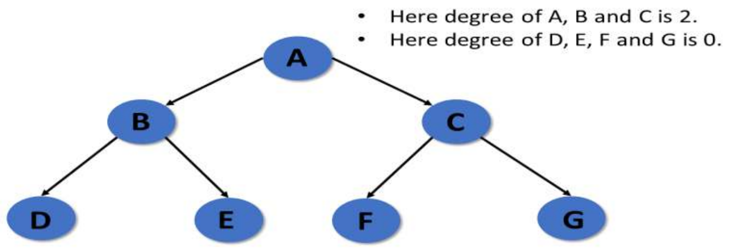
Internal nodes

- Trees in the data structure have at least one child node known as internal nodes.
- In trees, nodes other than leaf nodes are internal nodes.
- Sometimes root nodes are also called internal nodes if the tree has more than one node.



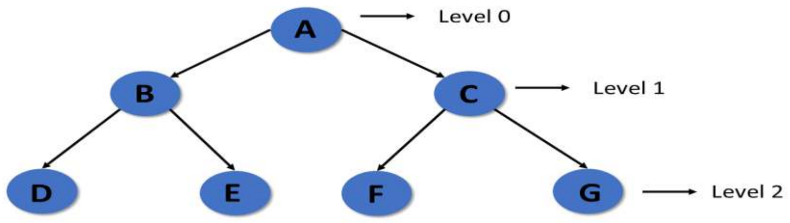
Degree

- In the tree data structure, the total number of children of a node is called the degree of the node.
- The highest degree of the node among all the nodes in a tree is called the Degree of Tree.


Here degree of A, B and C is 2.
Here degree of D, E, F and G is 0.
A
B
C
D
E
F
G

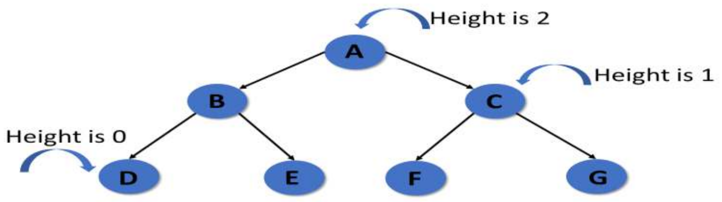
Level

In tree data structures, the root node is said to be at level 0, and the root node's children are at level 1, and the children of that node at level 1 will be level 2, and so on.



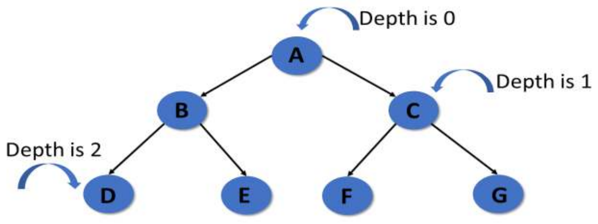
Height

- In a tree data structure, the number of edges from the leaf node to the particular node in the longest path is known as the height of that node.
- In the tree, the height of the root node is called "Height of Tree".
- The tree height of all leaf nodes is 0.



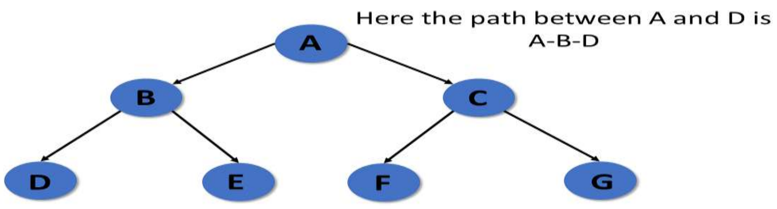
Depth

- In a tree, many edges from the root node to the particular node are called the depth of the tree.
- In the tree, the total number of edges from the root node to the leaf node in the longest path is known as "Depth of Tree".
- In the tree data structures, the depth of the root node is 0.



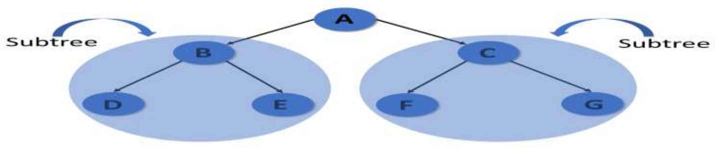
Path

- In the tree in data structures, the sequence of nodes and edges from one node to another node is called the path between those two nodes.
- The length of a path is the total number of nodes in a path.zx



Subtree

In the tree in data structures, each child from a node shapes a sub-tree recursively and every child in the tree will form a sub-tree on its parent node.

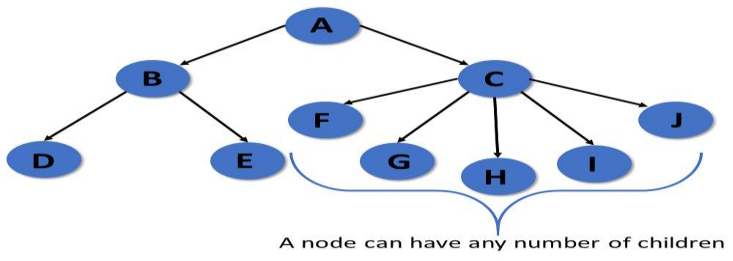


General Tree

The general tree is the type of tree where there are no constraints on the hierarchical structure.

Properties

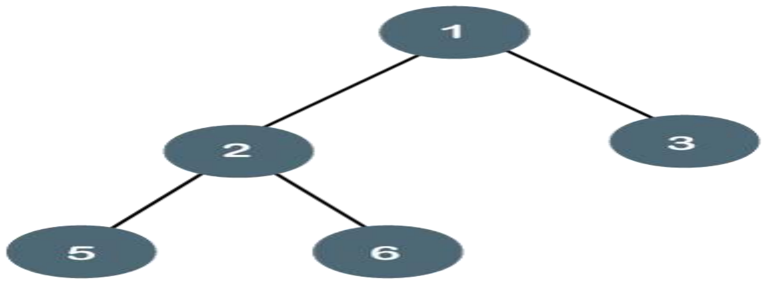
- The general tree follows all properties of the tree data structure.
- A node can have any number of nodes.



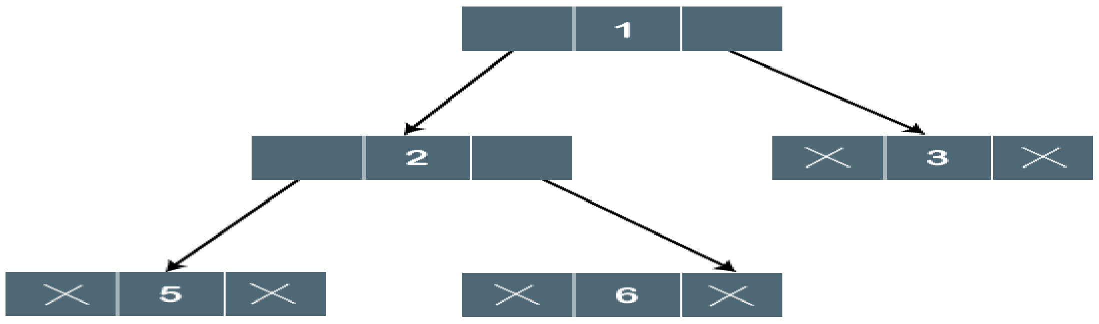
BINARY TREES

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.

Let's understand the binary tree through an example.



The above tree is a binary tree because each node contains the utmost two children. The logical representation of the above tree is given below:



In the above tree, node 1 contains two pointers, i.e., left and a right pointer pointing to the left and right node respectively. The node 2 contains both the nodes (left and right node); therefore, it has two pointers (left and right). The nodes 3, 5 and 6 are the leaf nodes, so all these nodes contain **NULL** pointer on both left and right parts.

Properties of Binary Tree

- At each level of i, the maximum number of nodes is 2^i.
- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (1+2+4+8) = 15. In general, the maximum number of nodes possible at height h is $(2^0 + 2^1 + 2^2 + \ldots 2^h) = 2^{h+1} - 1$.
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

If there are 'n' number of nodes in the binary tree.

The minimum height can be computed as:

As we know that,

$n = 2^{h+1} - 1$

$n+1 = 2^{h+1}$

Taking log on both the sides,

$\log_2(n+1) = \log 2(2^{h+1})$

$\log_2(n+1) = h+1$

$\mathbf{h = \log_2(n+1) - 1}$

The maximum height can be computed as:

As we know that,

$n = h+1$

$\mathbf{h = n-1}$

Types of Binary Tree

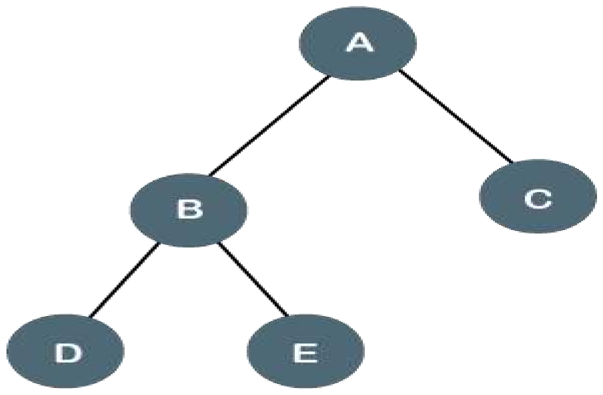
There are four types of Binary tree:

- Full/ proper/ strict Binary tree
- Complete Binary tree
- Perfect Binary tree
- Degenerate Binary tree
- Balanced Binary tree

1. Full/ proper/ strict Binary tree

The full binary tree is also known as a strict binary tree. The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children. The full binary tree can also be defined as the tree in which each node must contain 2 children except the leaf nodes.

Let's look at the simple example of the Full Binary tree.



In the above tree, we can observe that each node is either containing zero or two children; therefore, it is a Full Binary tree.

Properties of Full Binary Tree

- The number of leaf nodes is equal to the number of internal nodes plus 1. In the above example, the number of internal nodes is 5; therefore, the number of leaf nodes is equal to 6.
- The maximum number of nodes is the same as the number of nodes in the binary tree, i.e., 2^{h+1} -1.
- The minimum number of nodes in the full binary tree is 2*h-1.
- The minimum height of the full binary tree is $\mathbf{log_2(n+1) - 1}$.
- The maximum height of the full binary tree can be computed as:

n= 2*h - 1

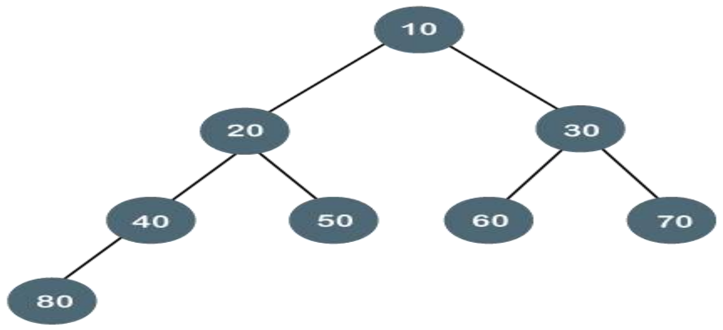
n+1 = 2*h

h = n+1/2

Complete Binary Tree

The complete binary tree is a tree in which all the nodes are completely filled except the last level. In the last level, all the nodes must be as left as possible. In a complete binary tree, the nodes should be added from the left.

Let's create a complete binary tree.



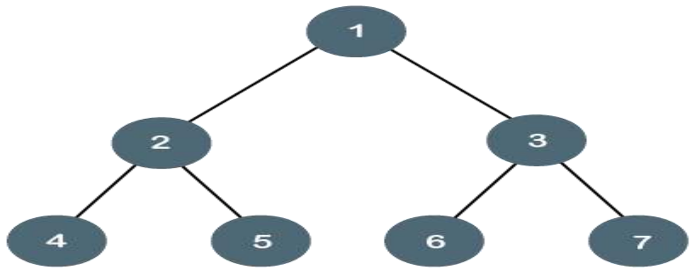
The above tree is a complete binary tree because all the nodes are completely filled, and all the nodes in the last level are added at the left first.

Properties of Complete Binary Tree

- The maximum number of nodes in complete binary tree is 2^{h+1} - 1.
- The minimum number of nodes in complete binary tree is 2^h.
- The minimum height of a complete binary tree is **$\log_2(n+1)$ - 1.**
- The maximum height of a complete binary tree is

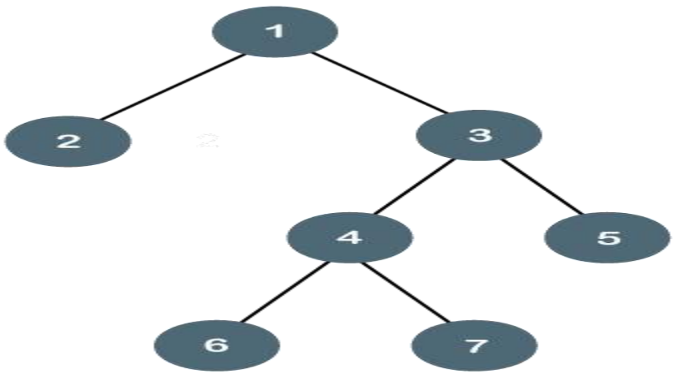
Perfect Binary Tree

A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.



Let's look at a simple example of a perfect binary tree.

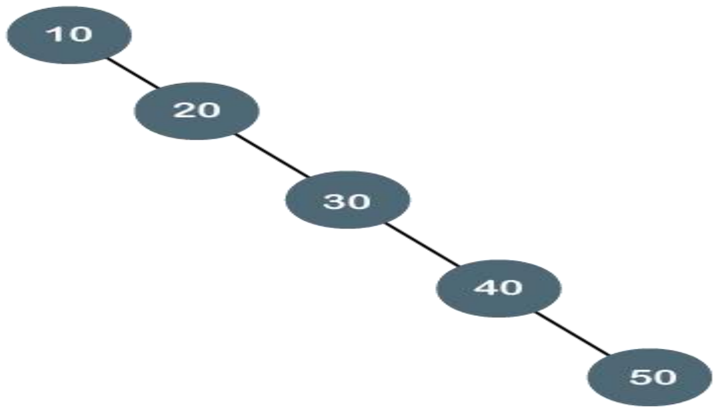
The below tree is not a perfect binary tree because all the leaf nodes are not at the same level.



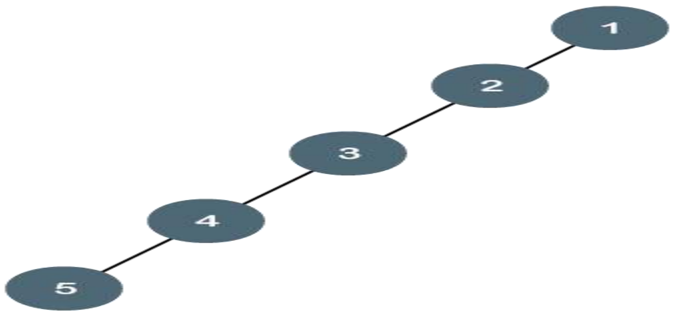
Degenerate Binary Tree

The degenerate binary tree is a tree in which all the internal nodes have only one children.

Let's understand the Degenerate binary tree through examples.



The above tree is a degenerate binary tree because all the nodes have only one child. It is also known as a right-skewed tree as all the nodes have a right child only.

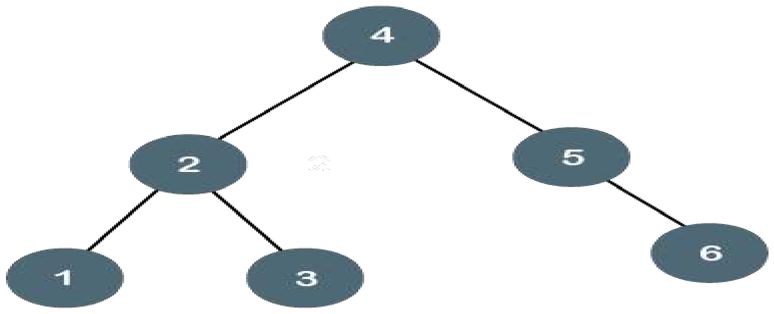


The above tree is also a degenerate binary tree because all the nodes have only one child. It is also known as a left-skewed tree as all the nodes have a left child only.

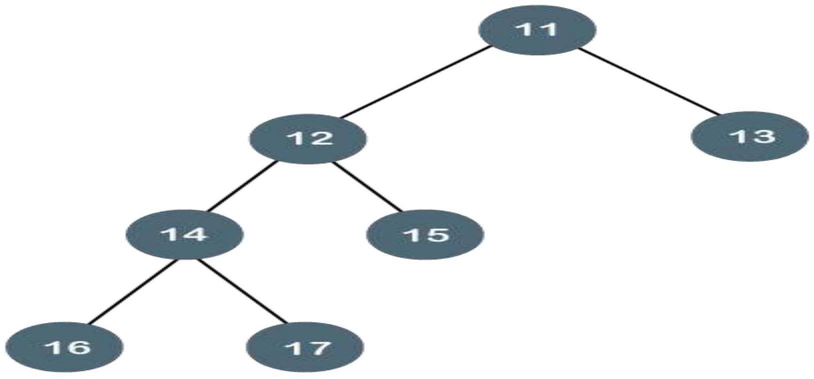
Balanced Binary Tree

The balanced binary tree is a tree in which both the left and right trees height differ by atmost 1. For example, ***AVL*** and ***Red-Black trees*** are balanced binary tree.

Let's understand the balanced binary tree through examples.



The above tree is a balanced binary tree because the difference between the height of left subtree and right subtree is zero.



The above tree is not a balanced binary tree because the difference between the height of left subtree and the right subtree is greater than 1.

Tree Traversals

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees**.**

Depth First Traversals:

(a) Inorder (Left, Root, Right)
(b) Preorder (Root, Left, Right)
(c) Postorder (Left, Right, Root)

Breadth-First or Level Order Traversal

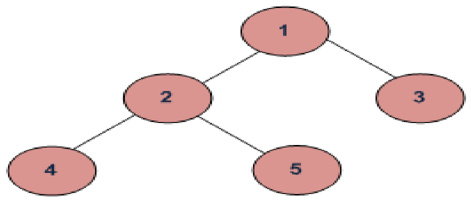
Let see each traversal method with example.

Inorder Traversal

Algorithm Inorder(tree)

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

Uses of Inorder
In the case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal s reversed can be used.
Example:



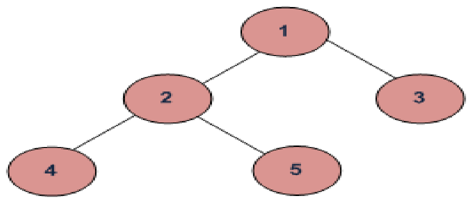
In order traversal for the above-given figure is 4 2 5 1 3.

Preorder Traversal

Algorithm Preorder(tree)

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

Uses of Preorder
Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on an expression tree**.**



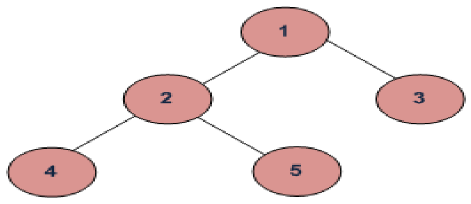
Example: Preorder traversal for the above-given figure is 1 2 4 5 3.

Postorder Traversal

Algorithm Postorder(tree)

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

EXAMPLE

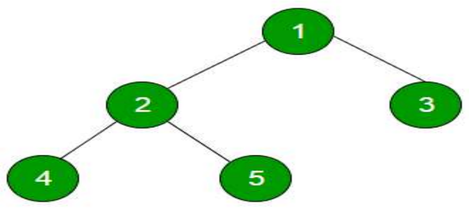


Example: Postorder traversal for the above-given figure is 4 5 2 3 1.

Uses of Postorder
Postorder traversal is also useful to get the postfix expression of an expression tree.

Level Order Binary Tree Traversal

Level order traversal of a tree is breadth first traversal for the tree.



Level order traversal of the above tree is 1 2 3 4 5

Construct a binary tree from inorder and postorder traversals

The idea is to start with the root node, which would be the last item in the postorder sequence, and find the boundary of its left and right subtree in the inorder sequence. To find the boundary, search for the index of the root node in the inorder sequence. All keys before the root node in the inorder sequence become part of the left subtree, and all keys after the root node become part of the right subtree. Repeat this recursively for all nodes in the tree and construct the tree in the process.

To illustrate, consider the following inorder and postorder sequence:

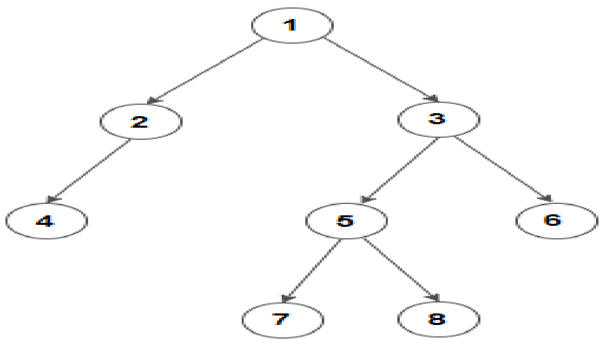
Inorder : { 4, 2, 1, 7, 5, 8, 3, 6 }
Postorder : { 4, 2, 7, 8, 5, 6, 3, 1 }

Root would be the last element in the postorder sequence, i.e., 1. Next, locate the index of the root node in the inorder sequence. Now since 1 is the root node, all nodes before 1 in the inorder sequence must be included in the left subtree of the root node, i.e., {4, 2} and all the nodes after 1 must be included in the right subtree, i.e., {7, 5, 8, 3, 6}. Now the problem is reduced to building the left and right subtrees and linking them to the root node.

Left subtree:
Inorder : {4, 2}
Postorder : {4, 2}
Right subtree:
Inorder : {7, 5, 8, 3, 6}
Postorder : {7, 8, 5, 6, 3}

The idea is to recursively follow the above approach until the complete tree is constructed.

The final tree will be

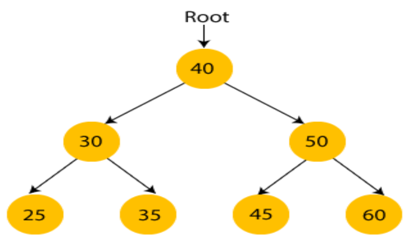


Binary Search Tree(BST)

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.

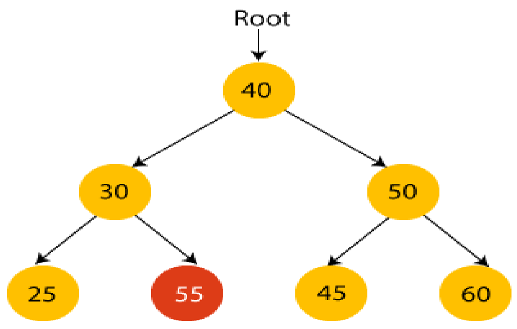
let's understand the concept of Binary search tree with an example.



In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.

Similarly, we can see the left child of root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search tree. Therefore, we can say that the tree in the above image is a binary search tree.

Suppose if we change the value of node 35 to 55 in the above tree, check whether the tree will be binary search tree or not.



In the above tree, the value of root node is 40, which is greater than its left child 30 but smaller than right child of 30, i.e., 55. So, the above tree does not satisfy the property of Binary search tree. Therefore, the above tree is not a binary search tree.

Advantages of Binary search tree

- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

Example of creating a Binary Search Tree (BST)

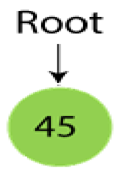
Now, let's see the creation of binary search tree using an example.

Suppose the data elements are **- 45, 15, 79, 90, 10, 55, 12, 20, 50**

- First, we have to insert **45** into the tree as the root of the tree.
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

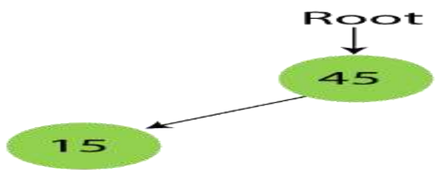
Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below -

Step 1 - Insert 45.



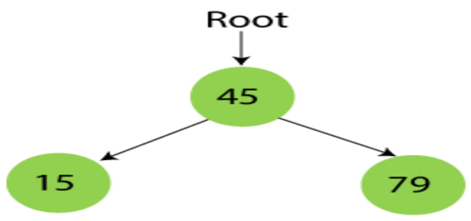
Step 2 - Insert 15.

As 15 is smaller than 45, so insert it as the root node of the left subtree.



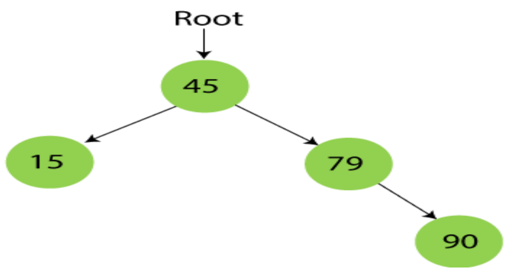
Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right subtree.



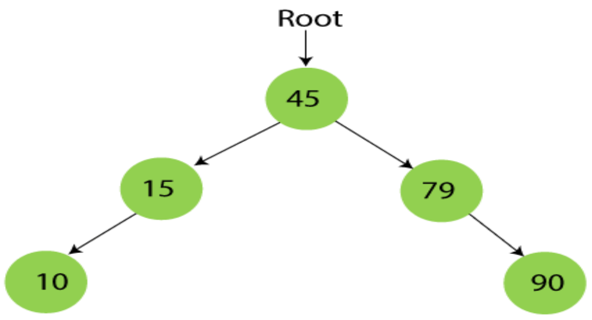
Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



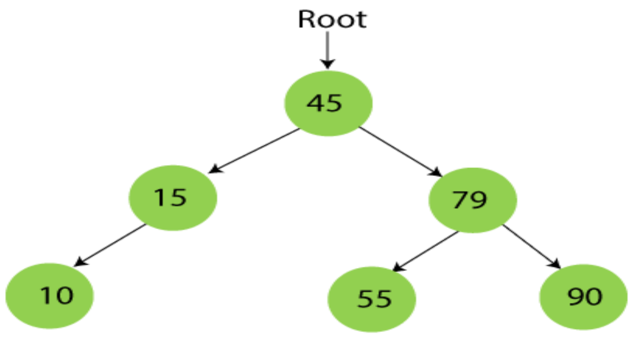
Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



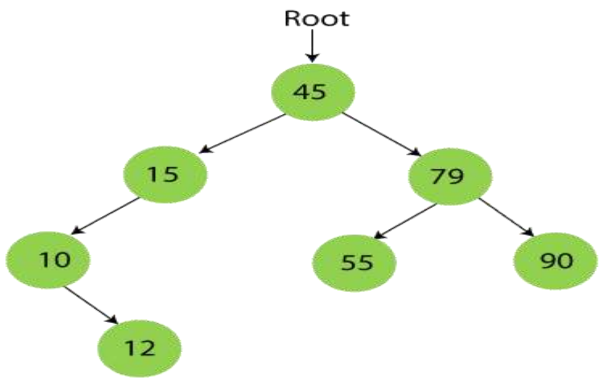
Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



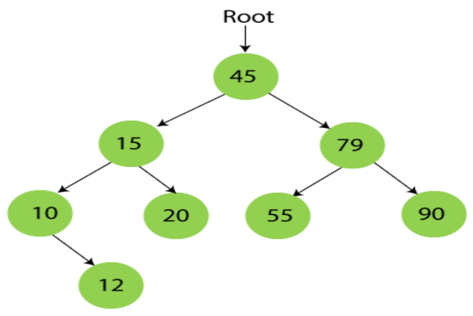
Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



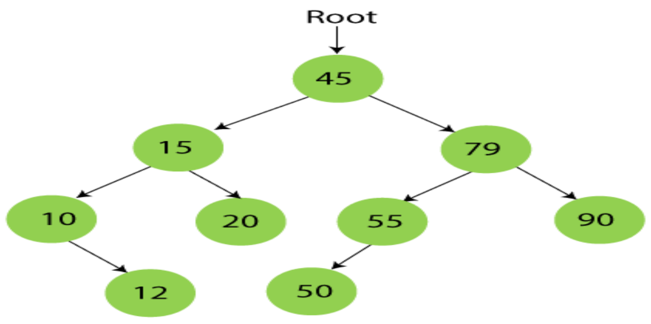
Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.



Now, the creation of binary search tree is completed. After that, let's move towards the operations that can be performed 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 tree.

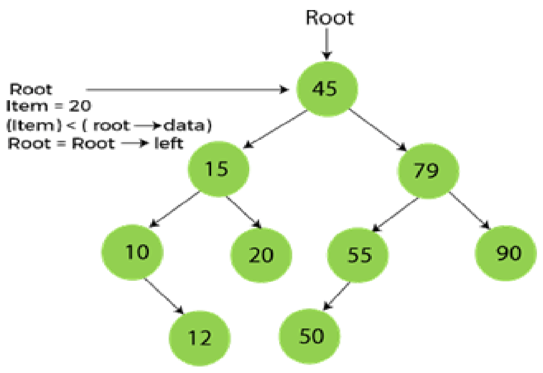
Searching in Binary search tree(BST)

Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows -

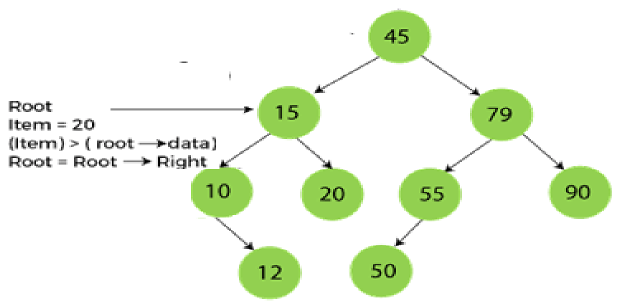
1. First, compare the element to be searched with the root element of the tree.
2. If root is matched with the target element, then return the node's location.
3. If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
4. If it is larger than the root element, then move to the right subtree.
5. Repeat the above procedure recursively until the match is found.
6. If the element is not found or not present in the tree, then return NULL.

Now, let's understand the searching in binary tree using an example. We are taking the binary search tree formed above. Suppose we have to find node 20 from the below tree.

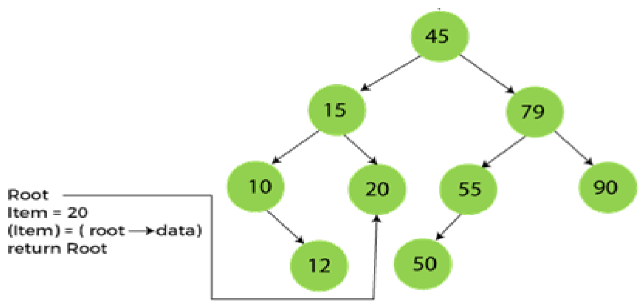
Step1:



Step2:



Step3:



Now, let's see the algorithm to search an element in the Binary search tree.

Now let's understand how the deletion is performed on a binary search tree. We will also see an example to delete an element from the given tree.

Deletion in Binary Search tree(BST)

In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur -

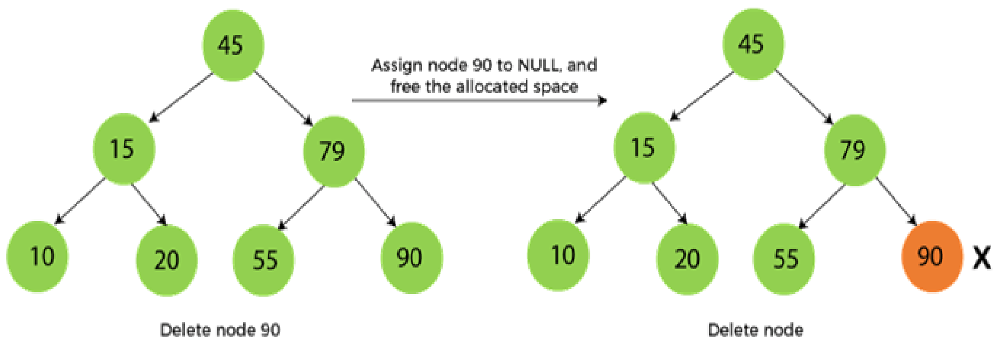
- The node to be deleted is the leaf node, or,
- The node to be deleted has only one child, and,
- The node to be deleted has two children

We will understand the situations listed above in detail.

When the node to be deleted is the leaf node

It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.

We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.



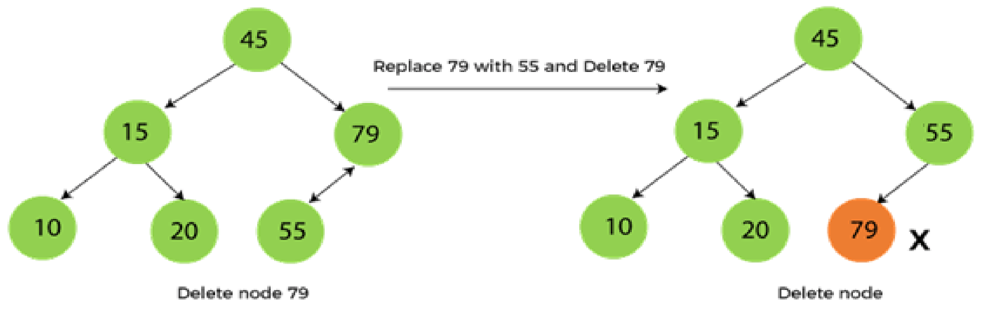
When the node to be deleted has only one child

In this case, we have to replace the target node(Deleting node) with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted. So, we simply have to replace the child node with NULL and free up the allocated space.

We can see the process of deleting a node with one child from BST in the below image.

In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.

So, the replaced node 79 will now be a leaf node that can be easily deleted.



When the node to be deleted has two children

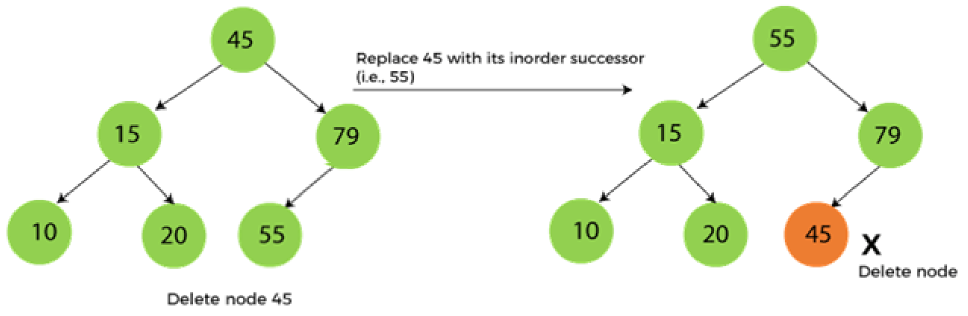
This case of deleting a node in BST is a bit complex among other two cases. In such a case, the steps to be followed are listed as follows -

- First, find the inorder successor of the node to be deleted.

- After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
- And at last, replace the node with NULL and free up the allocated space.

The inorder successor is required when the right child of the node is not empty. We can obtain the inorder successor by finding the minimum element in the right child of the node.

We can see the process of deleting a node with two children from BST in the below image. In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.

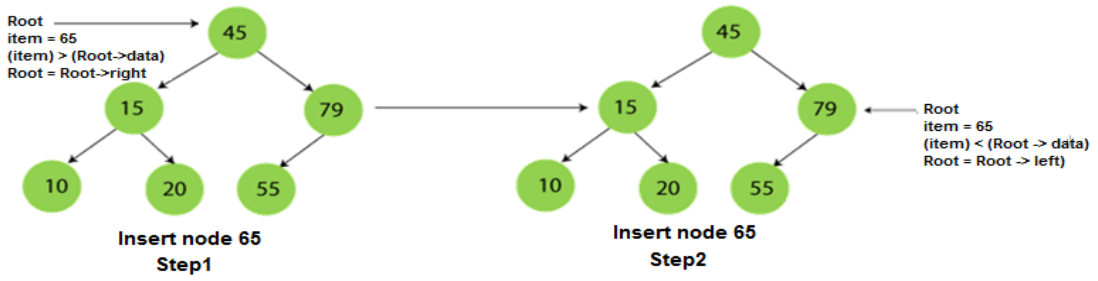


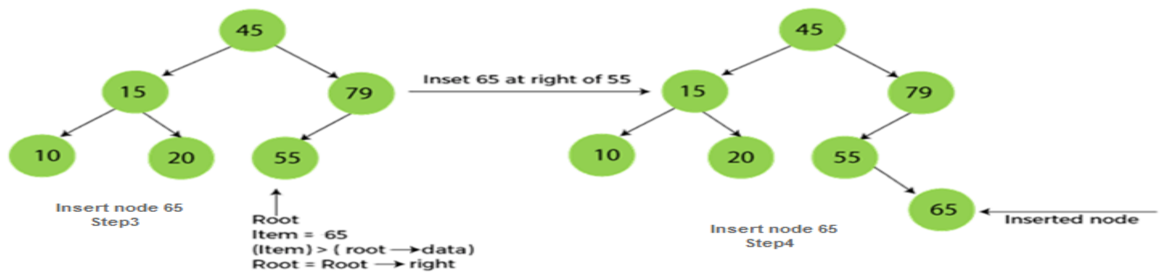
Now let's understand how insertion is performed on a binary search tree.

Insertion in Binary Search tree(BST)

A new key in BST is always inserted at the leaf. To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the root node, then search for an empty location in the left subtree. Else, search for the empty location in the right subtree and insert the data. Insert in BST is similar to searching, as we always have to maintain the rule that the left subtree is smaller than the root, and right subtree is larger than the root.

Now, let's see the process of inserting a node into BST using an example.





The complexity of the Binary Search tree

Let's see the time and space complexity of the Binary search tree. We will see the time complexity for insertion, deletion, and searching operations in best case, average case, and worst case.

1. Time Complexity

Operations	**Best case time complexity**	**Average case time complexity**	**Worst case time complexity**
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)
Search	O(log n)	O(log n)	O(n)

Worst case scenario indicates the BST is the Degenerated BST for all the operations (insertion, deletion and search)

Where 'n' is the number of nodes in the given tree.

2. Space Complexity

Operations	**Space complexity**
Insertion	O(n)
Deletion	O(n)
Search	O(n)

- The space complexity of all operations of Binary search tree is O(n).

Implementation of binary Search Tree traversal method

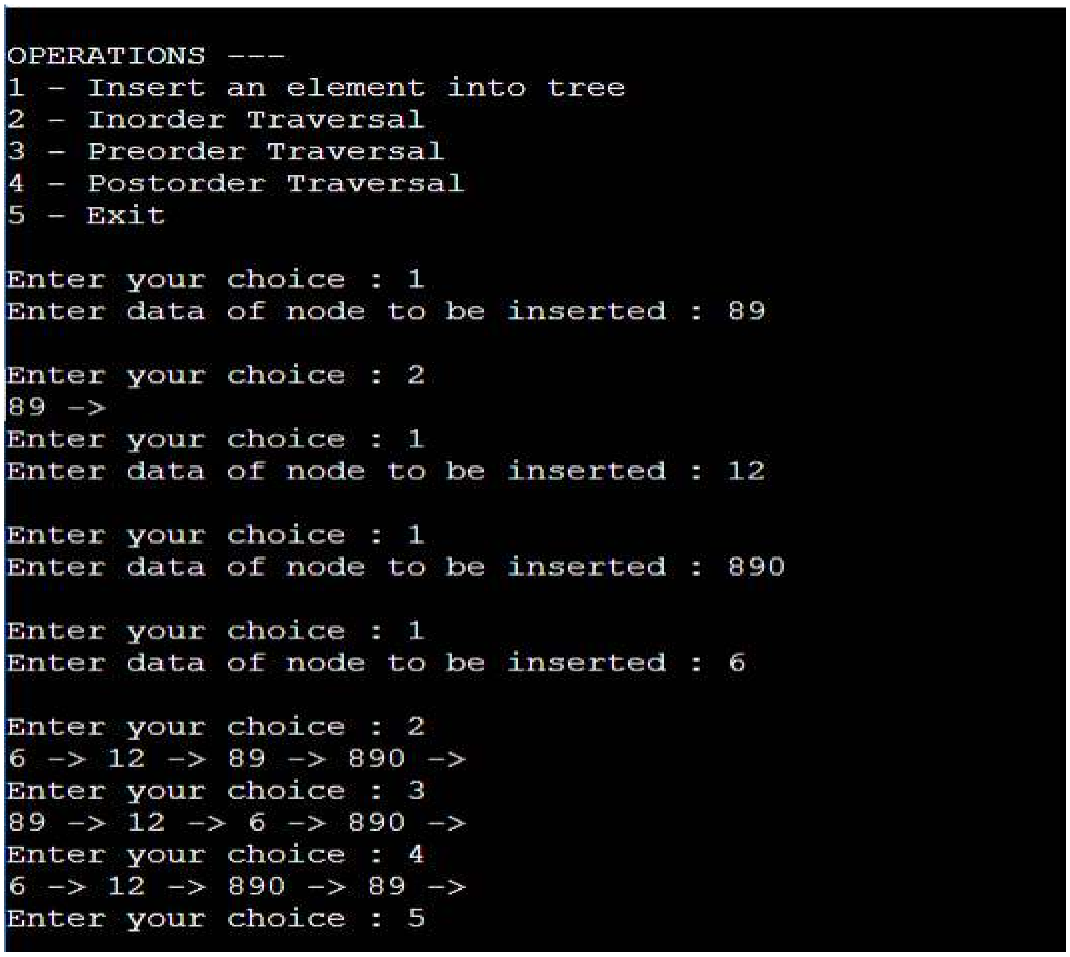
Program:

```
#include <stdio.h>
#include <stdlib.h>
 struct btnode
{
   int value;
   struct btnode *l;
   struct btnode *r;
}*root = NULL, *temp = NULL, *t2, *t1;
void insert();
void create();
void search( struct btnode *root);
void inorder(struct btnode *t);
void preorder(struct btnode *t);
void postorder(struct btnode *t);
void main()
{
   int ch;
   printf("\nOPERATIONS ---");
   printf("\n1 - Insert an element into tree\n");
   printf("2 - Inorder Traversal\n");
   printf("3 - Preorder Traversal\n");
   printf("4 - Postorder Traversal\n");
   printf("5 - Exit\n");
   while(1)
   {
      printf("\nEnter your choice : ");
      scanf("%d", &ch);
      switch (ch)
      {
      case 1:
         insert();
         break;
      case 2:
         inorder(root);
         break;
      case 3:
            preorder(root);
            break;
        case 4:
            postorder(root);
            break;
        case 5:
            exit(0);
        default :
            printf("Wrong choice, Please enter correct choice  ");
            break;
        }
    }
}
/* To insert a node in the tree */
void insert()
{
    create();
    if (root == NULL)
        root = temp;
    else
        search(root);
}
/* To create a node */
void create()
{
    int data;
    printf("Enter data of node to be inserted : ");
    scanf("%d", &data);
    temp = (struct btnode *)malloc(1*sizeof(struct btnode));
    temp->value = data;
    temp->l = temp->r = NULL;
}
/* Function to search the appropriate position to insert the new node */
void search(struct btnode *t)
{
    if ((temp->value > t->value) && (t->r != NULL))      /* value more than root node value insert
at right */
        search(t->r);
    else if ((temp->value > t->value) && (t->r == NULL))
        t->r = temp;
```

```
   else if ((temp->value < t->value) && (t->l != NULL))     /* value less than root node value
insert at left */
      search(t->l);
   else if ((temp->value < t->value) && (t->l == NULL))
      t->l = temp;
}
/* recursive function to perform inorder traversal of tree */
void inorder(struct btnode *t)
{
   if (root == NULL)
   {
      printf("No elements in a tree to display");
      return;
   }
   if (t->l != NULL)
      inorder(t->l);
   printf("%d -> ", t->value);
   if (t->r != NULL)
      inorder(t->r);
}
/* To find the preorder traversal */
void preorder(struct btnode *t)
{
   if (root == NULL)
   {
      printf("No elements in a tree to display");
      return;
   }
   printf("%d -> ", t->value);
   if (t->l != NULL)
      preorder(t->l);
   if (t->r != NULL)
      preorder(t->r);
}
 /* To find the postorder traversal */
void postorder(struct btnode *t)
{
   if (root == NULL)
   {
      printf("No elements in a tree to display ");
```

```
        return;
    }
    if (t->l != NULL)
        postorder(t->l);
    if (t->r != NULL)
        postorder(t->r);
    printf("%d -> ", t->value);
}
```

OUTPUT:



```
OPERATIONS ---
1 - Insert an element into tree
2 - Inorder Traversal
3 - Preorder Traversal
4 - Postorder Traversal
5 - Exit

Enter your choice : 1
Enter data of node to be inserted : 89

Enter your choice : 2
89 ->
Enter your choice : 1
Enter data of node to be inserted : 12

Enter your choice : 1
Enter data of node to be inserted : 890

Enter your choice : 1
Enter data of node to be inserted : 6

Enter your choice : 2
6 -> 12 -> 89 -> 890 ->
Enter your choice : 3
89 -> 12 -> 6 -> 890 ->
Enter your choice : 4
6 -> 12 -> 890 -> 89 ->
Enter your choice : 5
```

AVL Tree

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

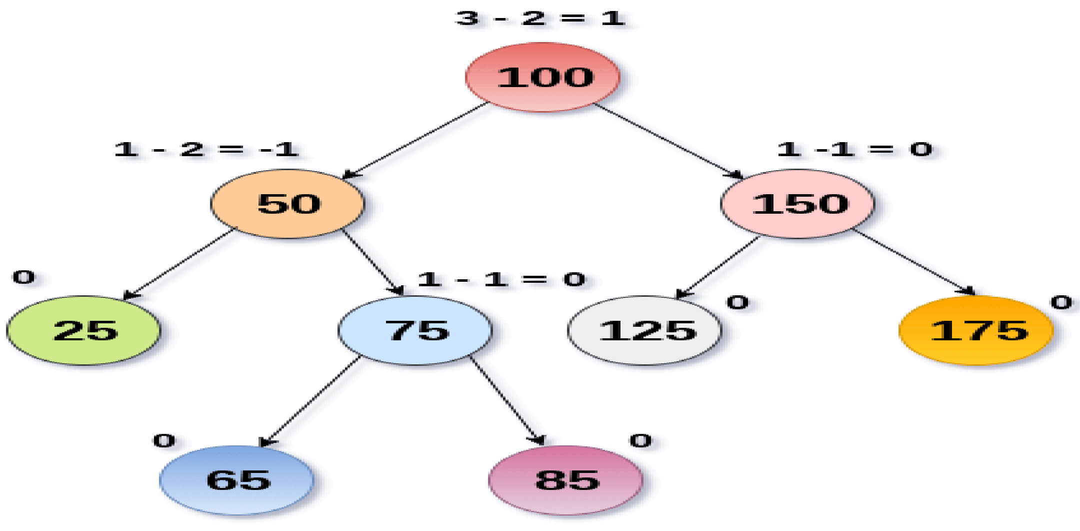
AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.

If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



AVL Tree

Why AVL Tree?

AVL tree controls the height of the binary search tree by not letting it to be skewed. The time taken for all operations in a binary search tree of height h is **O(h)**. However, it can be extended to **O(n)** if the BST becomes skewed (i.e. worst case). By limiting this height to log n, AVL tree imposes an upper bound on each operation to be **O(log n)** where n is the number of nodes.

AVL Rotations

We perform rotation in AVL tree only in case if Balance Factor is other than **-1, 0, and 1**. There are basically four types of rotations which are as follows:

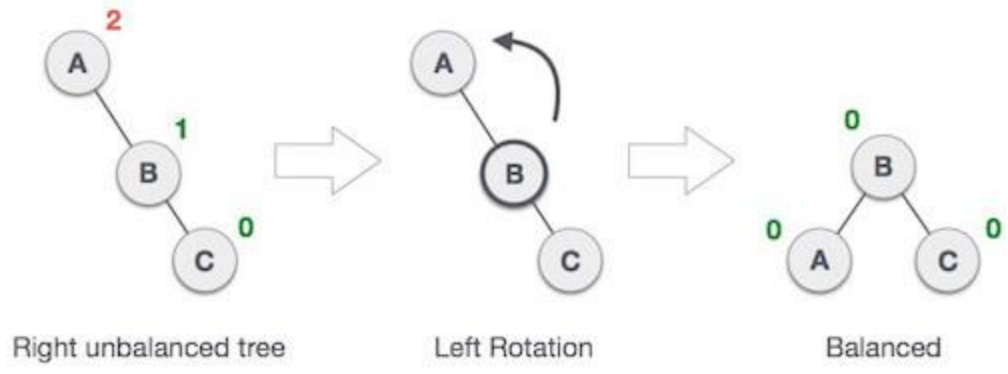
[Type text]

1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation : Inserted node is in the right subtree of right subtree of A
3. L R rotation : Inserted node is in the right subtree of left subtree of A
4. R L rotation : Inserted node is in the left subtree of right subtree of A

Where node A is the node whose balance Factor is other than -1, 0, 1.
The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

1. RR Rotation

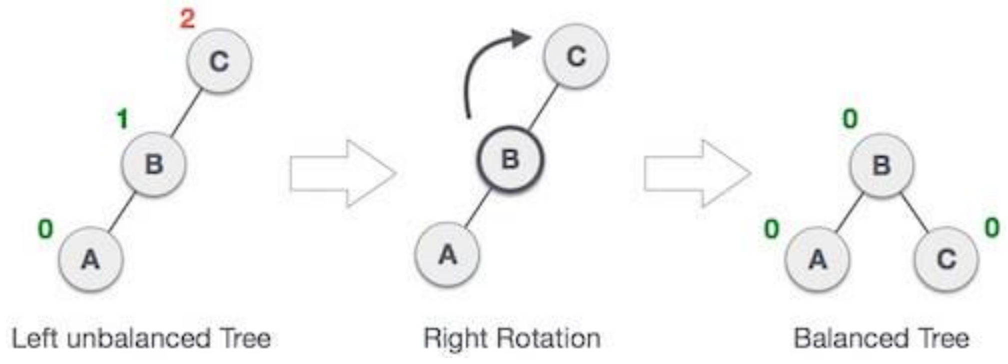
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation
is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

2. LL Rotation

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation,
LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.



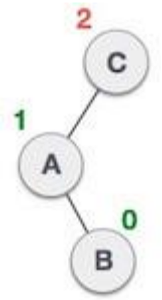
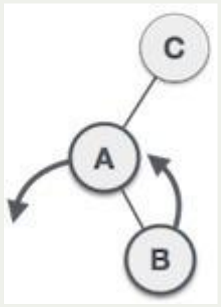
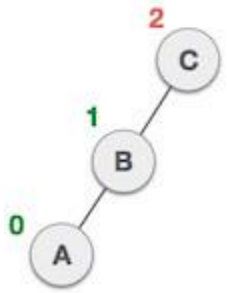
[Type text]

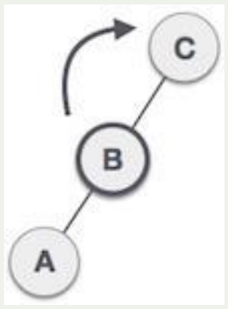
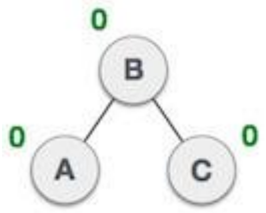
In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

3. LR Rotation

Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

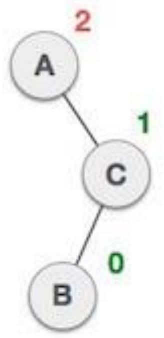
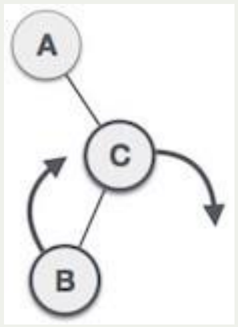
Let us understand each and every step very clearly:

State	**Action**
	A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C
	As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node **A**, has become the left subtree of **B**.
	After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of **C**

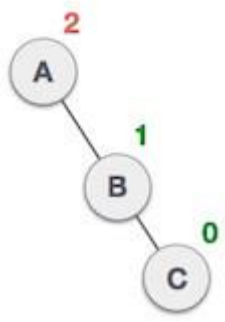
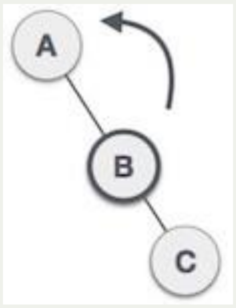
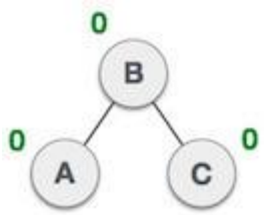
	Now we perform LL clockwise rotation on full tree, i.e. on node C. node **C** has now become the right subtree of node B, A is left subtree of B
	Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

4. RL Rotation

. R L rotation= LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

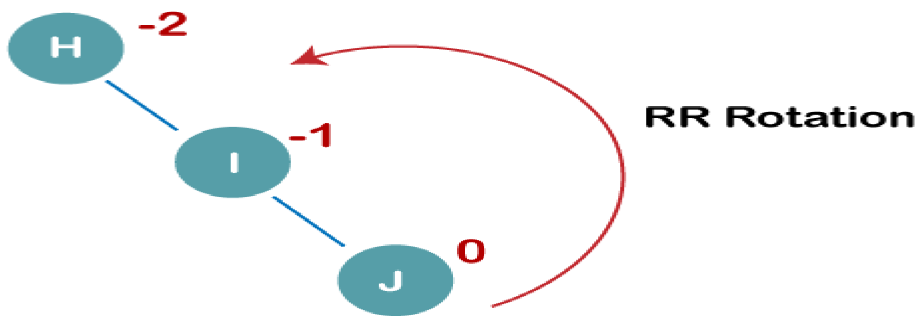
State	**Action**
	A node **B** has been inserted into the left subtree of **C** the right subtree of **A**, because of which A has become an unbalanced node having balance factor -2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A
	As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at **C** is performed first. By doing RR rotation, node **C** has become the right subtree of **B**.

[Type text]

	After performing LL rotation, node **A** is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.
	Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node **C** has now become the right subtree of node B, and node A has become the left subtree of B.
	Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

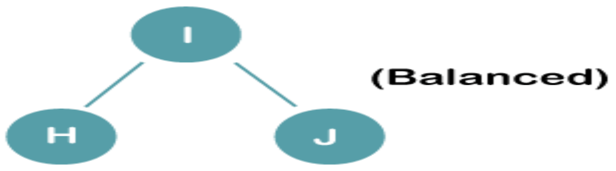
AVL TREE CONSTRUCTION

Construct an AVL tree having the following elements
H, I, J, B, A, E, C, F, D, G, K, L
1. Insert H, I, J

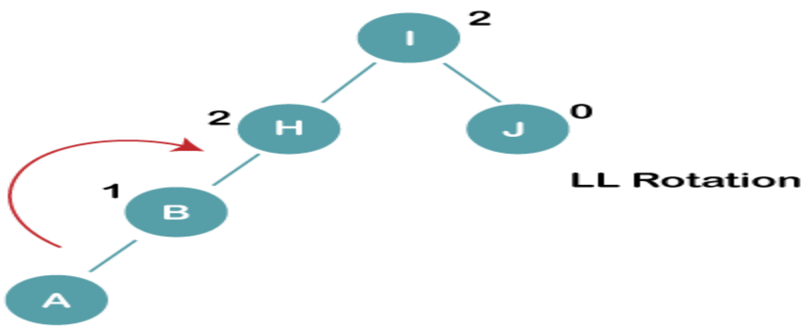


On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2. Since the BST is right-skewed, we will perform RR Rotation on node H.

The resultant balance tree is:

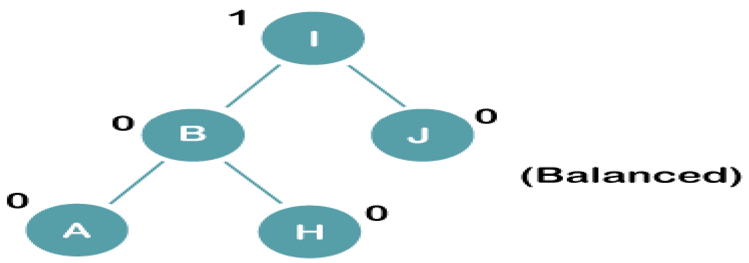


2. Insert B, A

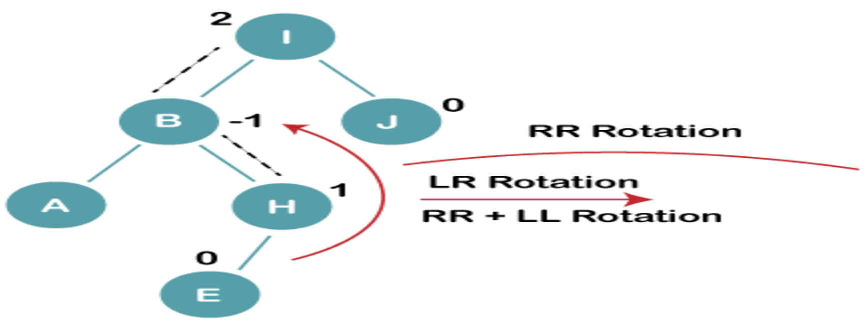


On inserting the above elements, especially in case of A, the BST becomes unbalanced as the Balance Factor of H and I is 2, we consider the first node from the last inserted node i.e. H. Since the BST from H is left-skewed, we will perform LL Rotation on node H.

The resultant balance tree is:



3. Insert E

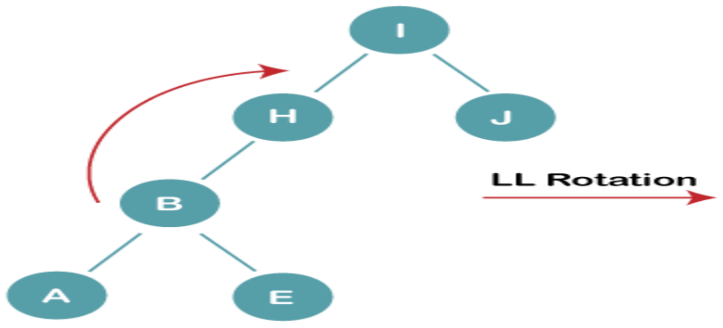


[Type text]

On inserting E, BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I we find that it is inserted in the left subtree of right subtree of I, we will perform LR Rotation on node I. LR = RR + LL rotation

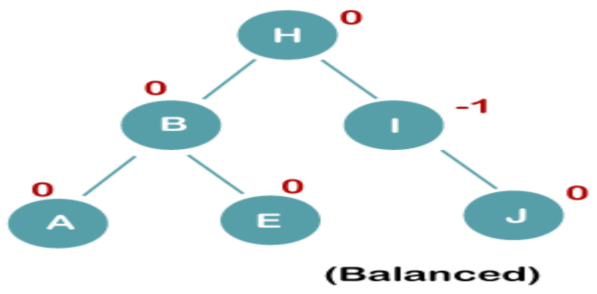
3 a) We first perform RR rotation on node B

The resultant tree after RR rotation is:

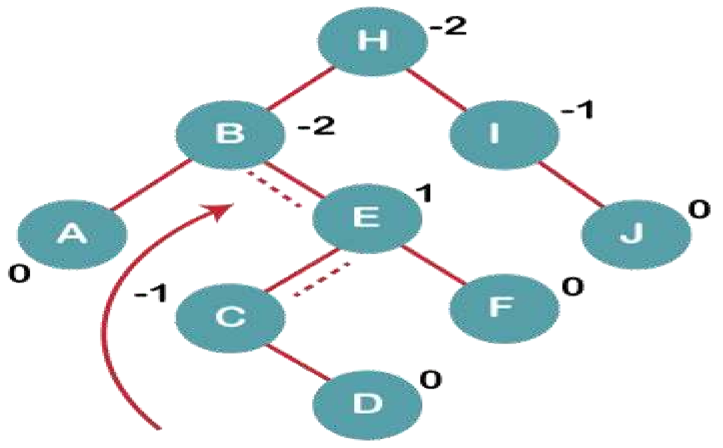


3b) We first perform LL rotation on the node I

The resultant balanced tree after LL rotation is**:**



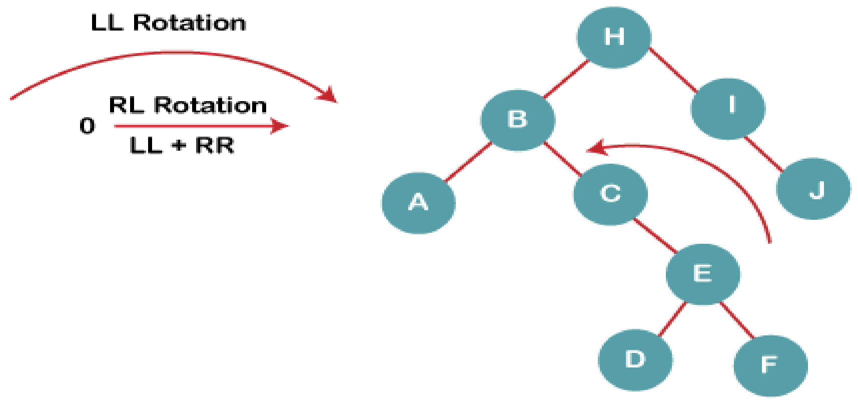
4. Insert C, F, D



On inserting C, F, D, BST becomes unbalanced as the Balance Factor of B and H is -2, since if we travel from D to B we find that it is inserted in the right subtree of left subtree of B, we will perform RL Rotation on node I. RL = LL + RR rotation.

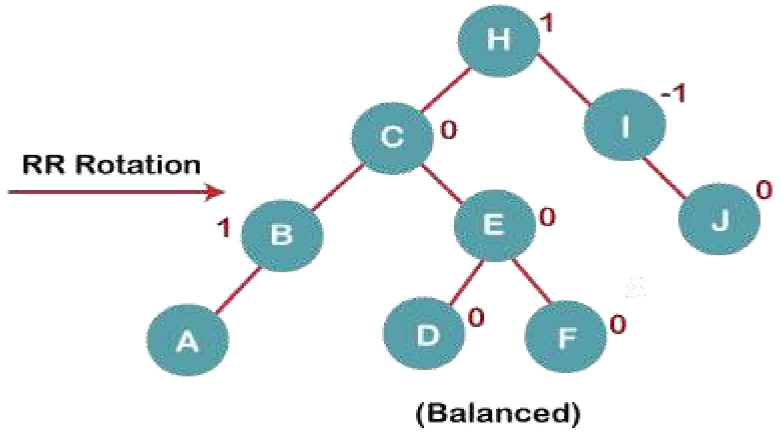
4a) We first perform LL rotation on node E

The resultant tree after LL rotation is:

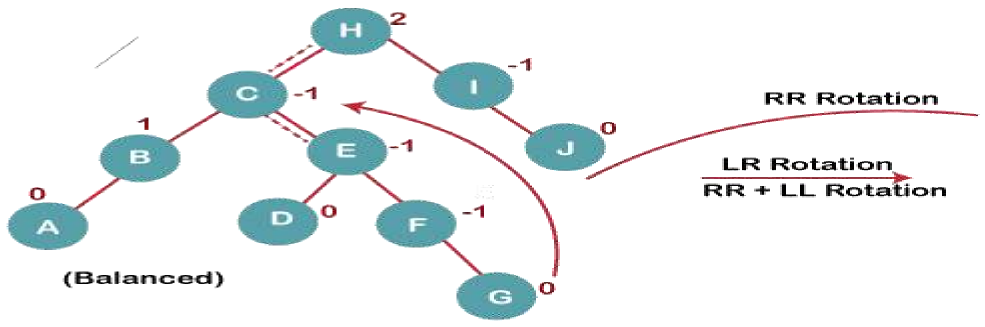


4b) We then perform RR rotation on node B

The resultant balanced tree after RR rotation is:



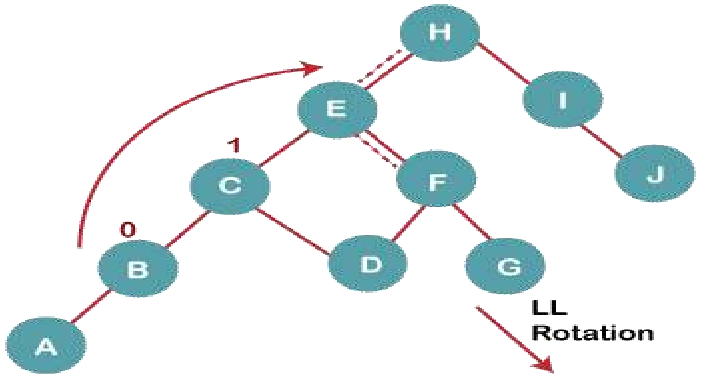
5. Insert G



On inserting G, BST become unbalanced as the Balance Factor of H is 2, since if we travel from G to H, we find that it is inserted in the left subtree of right subtree of H, we will perform LR Rotation on node I. LR = RR + LL rotation.

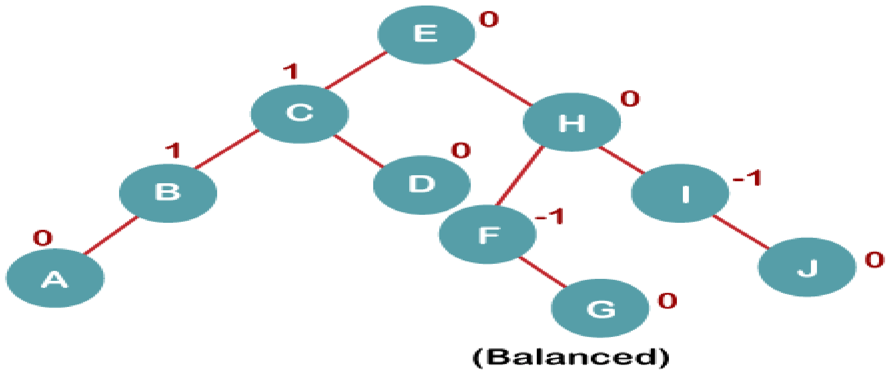
5 a) We first perform RR rotation on node C

The resultant tree after RR rotation is:

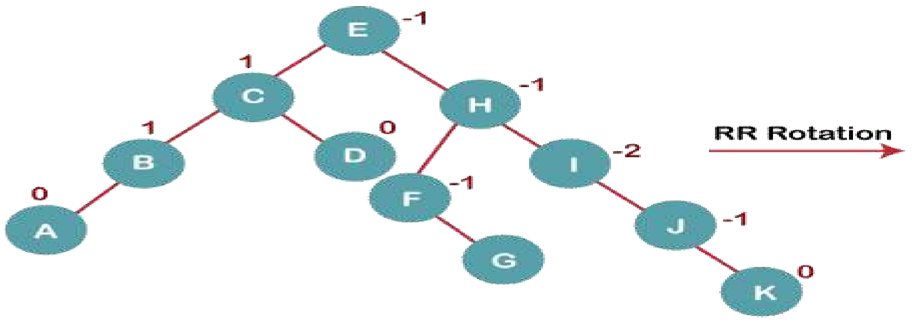


5 b) We then perform LL rotation on node H

The resultant balanced tree after LL rotation is:



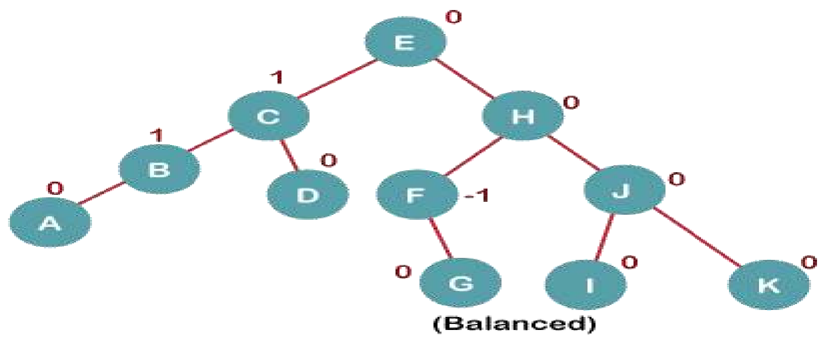
6. Insert K



On inserting K, BST becomes unbalanced as the Balance Factor of I is -2. Since the BST is right-skewed from I to K, hence we will perform RR Rotation on the node I.

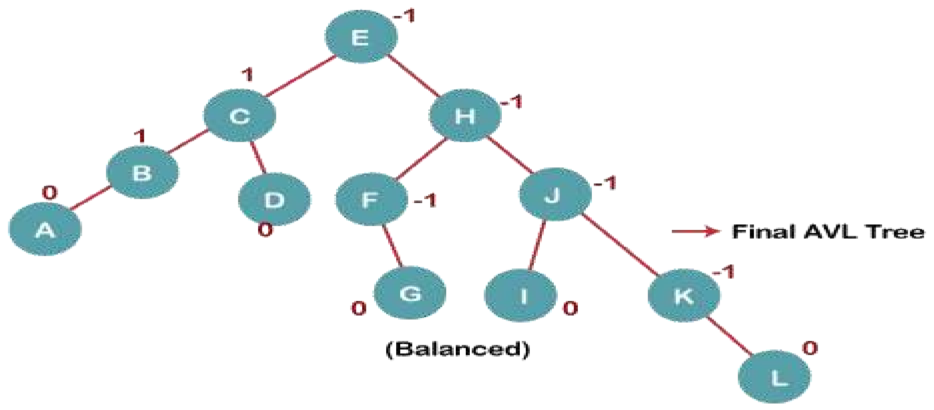
The resultant balanced tree after RR rotation is:

[Type text]



7. Insert L

On inserting the L tree is still balanced as the Balance Factor of each node is now either, -1, 0, +1. Hence the tree is a Balanced AVL tree



Operations on AVL tree

The following operations are performed on AVL Tree

1. Insertion
2. Deletion
3. Search

Insertion Operation on AVL Tree

Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. The new node is added into AVL tree as the leaf node. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing.

The tree can be balanced by applying rotations. Rotation is required only if, the balance factor of any node is disturbed upon inserting the new node, otherwise the rotation is not required.

[Type text]

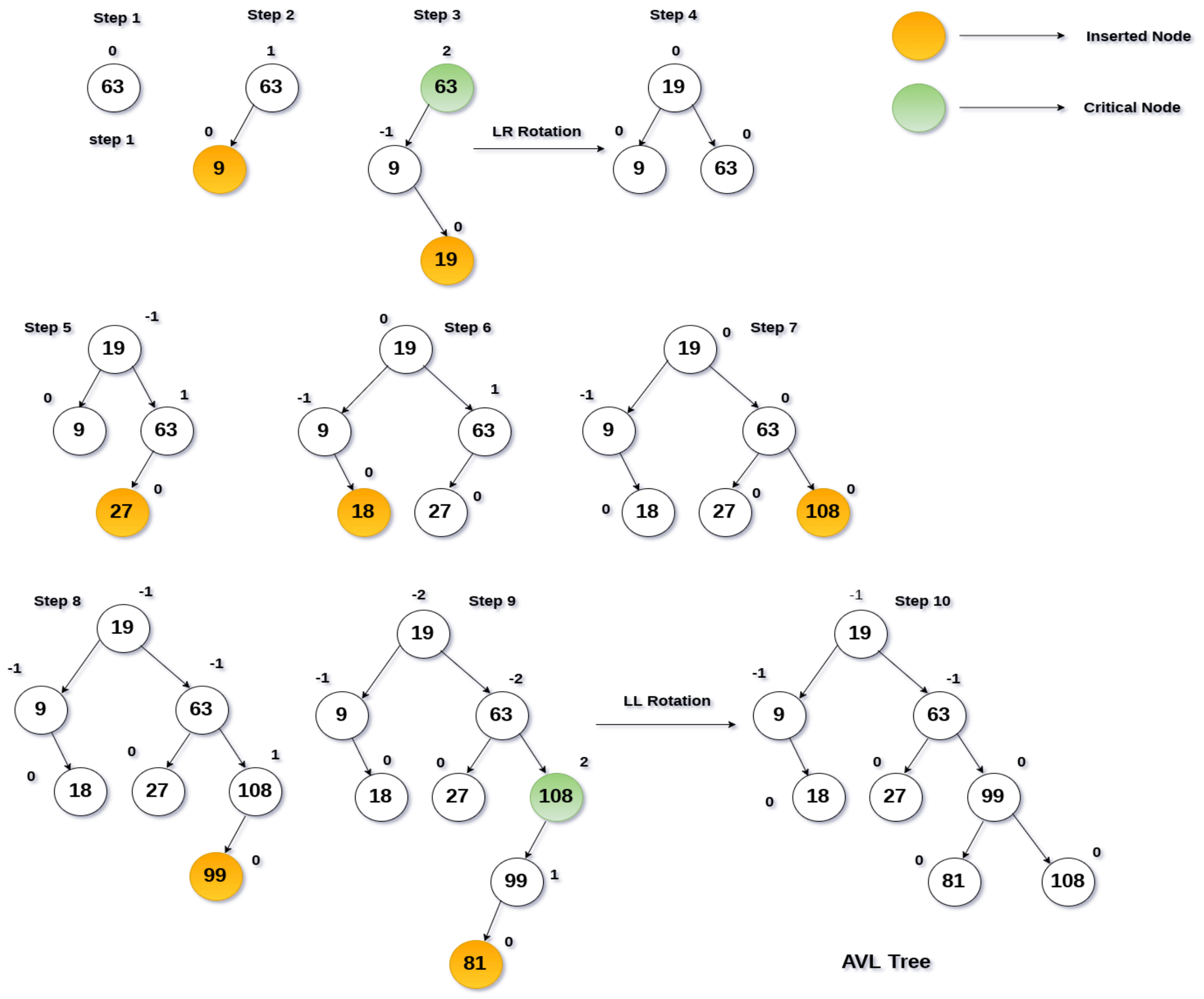
EXAMPLE

Construct an AVL tree by inserting the following elements in the given order.
63, 9, 19, 27, 18, 108, 99, 81

The process of constructing an AVL tree from the given set of elements is shown in the following figure.

At each step, we must calculate the balance factor for every node, if it is found to be more than 2 or less than -2, then we need a rotation to rebalance the tree. The type of rotation will be estimated by the location of the inserted element with respect to the critical node.

All the elements are inserted in order to maintain the order of binary search tree.



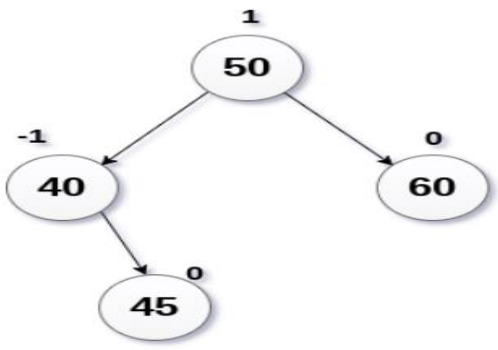
[Type text]

Deletion in AVL Tree

Deleting a node from an AVL tree is similar to that in a binary search tree. Deletion may disturb the balance factor of an AVL tree and therefore the tree needs to be rebalanced in order to maintain the AVLness. For this purpose, we need to perform rotations.

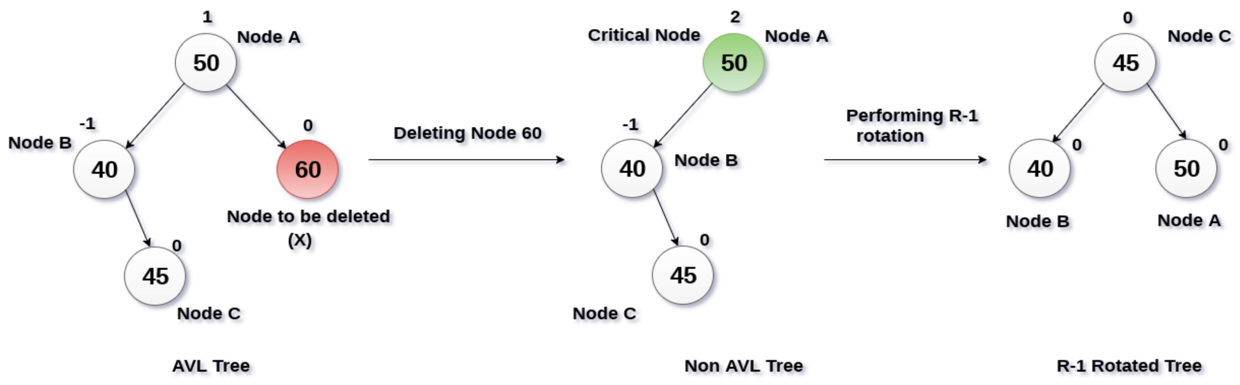
Example

Delete the node 60 from the AVL tree shown in the following image.



Solution:

in this case, node B has balance factor -1. Deleting the node 60, disturbs the balance factor of the node 50 therefore, it needs to be R-1 rotated. The node C i.e. 45 becomes the root of the tree with the node B(40) and A(50) as its left and right child.



Search Operation in AVL Tree

Search operation in AVL tree is similar to binary search tree search operation.

[Type text]

RED-BLACK TREE

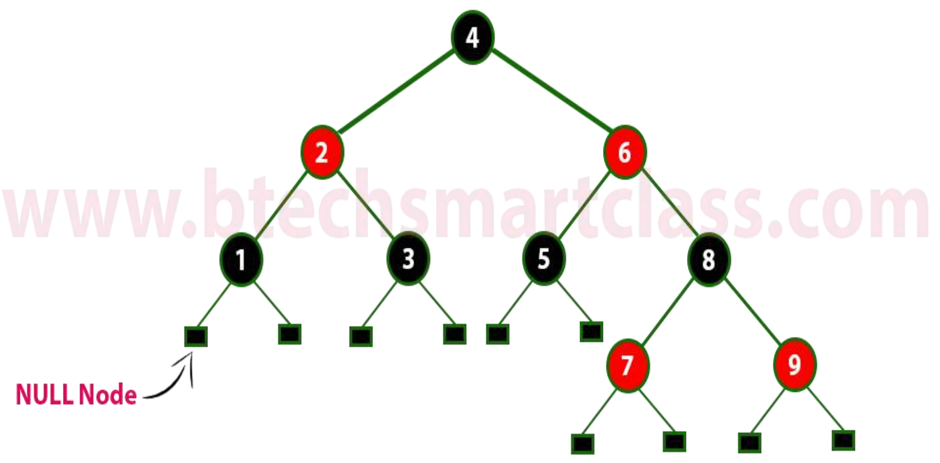
Introduction:

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black). These colors are used to ensure that the tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, This tree was invented in 1972 by Rudolf Bayer.

Rules That Every Red-Black Tree Follows:

1. Every node has a color either red or black.
2. The root of the tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes.
5. All leaf nodes are black nodes.

EXAMPLE



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

Why Red-Black Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that the height of the tree remains O(log n) after every insertion and deletion, then we can guarantee an upper bound of O(log n) for all these operations. The height of a Red-Black tree is always O(log n) where n is the number of nodes in the tree. Where **"n" is the total number of elements in the red-black tree.**

Comparison with AVL Tree:

The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves frequent insertions and deletions, then Red-Black trees should be preferred. And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over Red-Black Tree.

Interesting points about Red-Black Tree:

1. Black height of the red-black tree is the number of black nodes on a path from the root node to a leaf node. Leaf nodes are also counted as black nodes. So, a red-black tree of height h has black height >= h/2.
2. Height of a red-black tree with n nodes is $h \le 2 \log_2(n + 1)$.
3. All leaves (NIL) are black.
4. The black depth of a node is defined as the number of black nodes from the root to that node i.e the number of black ancestors.
5. Every red-black tree is a special case of a binary tree.

Black Height of a Red-Black Tree :

Black height is the number of black nodes on a path from the root to a leaf. Leaf nodes are also counted black nodes. From the above properties 3 and 4, we can derive, **a Red-Black Tree of height h has black-height >= h/2**.

NOTE: Every Red Black Tree with n nodes has height <= $2\text{Log}_2(n+1)$

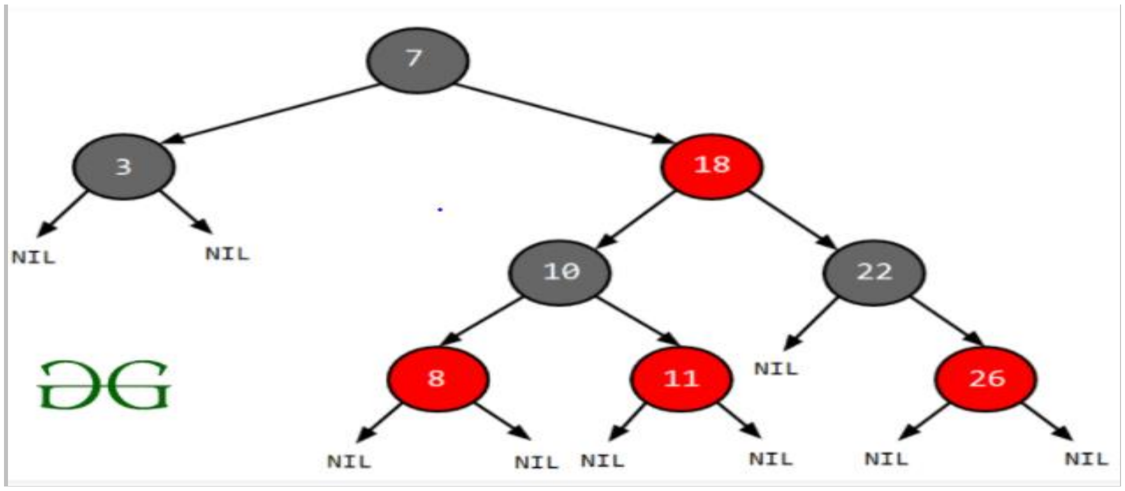
The following operations are performed on Red-Black Tree

1. Search
2. Insertion
3. Deletion

Search operation in Red-Black Tree

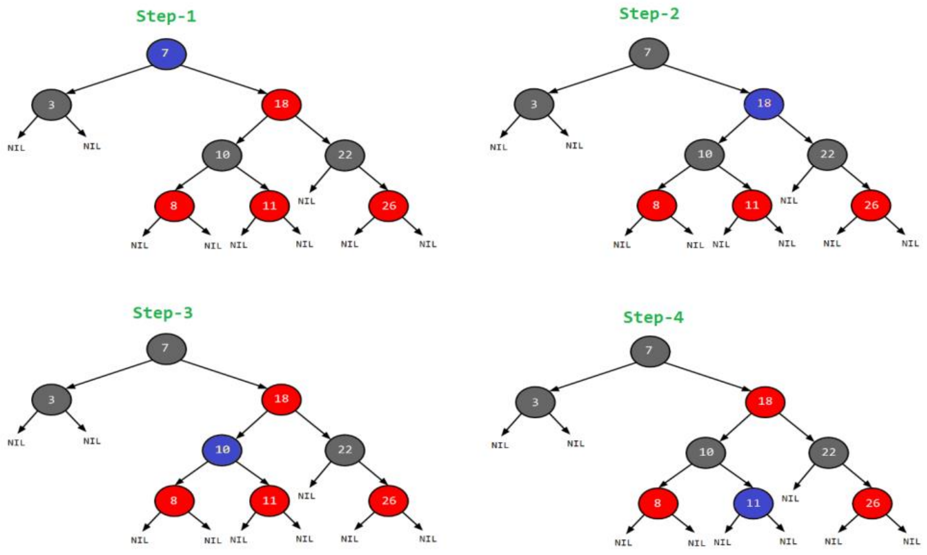
Every red-black tree is a special case of a binary tree so the searching algorithm of a red-black tree is similar to that of a binary tree.

Example: Searching 11 in the following red-black tree.



Solution:

1. Start from the root.
2. Compare the inserting element with root, if less than root, then recurse for left, else recurse for right.
3. If the element to search is found anywhere, return true, else return false.



Insertion operation in Red-Black Tree

In the Red-Black tree, we use two tools to do the balancing.

1. Recoloring
2. Rotation

Re-coloring is the change in color of the node i.e. if it is red then change it to black and vice versa. It must be noted that the color of the NULL node is always black. Moreover, we always try re-coloring first, if re-coloring doesn't work, then we go for rotation.

Following is a detailed algorithm. The algorithms have mainly two cases depending upon the color of the uncle(Uncle means new node parent sibling). If the uncle is red, we do recolor. If the uncle is black, we do rotations and/or re-coloring.

Logic:

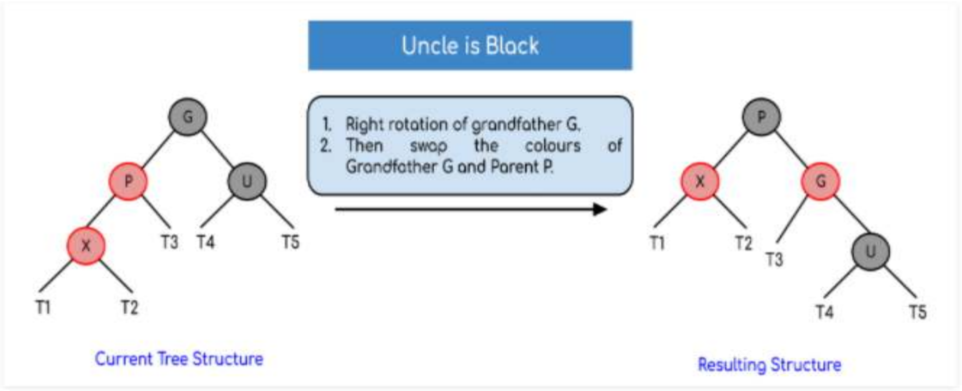
First, you have to insert the node similarly to that in a binary tree and assign a red color to it. Now, if the node is a root node then change its color to black, but if it is not then check the color of the parent node. If its color is black then don't change the color but if it is not i.e. it is red then check the color of the node's uncle. If the node's uncle has a red color then change the color of the node's parent and uncle to black and that of grandfather to red color and repeat the same process for him (i.e. grandfather).

Algorithm

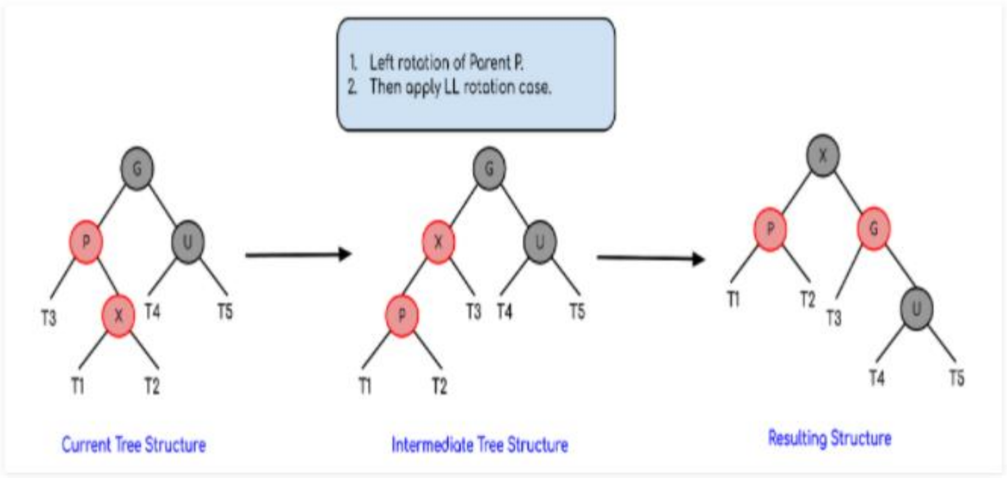
1. Perform standard BST insertion and make the color of newly inserted nodes as RED.
2. If new node (x) is the root, change the color of x as BLACK
3. Do the following if the color of new node (x's) parent is not BLACK **and** x is not the root.
   a) **If x's uncle(**Uncle means new node parent sibling) **is RED** (Grandparent must have been black from Red-Black Tree Property)
      - Change the colour of parent and uncle as BLACK.
      - Colour of a grandparent as RED.
      - Change x = x's grandparent, repeat steps 2 and 3 for new x.

   b). **If x's uncle is BLACK**, then there can be four configurations for x, x's parent (**p**) and x's grandparent (**g**)
   - Left Left Case (p is left child of g and x is left child of p)
   - Left Right Case (p is left child of g and x is the right child of p)
   - Right Right Case
   - Right Left Case

- Left Left Case (LL rotation):

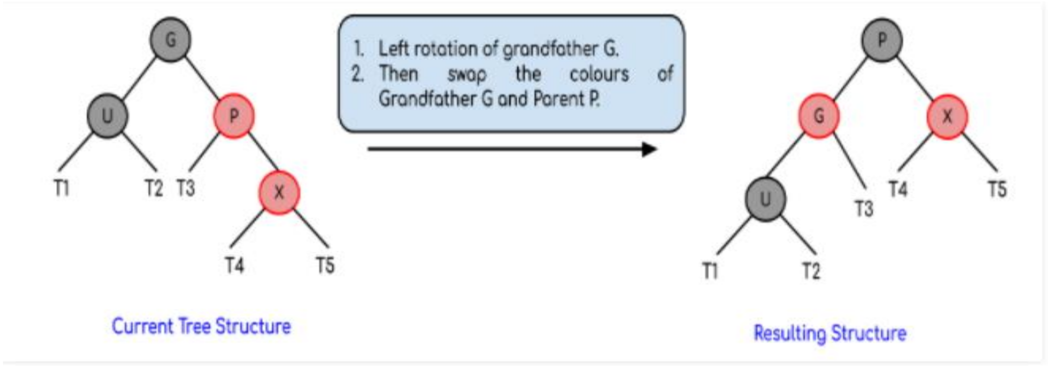

Uncle is Black
1. Right rotation of grandfather G.
2. Then swap the colours of Grandfather G and Parent P.
G
P
U
X
T1
T2
T3
T4
T5
Current Tree Structure
P
X
G
U
Resulting Structure

- Left Right Case (LR rotation):

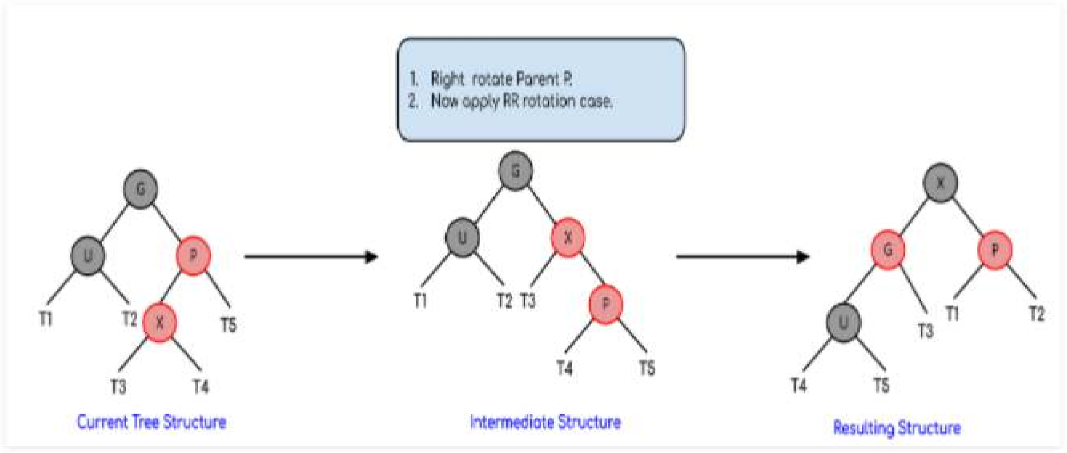

1. Left rotation of Parent P.
2. Then apply LL rotation case.
G
P
U
X
T1
T2
T3
T4
T5
Current Tree Structure
Intermediate Tree Structure
Resulting Structure

➢

- Right Right Case (RR rotation):


1. Left rotation of grandfather G.
2. Then swap the colours of Grandfather G and Parent P.
G
U
P
X
T1
T2
T3
T4
T5
Current Tree Structure
Resulting Structure

- Right Left Case (RL rotation):

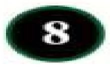


EXAMPLE:

Create a Red-Black Tree with the following sequence of numbers 8,18,5,15,17,25,40 and 80

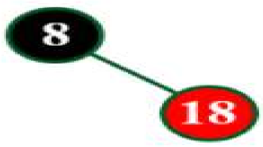
insert (8)

Tree is Empty. So insert newNode as Root node with black color.



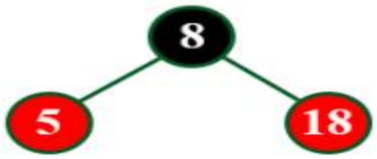
insert (18)

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



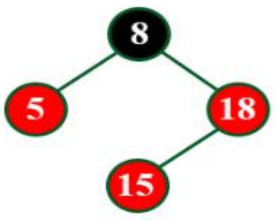
insert (5)

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



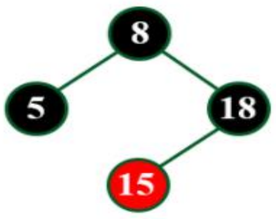
insert (15)

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
and parent's parent is root node.
So we use RECOLOR to make it Red Black Tree.

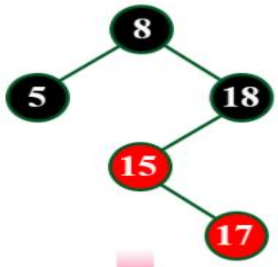
After RECOLOR



After Recolor operation, the tree is satisfying all Red Black Tree properties.

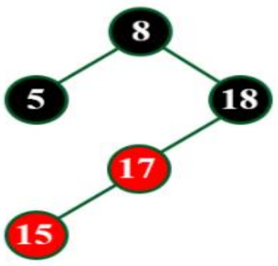
insert (17)

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

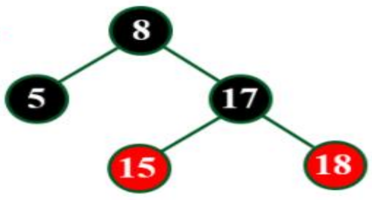


Here there are two consecutive Red nodes (15 & 17).
The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.

After Left Rotation

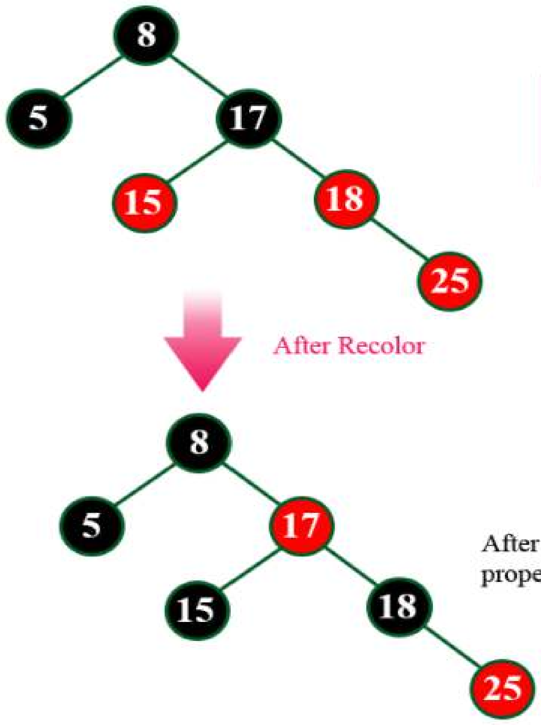


After Right Rotation & Recolor



insert (25)

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

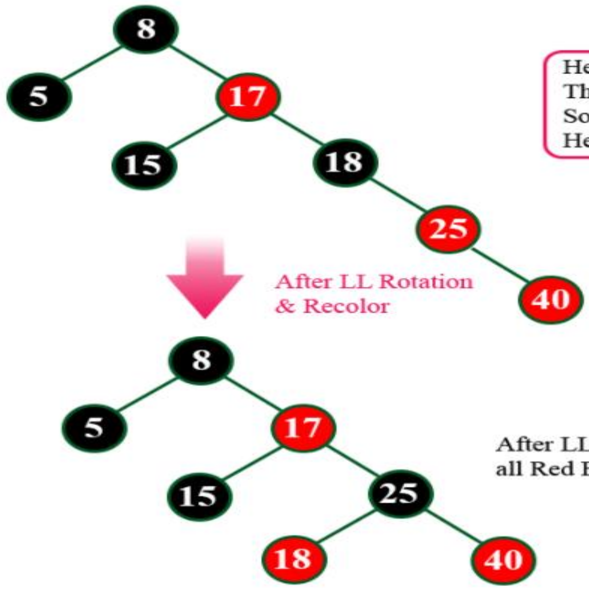


Here there are two consecutive Red nodes (18 & 25).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

After Recolor operation, the tree is satisfying all Red Black Tree properties.

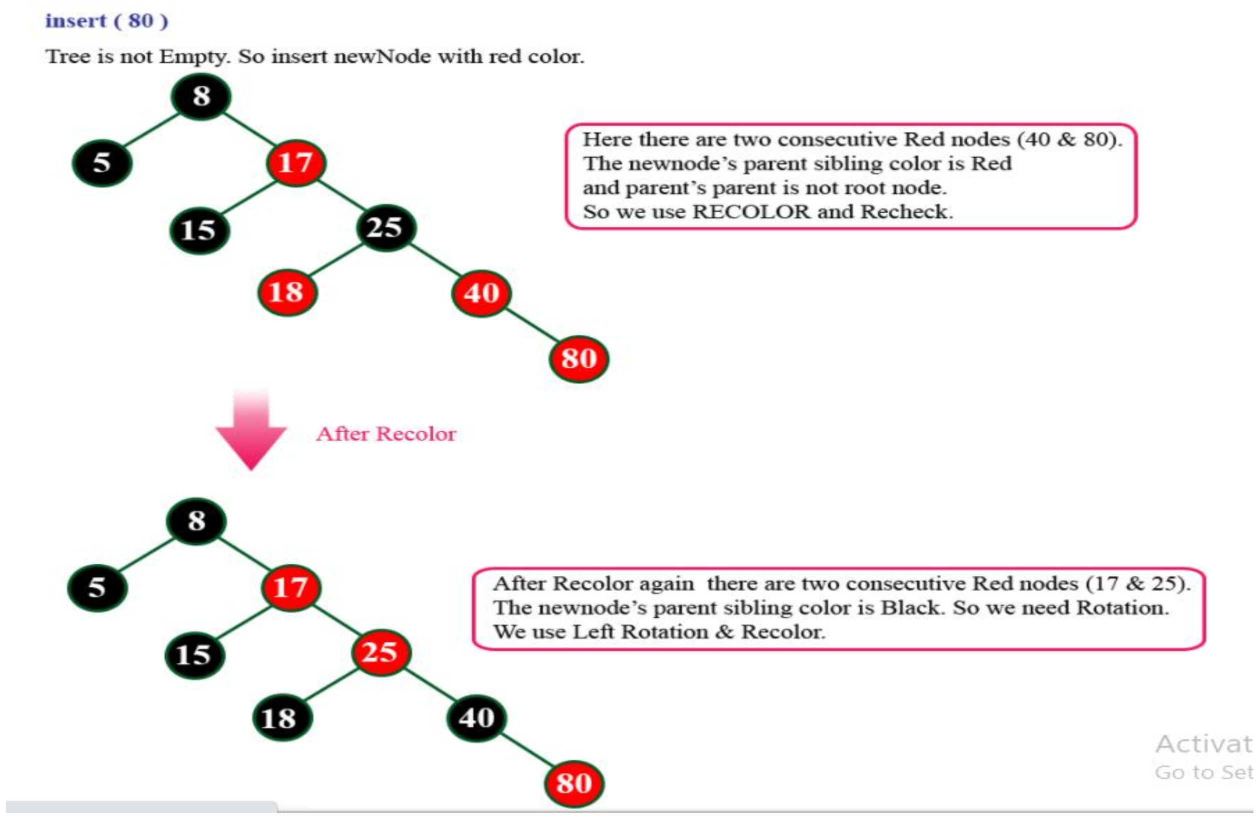
insert (40)

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
So we need a Rotation & Recolor.
Here, we use LL Rotation and Recheck.

After LL Rotation & Recolor operation, the tree is satisfying all Red Black Tree properties.

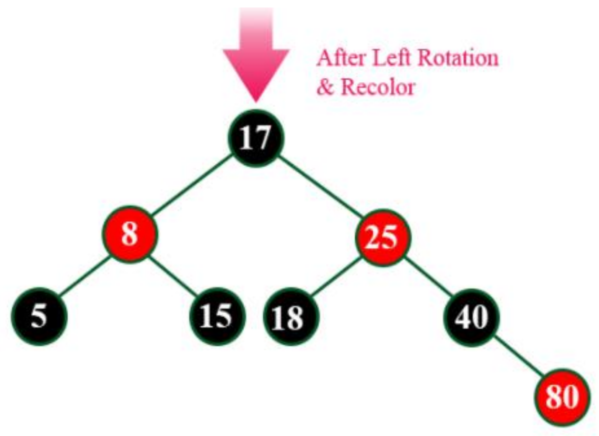


insert (80)

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
and parent's parent is not root node.
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. So we need Rotation.
We use Left Rotation & Recolor.



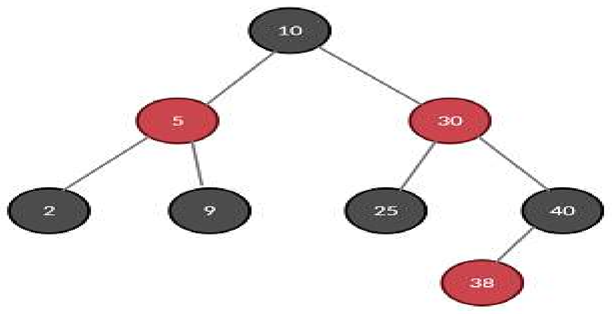
Finally above tree is satisfying all the properties of Red Black Tree and
it is a perfect Red Black tree.

Deletion in Red Black Tree

The below table is useful to identify the case and its corresponding set of actions to be performed when deleting a node from Red Black Tree

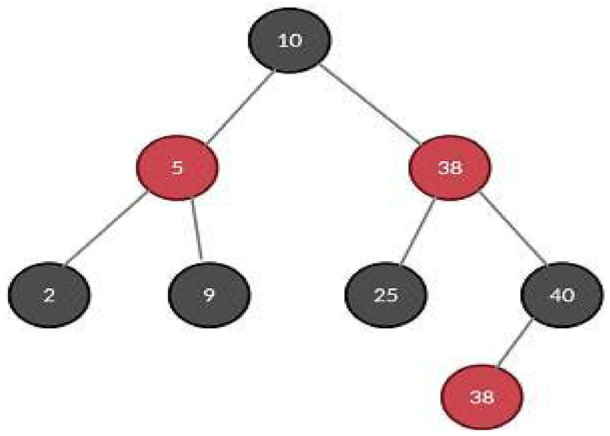
Case #	Check condition	Action
1	If node to be delete is a red leaf node	Just remove it from the tree
2	If DB node is root	Remove the DB and root node becomes black.
3	(a) If DB's sibling is black, and (b) DB's sibling's children are black	(a) Remove the DB (if null DB then delete the node and for other nodes remove the DB sign) (b) Make DB's sibling red. (c) If DB's parent is black, make it DB, else make it black
4	If DB's sibling is red	(a) Swap color DB's parent with DB's sibling (b) Perform rotation at parent node in the direction of DB node (c) Check which case can be applied to this new tree and perform that action
5	(a) DB's sibling is black (b) DB's sibling's child which is far from DB is black (c) DB's sibling's child which is near to DB is red	(a) Swap color of sibling with sibling's red child (b) Perform rotation at sibling node in direction opposite of DB node (c) Apply case 6
6	(a) DB's sibling is black, and (b) DB's sibling's far child is red (remember this node)	(a) Swap color of DB's parent with DB's sibling's color (b) Perform rotation at DB's parent in direction of DB (c) Remove DB sign and make the node normal black node (d) Change colour of DB's sibling's far red child to black.

Example 1: Delete 30 from the RB tree in fig. 3



Initial RB Tree

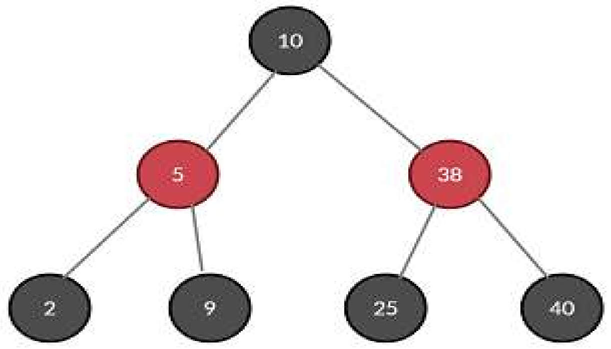
You first have to search for 30, once found perform BST deletion . For a node with value '30', find either the maximum of the left subtree or a minimum of the right subtree and replace 30 with that value. This is BST deletion .



RB Tree after replacing 30 with min element from right subtree

The resulting RB tree will be like one in fig. 4. Element 30 is deleted and the value is successfully replaced by 38. But now the task is to delete duplicate element 38.

Go to the table above and you'll observe **case 1** is satisfied by this tree.

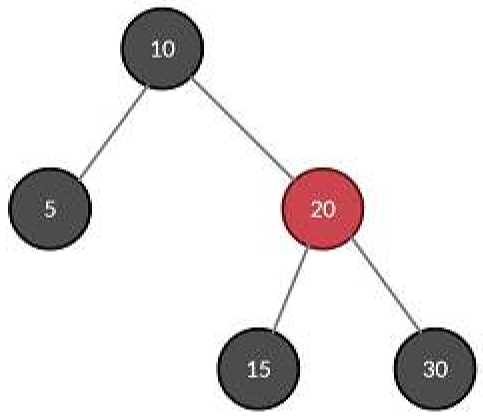


After removing the red leaf node

Since node with element 38 is a *red* leaf node, remove it and the tree looks like the one in fig. 5.

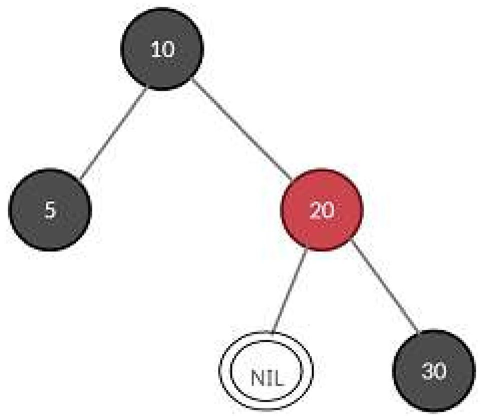
Observe that if you perform correct actions, the tree will still hold all the properties of the RB tree.

Example 2: Delete 15 from below RB tree



Initial RB Tree

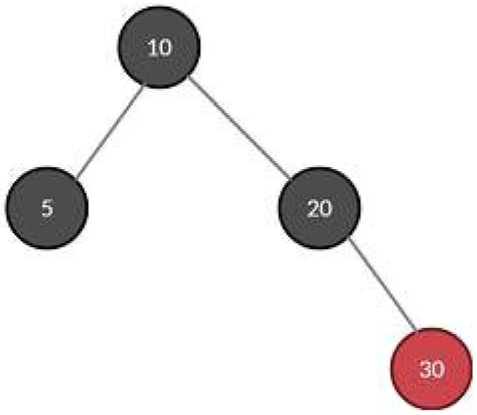
15 can be removed easily from the tree (BST deletion). In the case of RB trees, if a leaf node is deleted you replace it with a **double black (DB)** nil node . It is represented by a double circle.



NIL node added in place of 15

The entire problem is now drilled down to get rid of this bad boy, DB, via some actions.

Go back to our rule book (table) and **case 3** fits perfectly.



NIL Node removed after applying actions

In short, remove DB and then swap the color of its sibling with its parent

Example 3: Delete '15' from fig.(A).

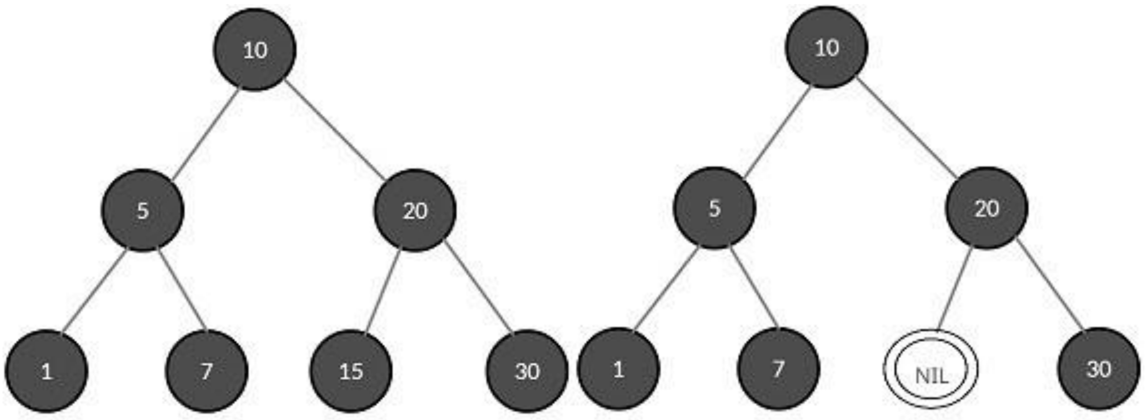


Fig: (A) Initial RB Tree, (B) NIL node added in place of 15

Delete node with value 15 and, as a rule, replace it with DB nil node as shown. Now, DB's sibling is black and sibling's both children are also black (don't forget the hidden NIL nodes!), it satisfies all the conditions of case 3. Here,

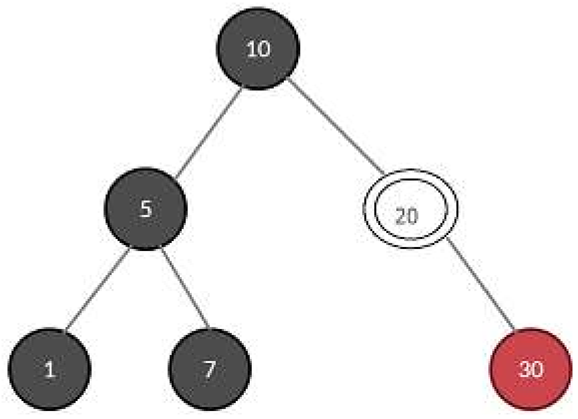


Fig: RB Tree after case 3 is applied

1. DB’s parent is 20
2. DB’s parent is *black*
3. DB’s sibling is 30

With these points in mind perform the actions and you get an RB tree as in fig. 10.

20 becomes DB and hence the problem is not resolved yet. Reapply case 3

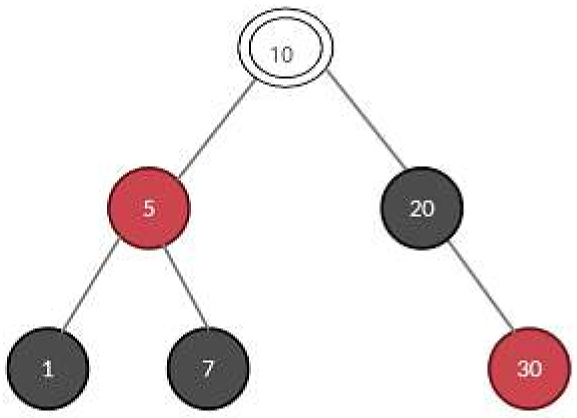


Fig. RB Tree after case 3 is applied

The resulting tree looks like the one in the above fig.

The DB still exist . Recheck which case will be applicable.

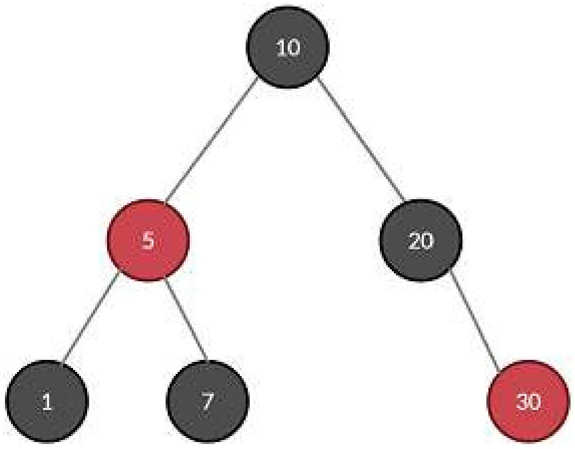
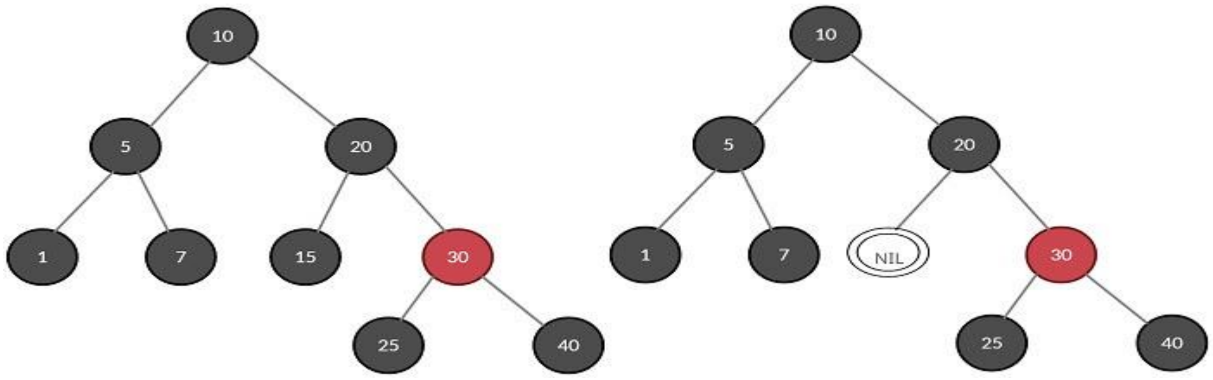


Fig.: NIL Node removed after applying actions

Found it? It's case 2, the simplest of all!

The root resolved DB and becomes a *black* node. And you're done deleting 15 successfully.

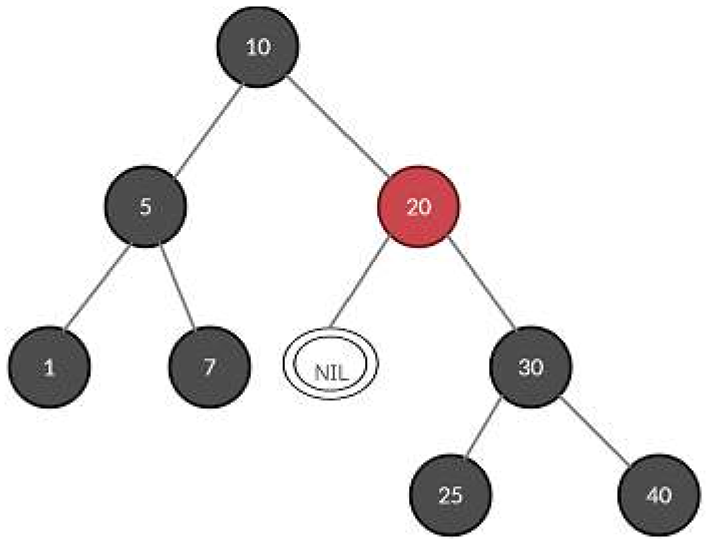
Example 4: Delete '15' from below fig. (A).



(A) Initial RB Tree, (B) NIL node added in place of 15

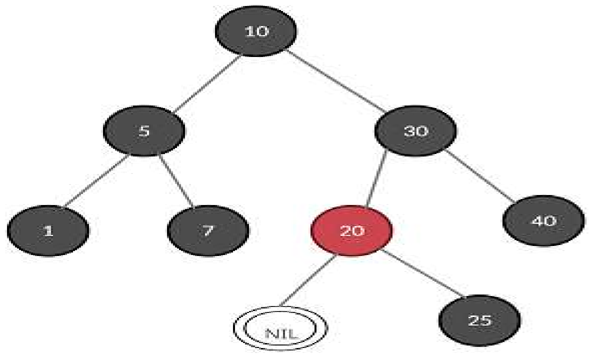
First, Search 15 as per BST rules and then delete it. Second, replace deleted node with DB NIL node as shown in fig. 13 (B).

DB's sibling is *red.* Clearly**, case 4** is applicable.



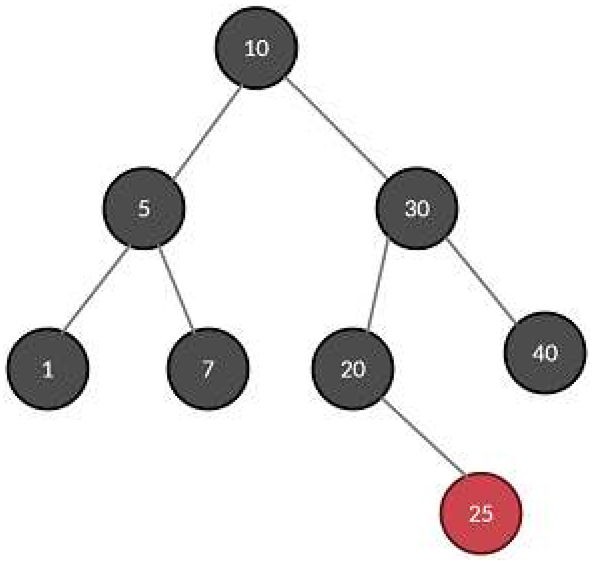
RB Tree after case 4 is applied

(a) Swap DB’s parent’s color with DB’s sibling’s color. I know this is confusing, but take it easy and keep following. The tree looks like fig. 14.



(b) Perform rotation at parent node in direction of DB. The tree becomes like the one in fig. 15. DB is still there (what’s its problem!).

(c) Check which case can be applied in the current tree. And got it, **case 3.**

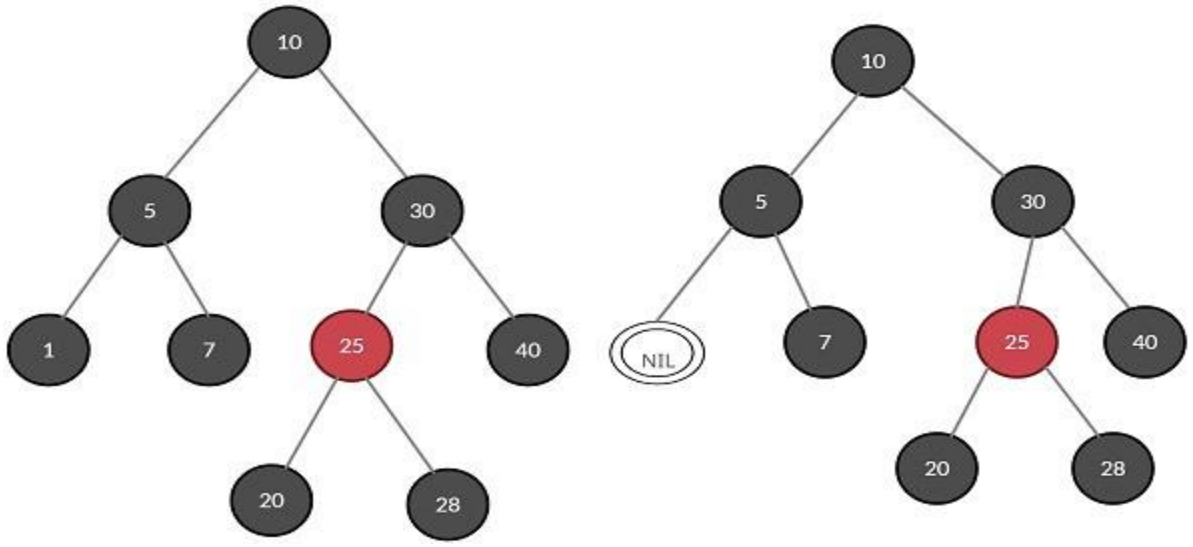


NIL Node removed after applying actions

(d) Apply case 3 as explained and the RB tree is free from the DB node as shown in fig. 16.

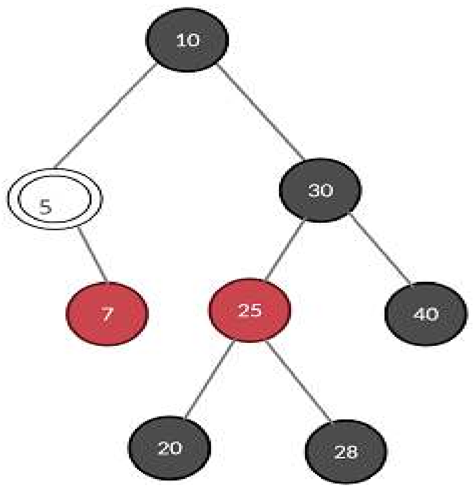
I know it's tiresome, but I swear if you practice these examples 2–3 times, you will have a good grasp of the concept of deletion in RB trees.

Example 5: Delete '1' from below fig(A).



(A) Initial RB Tree, (B) NIL node added in place of 1

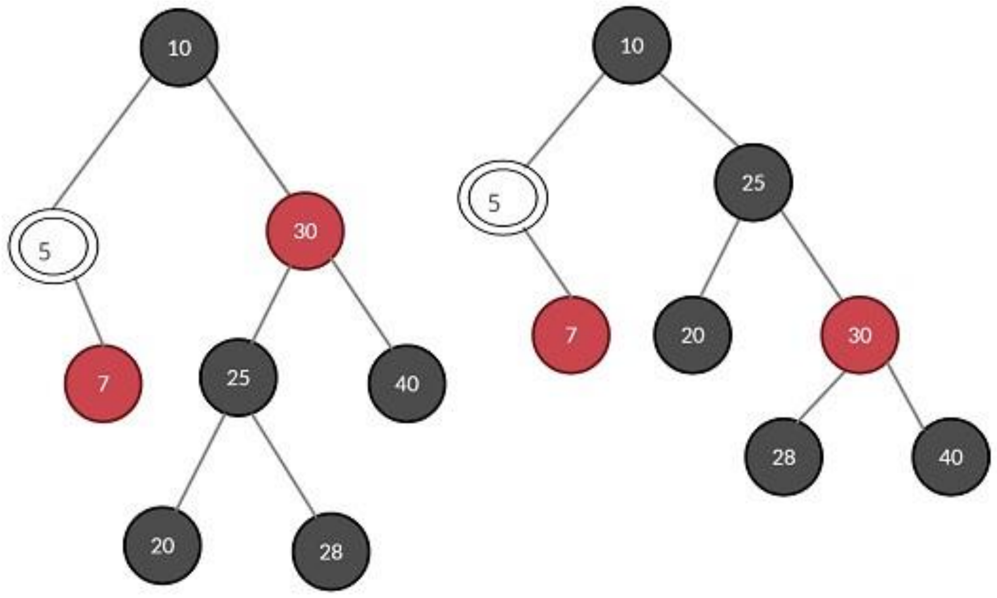
Perform the basic preliminary steps- delete the node with value 1 and replace it with DB NIL node as shown in fig. 17(B). Check for the cases which fit the current tree and it's case 3(DB's sibling is *black*).



RB Tree after case 3 is applied

Node 5 has now become a *double black* node. We need to get rid of it.

Search for cases that can be applied and case 5 seems to fit here (not case 3).



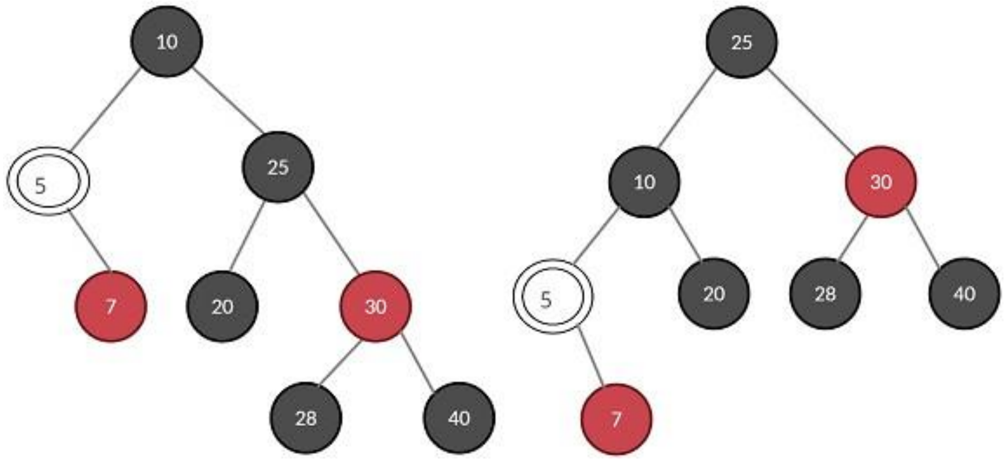
(A) Tree after swapping colors of 30 & 25 (B) Tree after rotation

Case 5 is applied as follows-

(a) swap colors of nodes 30 and 25 (fig. 19(A))

(b) Rotate at sibling node in the direction opposite to the DB node. Hence, perform right rotation at node 30 and the tree becomes like fig. 19 (B).

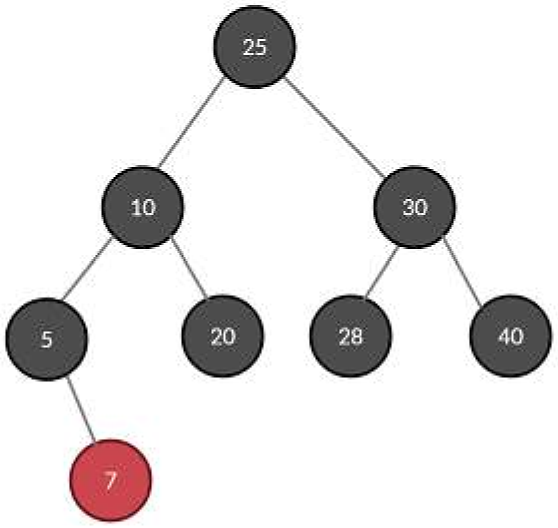
The *double black* node still haunts the tree! Re-check the case that can be applied to this tree and we find that case 6 (don't fall for case 3) seems to fit.



Apply **case 6** as follow-

(a) Swap colors of DB's parent with DB's sibling.

(b) Perform rotation at DB's parent node in the direction of DB (fig, 20(B)).



NIL Node removed after applying actions

(c) Change DB node to *black* node. Also, change the color of DB's sibling's far-*red* child to black and the final RB tree will look fig. 21.

And, voilà! The RB tree is free of element 1 as well as of any *double node*. Life is good now.

Applications of Red-Black Trees

Real-world uses of red-black trees include TreeSet, TreeMap, and Hashmap in the Java Collections Library.

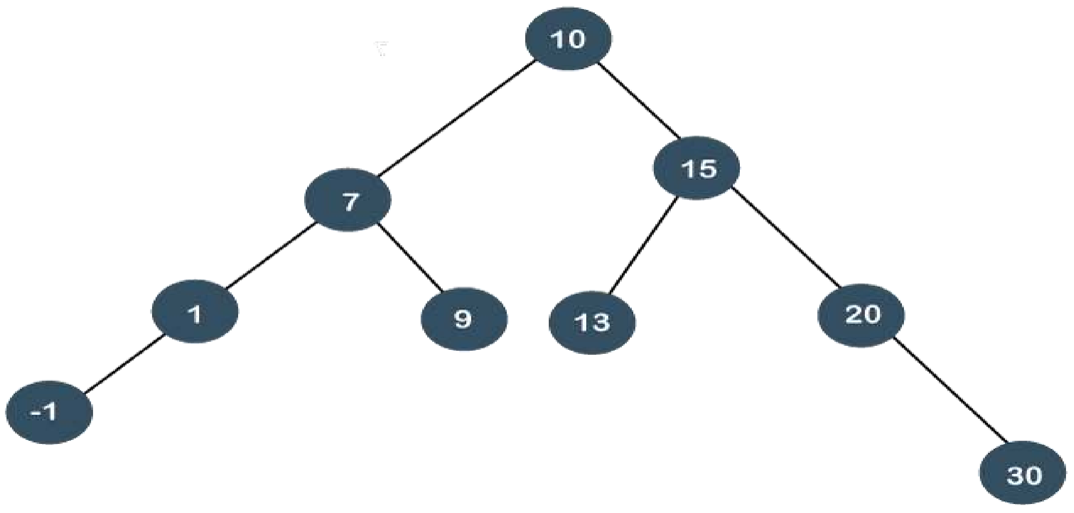
Splay Tree

Splay trees are the self-balancing or self-adjusted binary search trees. In other words, we can say that the splay trees are the variants of the binary search trees. The prerequisite for the splay trees that we should know about the binary search trees.

As we already know, the time complexity of a binary search tree in every case. The time complexity of a binary search tree in the average case is **O(logn)** and the time complexity in the worst case is O(n). In a binary search tree, the value of the left subtree is smaller than the root node, and the value of the right subtree is greater than the root node; in such case, the time complexity would be **O(logn)**. If the binary tree is left-skewed or right-skewed, then the time complexity would be O(n). To limit the skewness, the AVL and Red-Black tree came into the picture, having **O(logn**) time complexity for all the operations in all the cases. We can also improve this time complexity by doing more practical implementations, so the new Tree data structure was designed, known as a Splay tree.

What is a Splay Tree?

A splay tree is a self-balancing tree, but AVL and Red-Black trees are also self-balancing trees then. What makes the splay tree unique two trees. It has one extra property that makes it unique is splaying.
A splay tree contains the same operations as a Binary search tree, i.e., Insertion, deletion and searching, but it also contains one more operation, i.e., splaying. So. all the operations in the splay tree are followed by splaying.
Splay trees are not strictly balanced trees, but they are roughly balanced trees. Let's understand the search operation in the splay-tree.
Suppose we want to search 7 element in the tree, which is shown below:



To search any element in the splay tree, first, we will perform the standard binary search tree operation. As 7 is less than 10 so we will come to the left of the root node. After performing the search operation, we need to perform splaying. Here splaying means that the operation that we are performing on any element should become the root node after performing some rearrangements. The rearrangement of the tree will be done through the rotations.

Note: The splay tree can be defined as the self-adjusted tree in which any operation performed on the element would rearrange the tree so that the element on which operation has been performed becomes the root node 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.

In a splay tree, splaying an element rearranges all the elements in the tree so that splayed element is placed at the root of the tree.

By splaying elements we bring more frequently used elements closer to the root of the tree so that any operation on those elements is performed quickly. That means the splaying operation automatically brings more frequently used elements closer to the root of the tree.

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.

In splay tree, to splay any element we use the following rotation operations...

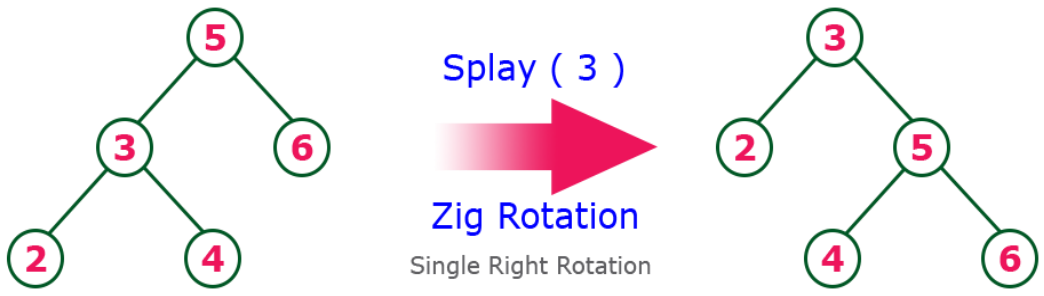
Rotations in Splay Tree

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

Example

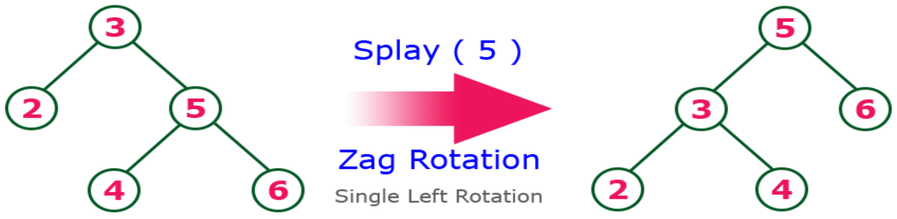
Zig Rotation

The **Zig Rotation** in splay tree is similar to the single right rotation in AVL Tree rotations. In zig rotation, every node moves one position to the right from its current position. Consider the following example...



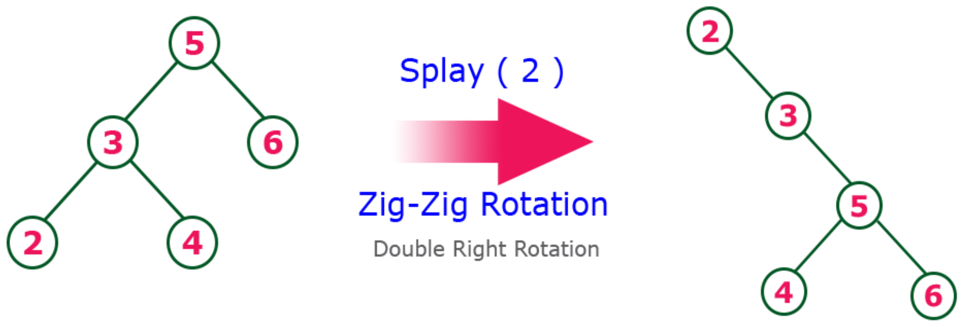
Zag Rotation

The **Zag Rotation** in splay tree is similar to the single left rotation in AVL Tree rotations. In zag rotation, every node moves one position to the left from its current position. Consider the following example...



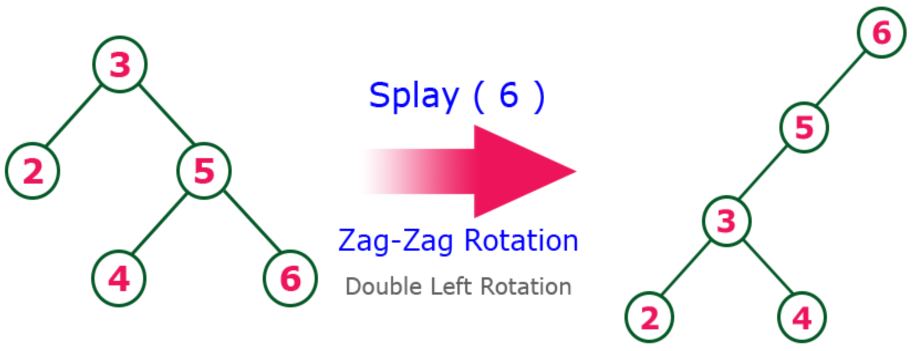
Zig-Zig Rotation

The **Zig-Zig Rotation** in splay tree is a double zig rotation. In zig-zig rotation, every node moves two positions to the right from its current position. Consider the following example...



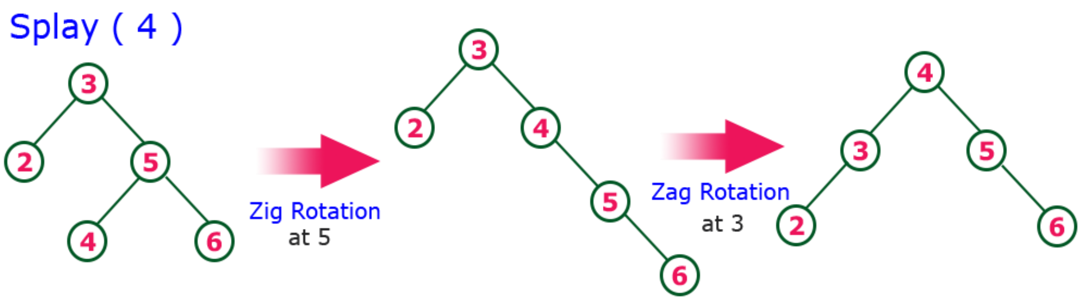
Zag-Zag Rotation

The **Zag-Zag Rotation** in splay tree is a double zag rotation. In zag-zag rotation, every node moves two positions to the left from its current position. Consider the following example...



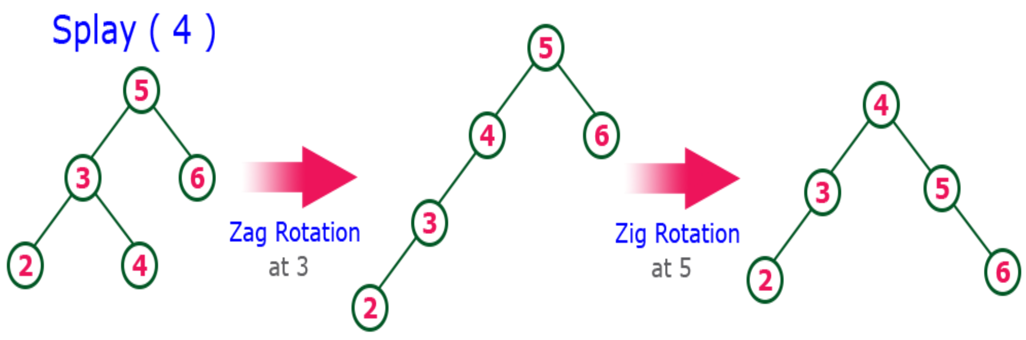
Zig-Zag Rotation

The **Zig-Zag Rotation** in splay tree is a sequence of zig rotation followed by zag rotation. In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position. Consider the following example...



Zag-Zig Rotation

The **Zag-Zig Rotation** in splay tree is a sequence of zag rotation followed by zig rotation. In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position. Consider the following example...



Rotations

There are six types of rotations used for splaying:

1. Zig rotation (Right rotation)
2. Zag rotation (Left rotation)
3. Zig zag (Zig followed by zag)
4. Zag zig (Zag followed by zig)
5. Zig zig (two right rotations)
6. Zag zag (two left rotations)

Factors required for selecting a type of rotation

The following are the factors used for selecting a type of rotation:

- Does the node which we are trying to rotate have a grandparent?
- Is the node left or right child of the parent?
- Is the node left or right child of the grandparent?

Cases for the Rotations

Case 1: If the node does not have a grand-parent, and if it is the right child of the parent, then we carry out the left rotation; otherwise, the right rotation is performed.

Case 2: If the node has a grandparent, then based on the following scenarios; the rotation would be performed:

Scenario 1: If the node is the right of the parent and the parent is also right of its parent, then ***zig zig right right rotation*** is performed.

Scenario 2: If the node is left of a parent, but the parent is right of its parent, then ***zig zag right left rotation*** is performed.

Scenario 3: If the node is right of the parent and the parent is right of its parent, then ***zig zig left left rotation*** is performed.

Scenario 4: If the node is right of a parent, but the parent is left of its parent, then ***zig zag right-left rotation*** is performed.

Now, let's understand the above rotations with examples.

To rearrange the tree, we need to perform some rotations. The following are the types of rotations in the splay tree:

- **Zig rotations**

The zig rotations are used when the item to be searched is either a root node or the child of a root node (i.e., left or the right child).

The following are the cases that can exist in the splay tree while searching:

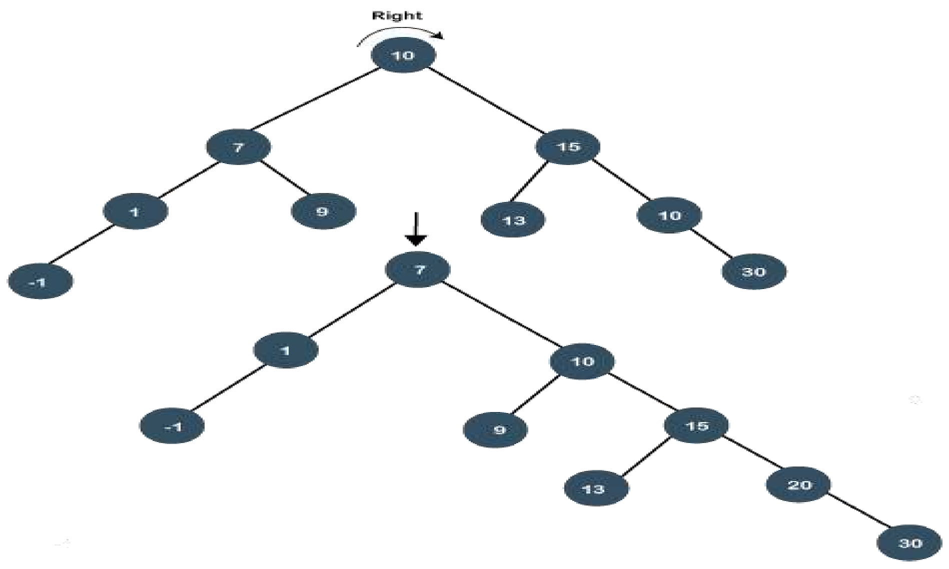
Case 1: If the search item is a root node of the tree.

Case 2: If the search item is a child of the root node, then the two scenarios will be there:

1. If the child is a left child, the right rotation would be performed, known as a zig right rotation.
2. If the child is a right child, the left rotation would be performed, known as a zig left rotation.

Let's look at the above two scenarios through an example.

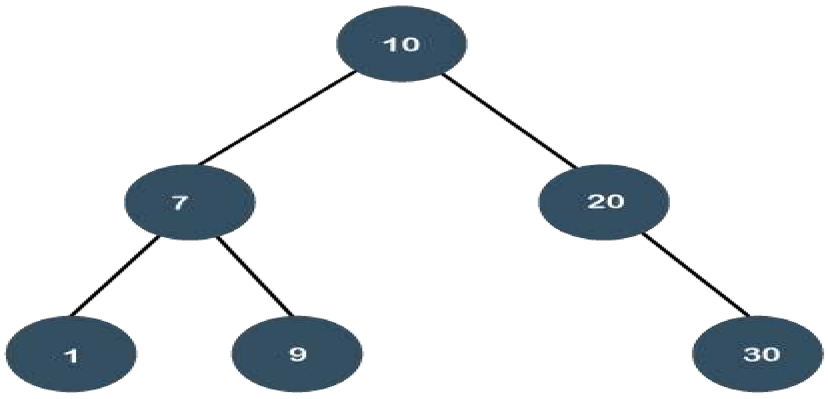
Consider the below example:
In the above example, we have to search 7 element in the tree. We will follow the below steps:
Step 1: First, we compare 7 with a root node. As 7 is less than 10, so it is a left child of the root node.
Step 2: Once the element is found, we will perform splaying. The right rotation is performed so that 7 becomes the root node of the tree, as shown below:



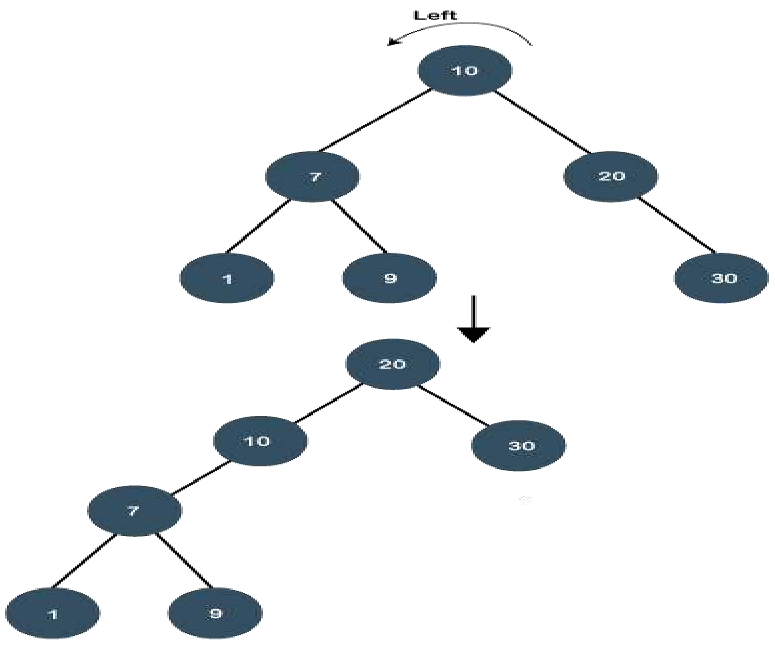
Let's consider another example.

In the above example, we have to search 20 element in the tree. We will follow the below steps:

Step 1: First, we compare 20 with a root node. As 20 is greater than the root node, so it is a right child of the root node.



Step 2: Once the element is found, we will perform splaying. The left rotation is performed so that 20 element becomes the root node of the tree.



- **Zig zig rotations**

Sometimes the situation arises when the item to be searched is having a parent as well as a grandparent. In this case, we have to perform four rotations for splaying.

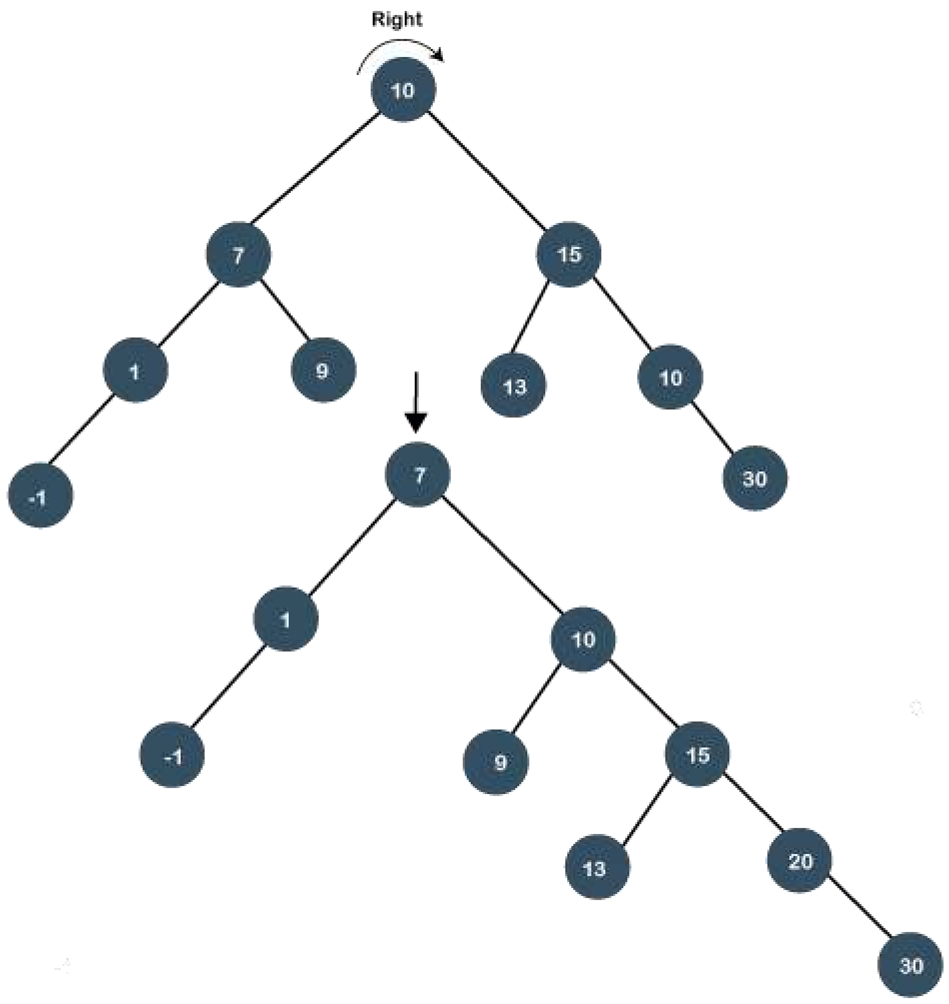
Let's understand this case through an example.

Suppose we have to search 1 element in the tree, which is shown below:

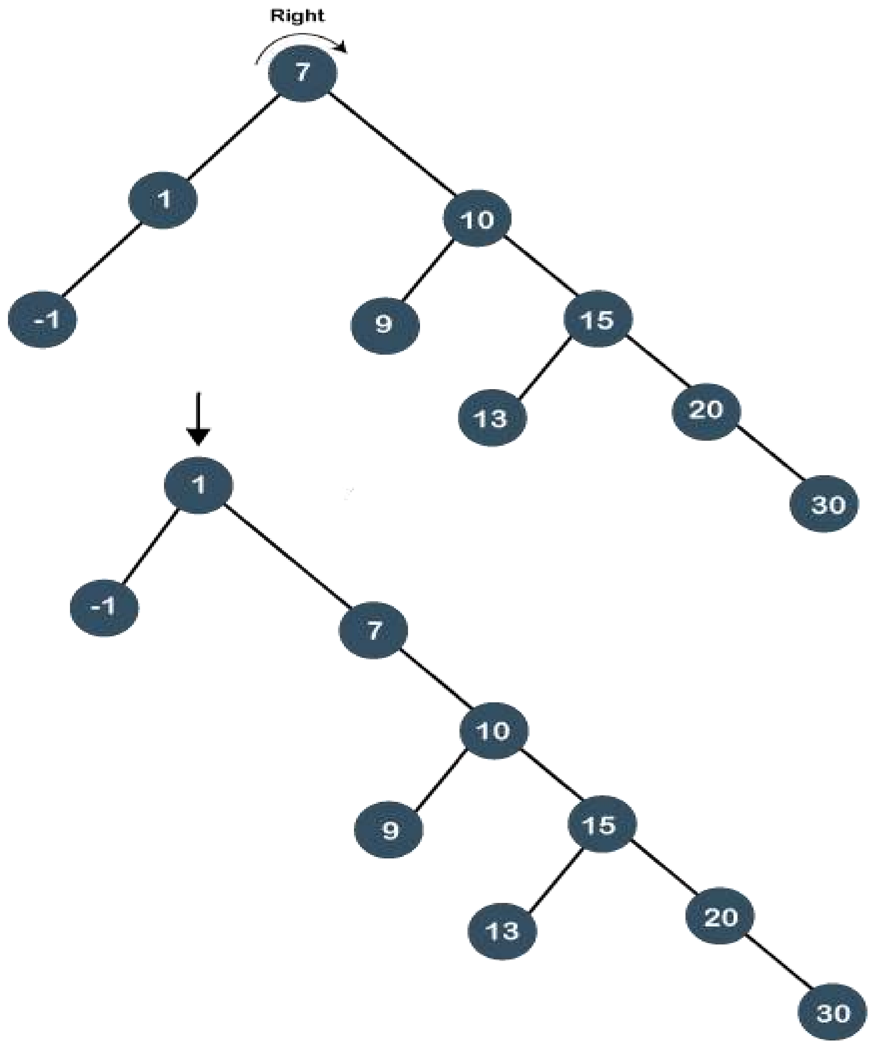
Step 1: First, we have to perform a standard BST searching operation in order to search the 1 element. As 1 is less than 10 and 7, so it will be at the left of the node 7. Therefore, element 1 is having a parent, i.e., 7 as well as a grandparent, i.e., 10.

Step 2: In this step, we have to perform splaying. We need to make node 1 as a root node with the help of some rotations. In this case, we cannot simply perform a zig or zag rotation; we have to implement zig zig rotation.

In order to make node 1 as a root node, we need to perform two right rotations known as zig zig rotations. When we perform the right rotation then 10 will move downwards, and node 7 will come upwards as shown in the below figure:



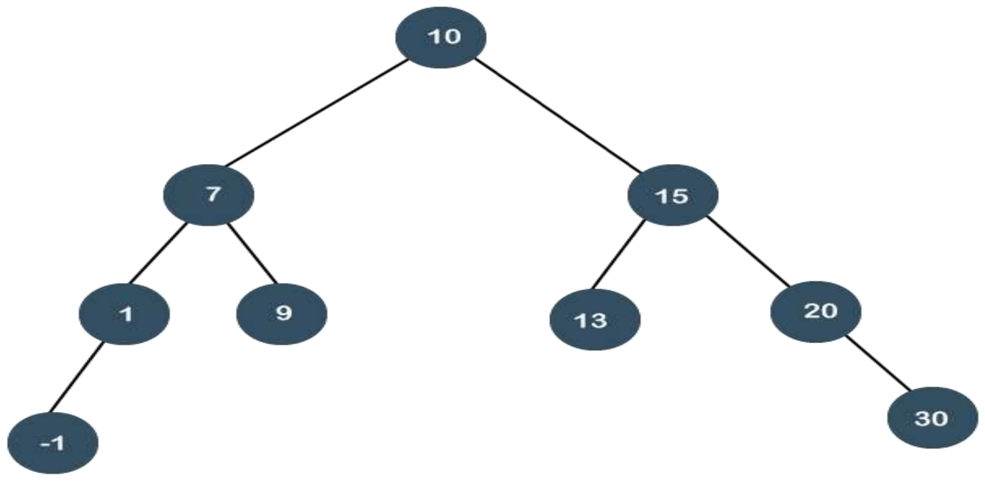
Again, we will perform zig right rotation, node 7 will move downwards, and node 1 will come upwards as shown below:



As we observe in the above figure that node 1 has become the root node of the tree; therefore, the searching is completed.

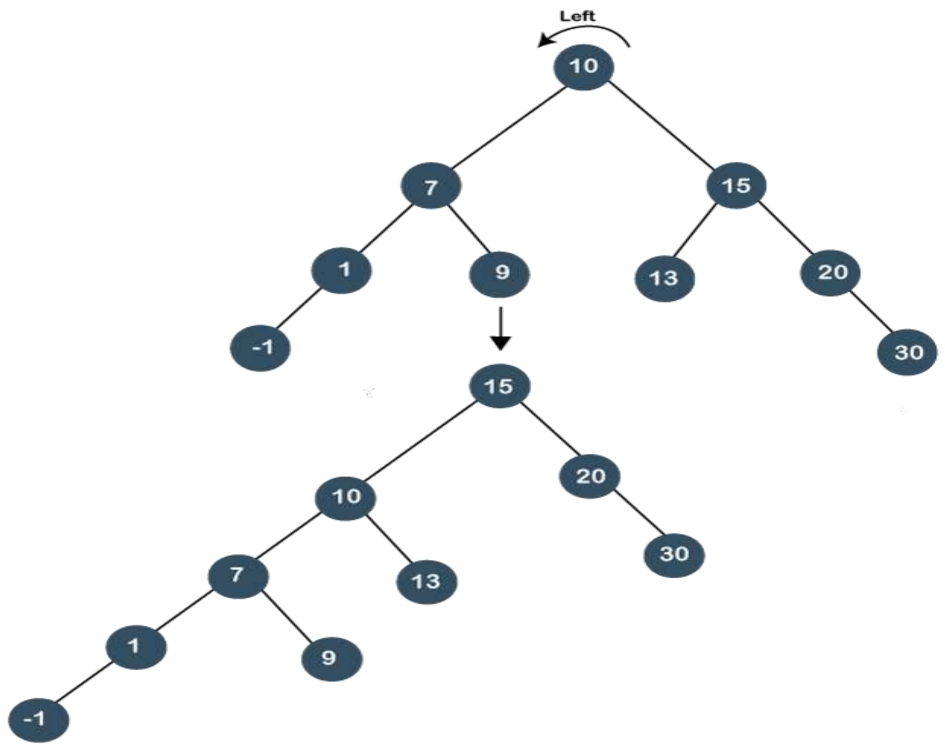
Suppose we want to search 20 in the below tree.

In order to search 20, we need to perform two left rotations. Following are the steps required to search 20 node:

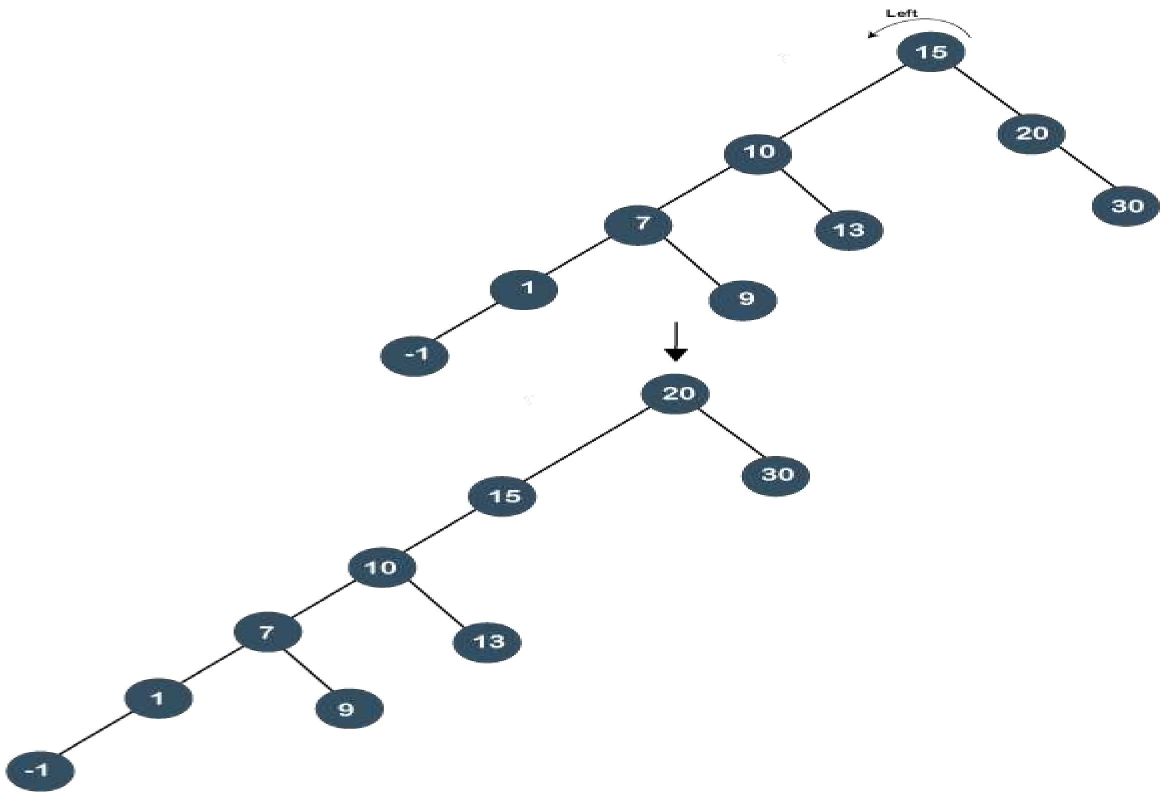


Step 1: First, we perform the standard BST searching operation. As 20 is greater than 10 and 15, so it will be at the right of node 15.

Step 2: The second step is to perform splaying. In this case, two left rotations would be performed. In the first rotation, node 10 will move downwards, and node 15 would move upwards as shown below:



In the second left rotation, node 15 will move downwards, and node 20 becomes the root node of the tree, as shown below:



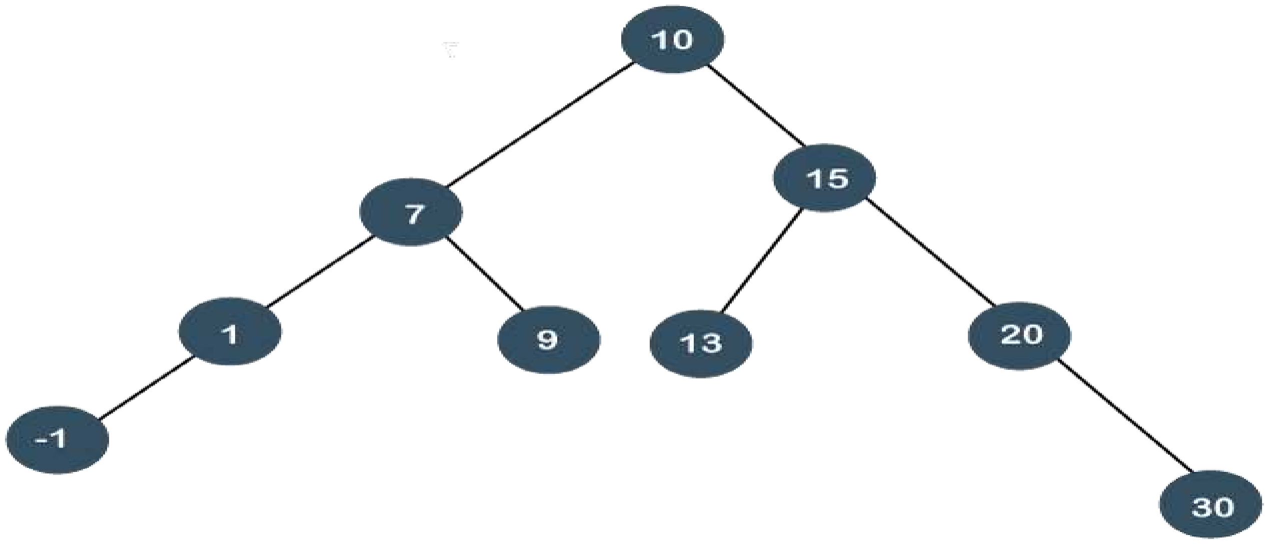
As we have observed that two left rotations are performed; so it is known as a zig zig left rotation.

- **Zig zag rotations**

Till now, we have read that both parent and grandparent are either in RR or LL relationship. Now, we will see the RL or LR relationship between the parent and the grandparent.

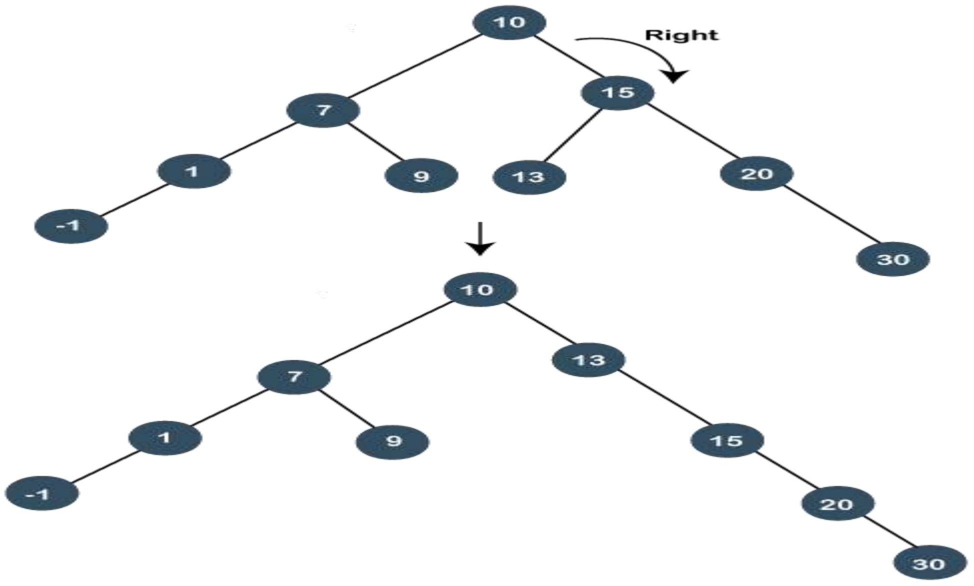
Let's understand this case through an example.

Suppose we want to search 13 element in the tree which is shown below:

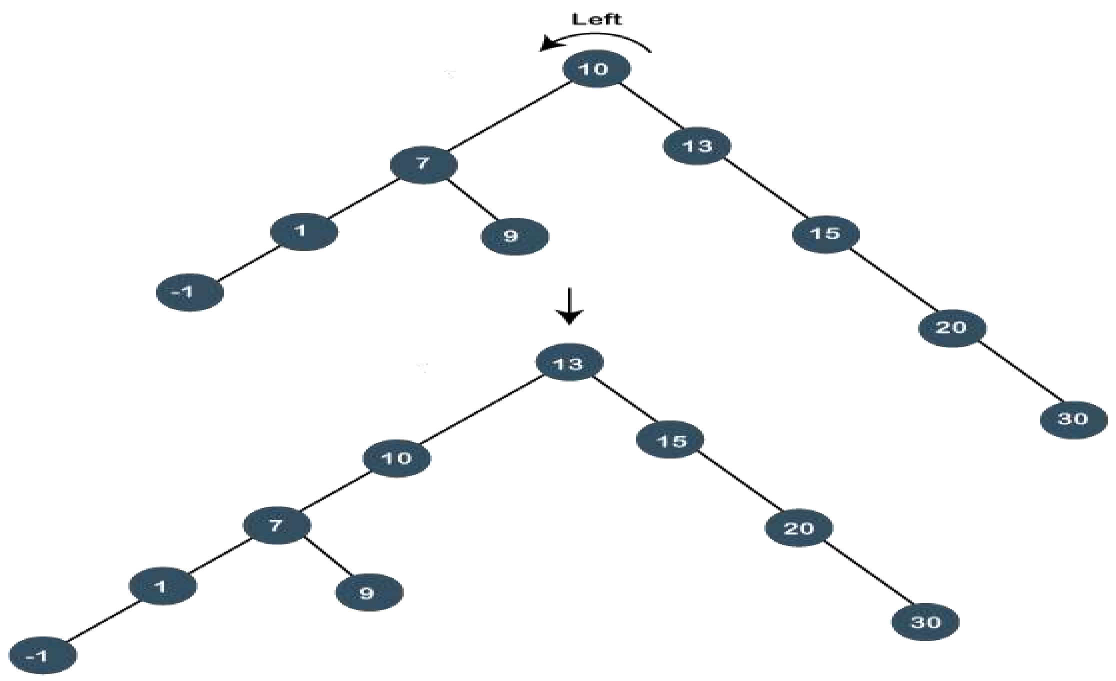


Step 1: First, we perform standard BST searching operation. As 13 is greater than 10 but less than 15, so node 13 will be the left child of node 15.

Step 2: Since node 13 is at the left of 15 and node 15 is at the right of node 10, so RL relationship exists. First, we perform the right rotation on node 15, and 15 will move downwards, and node 13 will come upwards, as shown below:



Still, node 13 is not the root node, and 13 is at the right of the root node, so we will perform left rotation known as a zag rotation. The node 10 will move downwards, and 13 becomes the root node as shown below:

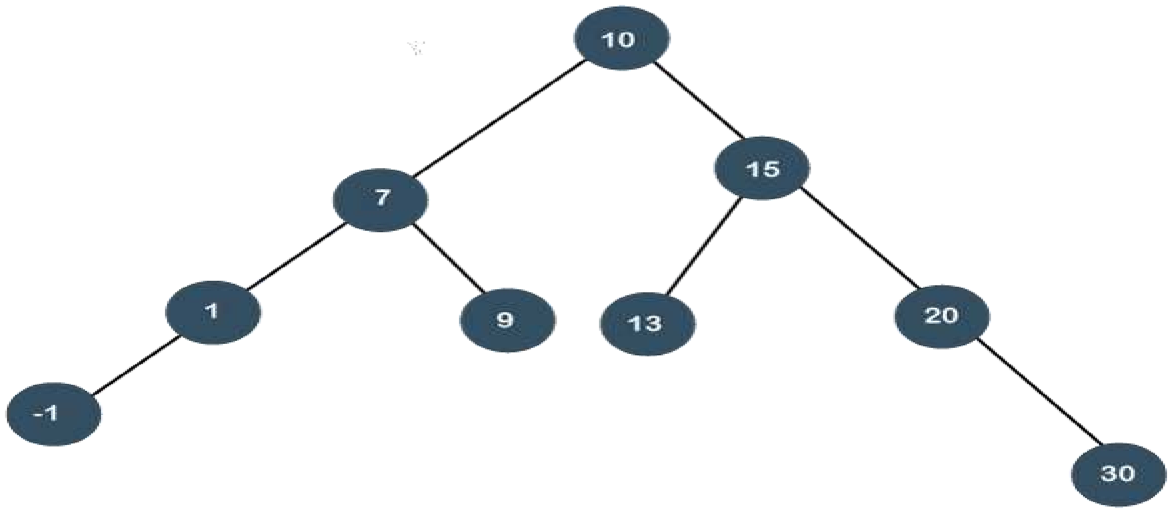


As we can observe in the above tree that node 13 has become the root node; therefore, the searching is completed. In this case, we have first performed the zig rotation and then zag rotation; so, it is known as a zig zag rotation.

- **Zag zig rotation**

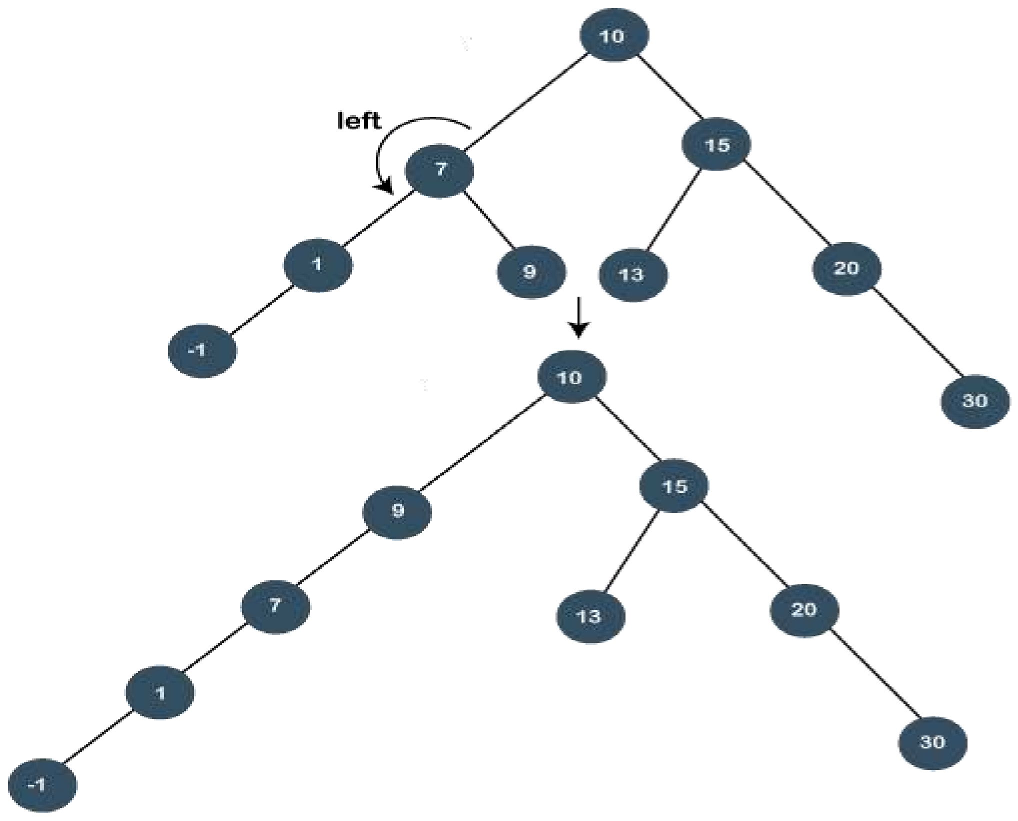
Let's understand this case through an example.

Suppose we want to search 9 element in the tree, which is shown below:

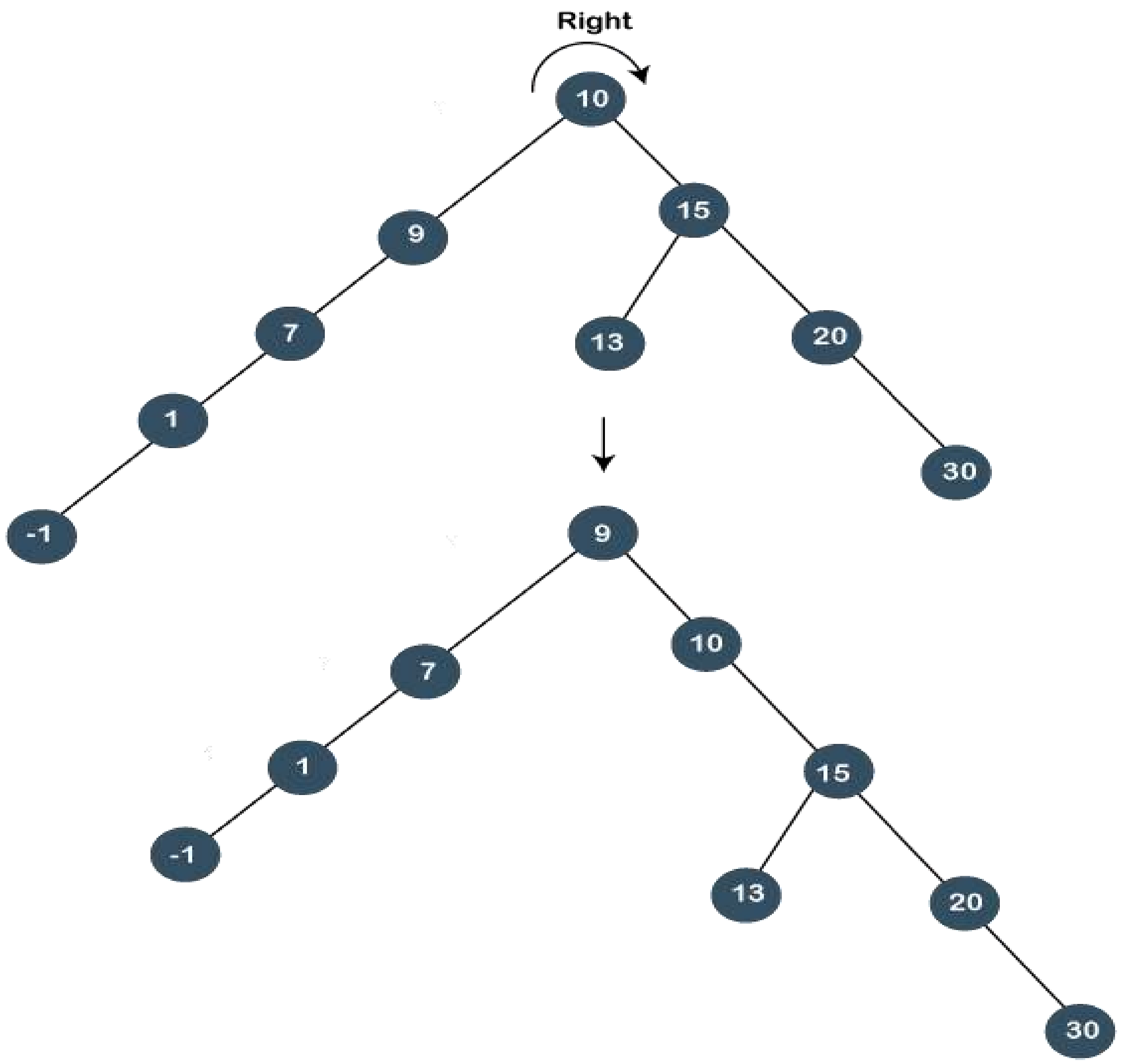


Step 1: First, we perform the standard BST searching operation. As 9 is less than 10 but greater than 7, so it will be the right child of node 7.

Step 2: Since node 9 is at the right of node 7, and node 7 is at the left of node 10, so LR relationship exists. First, we perform the left rotation on node 7. The node 7 will move downwards, and node 9 moves upwards as shown below:



Still the node 9 is not a root node, and 9 is at the left of the root node, so we will perform the right rotation known as zig rotation. After performing the right rotation, node 9 becomes the root node, as shown below:



As we can observe in the above tree that node 13 is a root node; therefore, the searching is completed. In this case, we have first performed the zag rotation (left rotation), and then zig rotation (right rotation) is performed, so it is known as a zag zig rotation.

Advantages of Splay tree

- In the splay tree, we do not need to store the extra information. In contrast, in AVL trees, we need to store the balance factor of each node that requires extra space, and Red-Black trees also require to store one extra bit of information that denotes the color of the node, either Red or Black.
- It is the fastest type of Binary Search tree for various practical applications. It is used in **Windows NT and GCC compilers**.
- It provides better performance as the frequently accessed nodes will move nearer to the root node, due to which the elements can be accessed quickly in splay trees. It is used in

the cache implementation as the recently accessed data is stored in the cache so that we do not need to go to the memory for accessing the data, and it takes less time.

Drawback of Splay tree

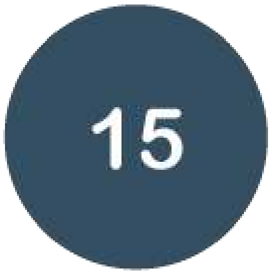
The major drawback of the splay tree would be that trees are not strictly balanced, i.e., they are roughly balanced. Sometimes the splay trees are linear, so it will take O(n) time complexity.

Insertion operation in Splay tree

In the ***insertion*** operation, we first insert the element in the tree and then perform the splaying operation on the inserted element.

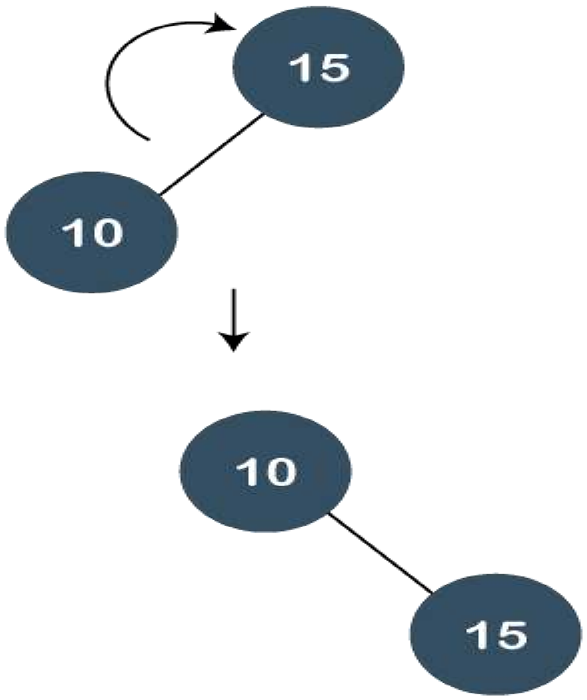
15, 10, 17, 7

Step 1: First, we insert node 15 in the tree. After insertion, we need to perform splaying. As 15 is a root node, so we do not need to perform splaying.



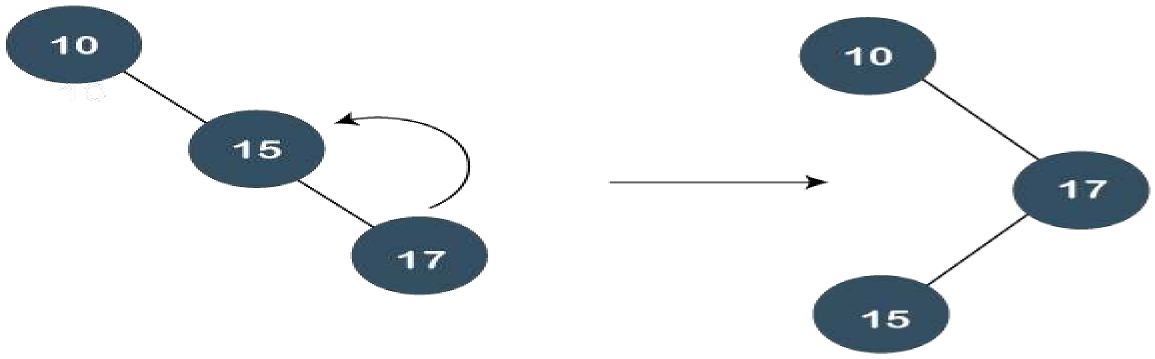
Step 2: The next element is 10. As 10 is less than 15, so node 10 will be the left child of node 15, as shown below:

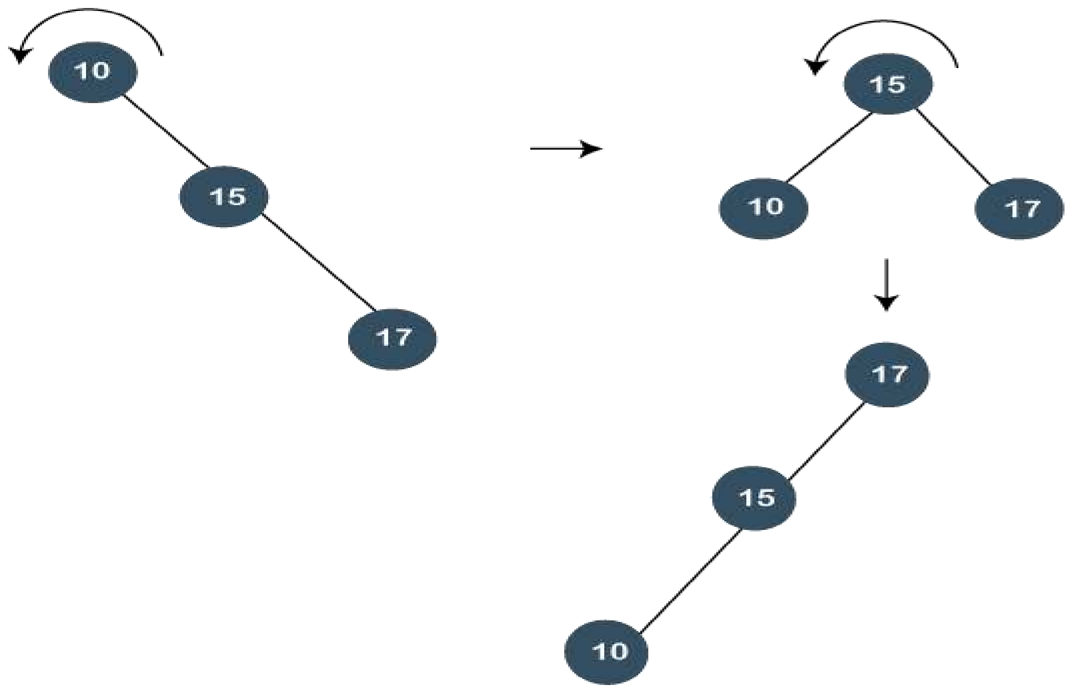
Now, we perform ***splaying***. To make 10 as a root node, we will perform the right rotation, as shown below:



Step 3: The next element is 17. As 17 is greater than 10 and 15 so it will become the right child of node 15.

Now, we will perform splaying. As 17 is having a parent as well as a grandparent so we will perform zig zig rotations

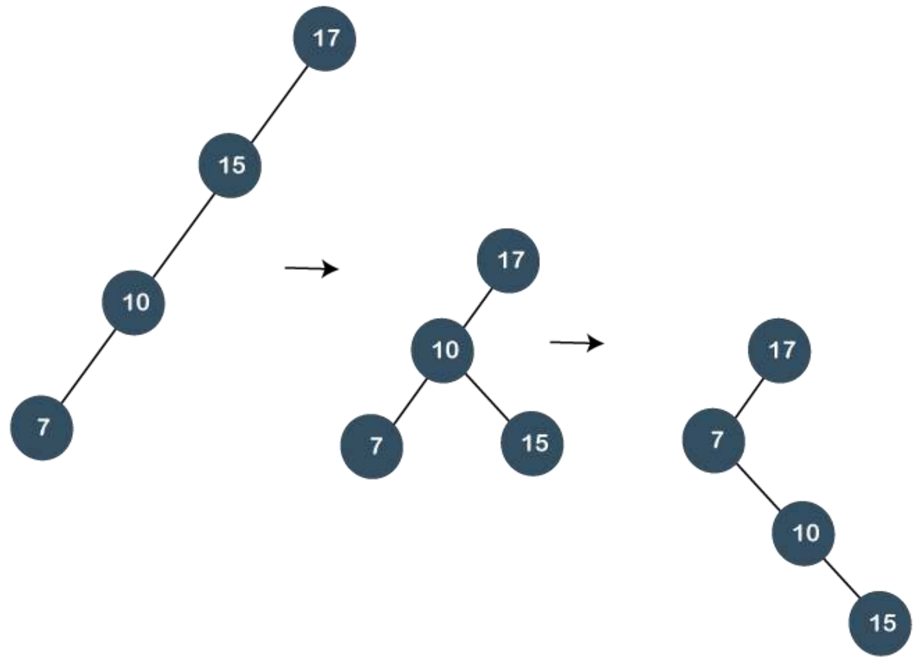




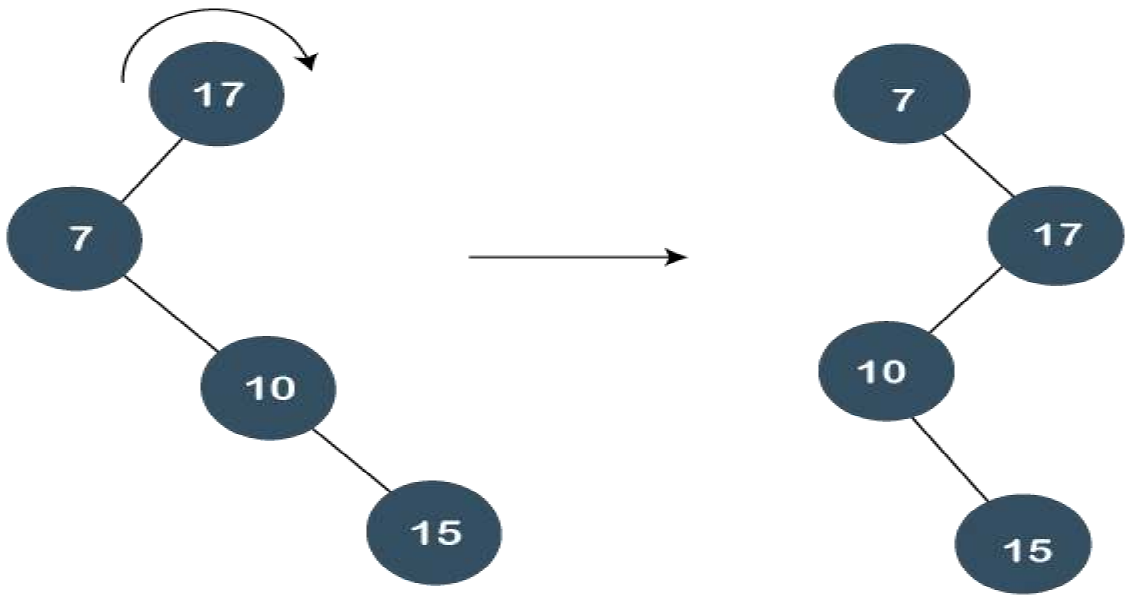
In the above figure, we can observe that 17 becomes the root node of the tree; therefore, the insertion is completed.

Step 4: The next element is 7. As 7 is less than 17, 15, and 10, so node 7 will be left child of 10.

Now, we have to splay the tree. As 7 is having a parent as well as a grandparent so we will perform two right rotations as shown below:



Still the node 7 is not a root node, it is a left child of the root node, i.e., 17. So, we need to perform one more right rotation to make node 7 as a root node as shown below:



Deletion in Splay tree

As we know that splay trees are the variants of the Binary search tree, so deletion operation in the splay tree would be similar to the BST, but the only difference is that the delete operation is followed in splay trees by the splaying operation.

Types of Deletions:

There are two types of deletions in the splay trees:

1. Bottom-up splaying
2. Top-down splaying

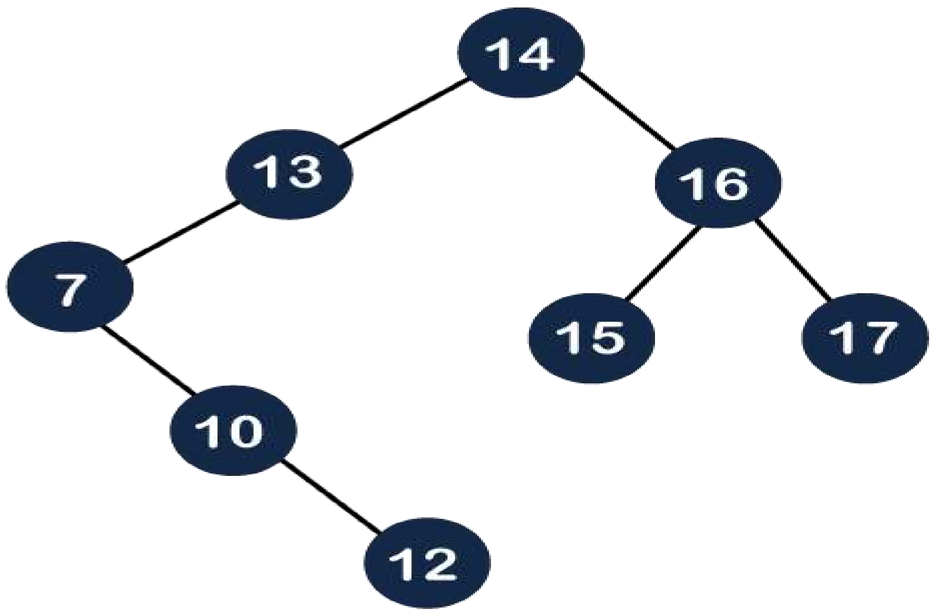
Bottom-up splaying

In bottom-up splaying, first we delete the element from the tree and then we perform the splaying on the deleted node.

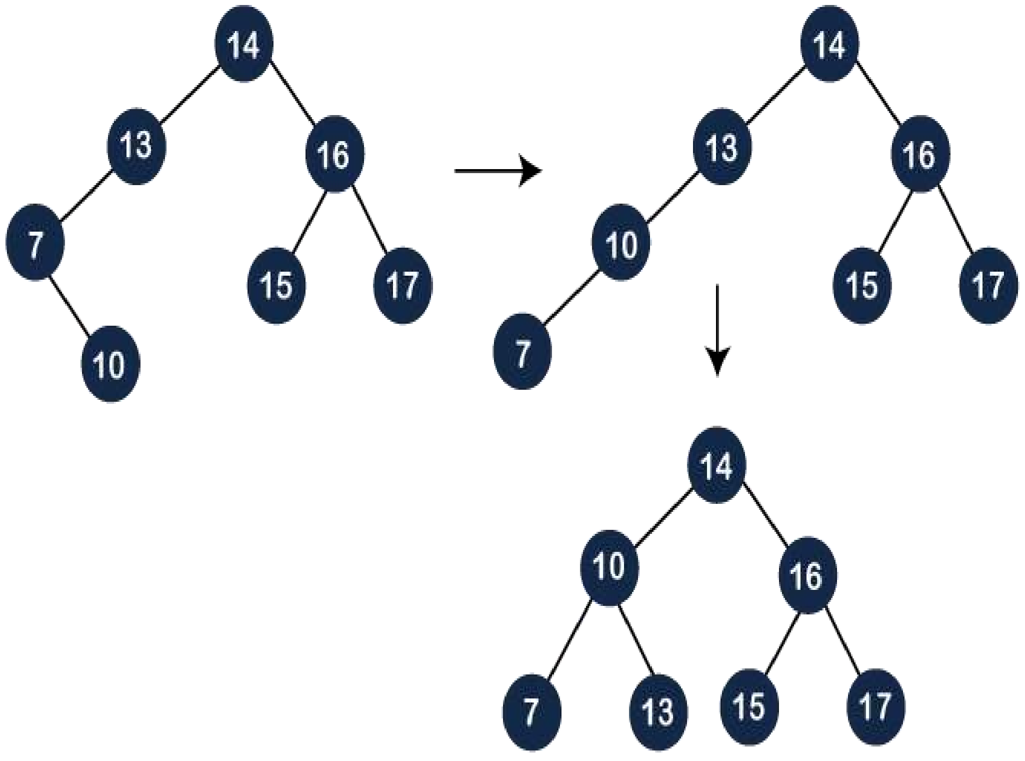
Let's understand the deletion in the Splay tree.

Suppose we want to delete 12, 14 from the tree shown below:

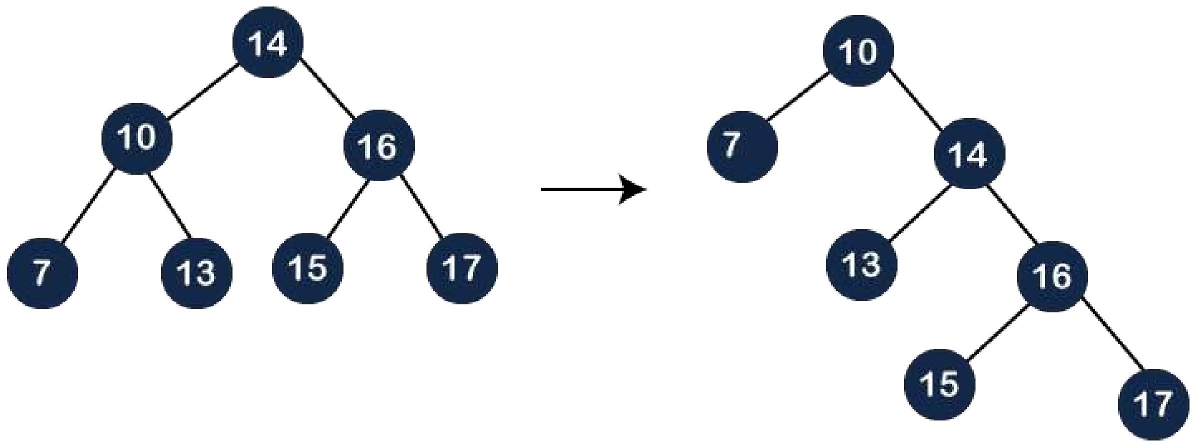
- First, we simply perform the standard BST deletion operation to delete 12 element. As 12 is a leaf node, so we simply delete the node from the tree.



The deletion is still not completed. We need to splay the parent of the deleted node, i.e., 10. We have to perform ***Splay(10)*** on the tree. As we can observe in the above tree that 10 is at the right of node 7, and node 7 is at the left of node 13. So, first, we perform the left rotation on node 7 and then we perform the right rotation on node 13, as shown below:

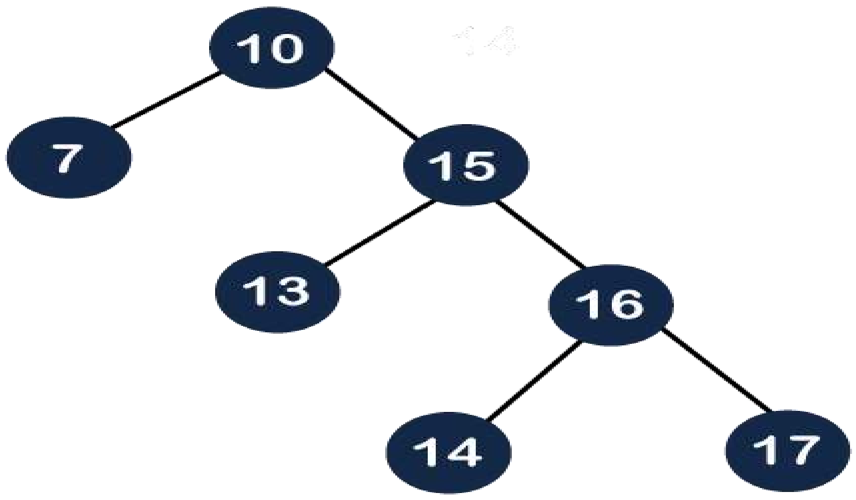


Still, node 10 is not a root node; node 10 is the left child of the root node. So, we need to perform the right rotation on the root node, i.e., 14 to make node 10 a root node as shown below:

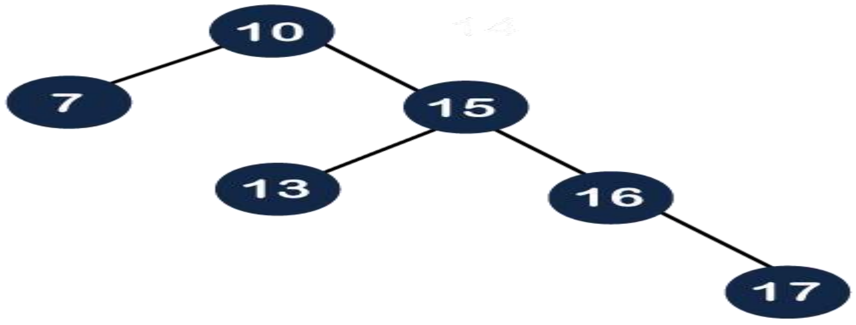


- Now, we have to delete the 14 element from the tree, which is shown below:

As we know that we cannot simply delete the internal node. We will replace the value of the node either using ***inorder predecessor*** or ***inorder successor***. Suppose we use inorder successor in which we replace the value with the lowest value that exist in the right subtree. The lowest value in the right subtree of node 14 is 15, so we replace the value 14 with 15. Since node 14 becomes the leaf node, so we can simply delete it as shown below:



Still, the deletion is not completed. We need to perform one more operation, i.e., splaying in which we need to make the parent of the deleted node as the root node. Before deletion, the parent of node 14 was the root node, i.e., 10, so we do need to perform any splaying in this case.

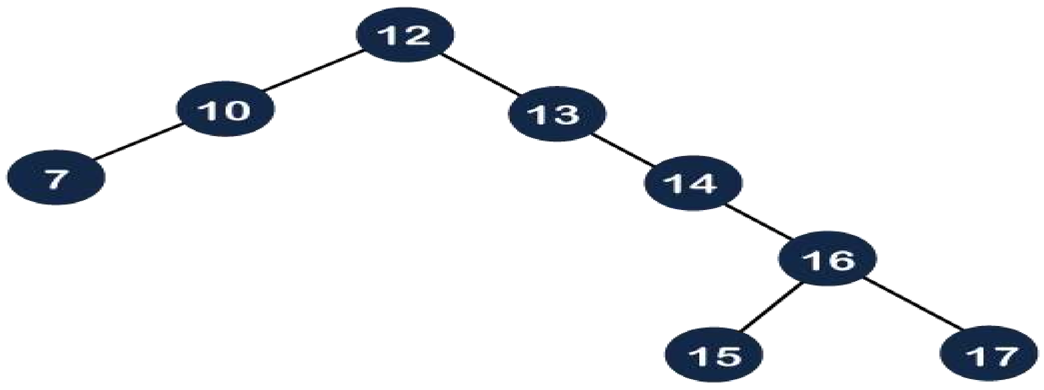


Top-down splaying

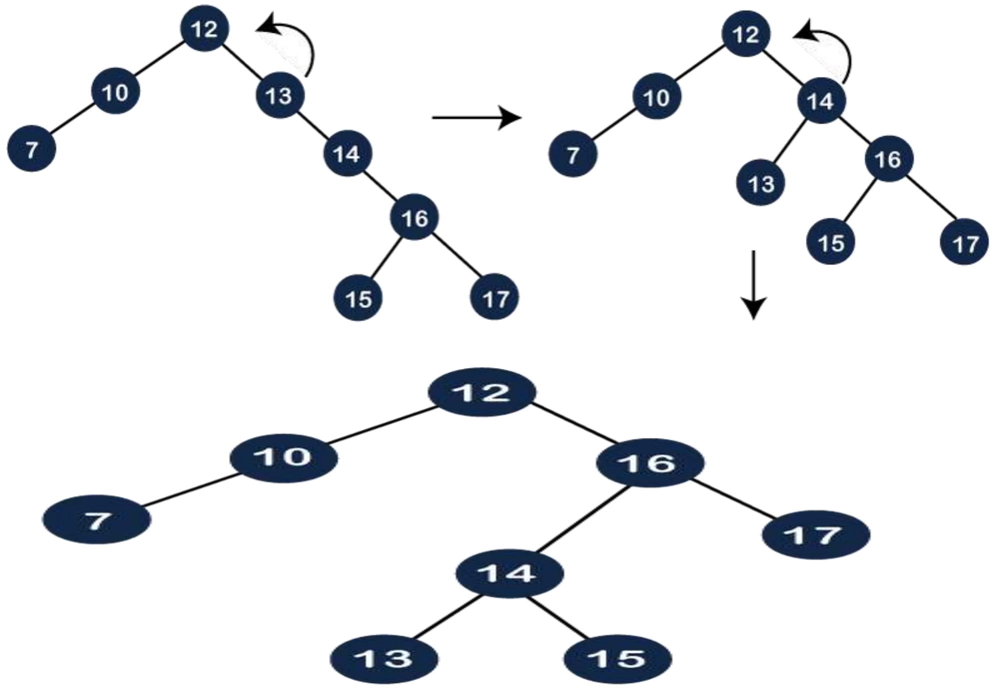
In top-down splaying, we first perform the splaying on which the deletion is to be performed and then delete the node from the tree. Once the element is deleted, we will perform the join operation.

Let's understand the top-down splaying through an example.

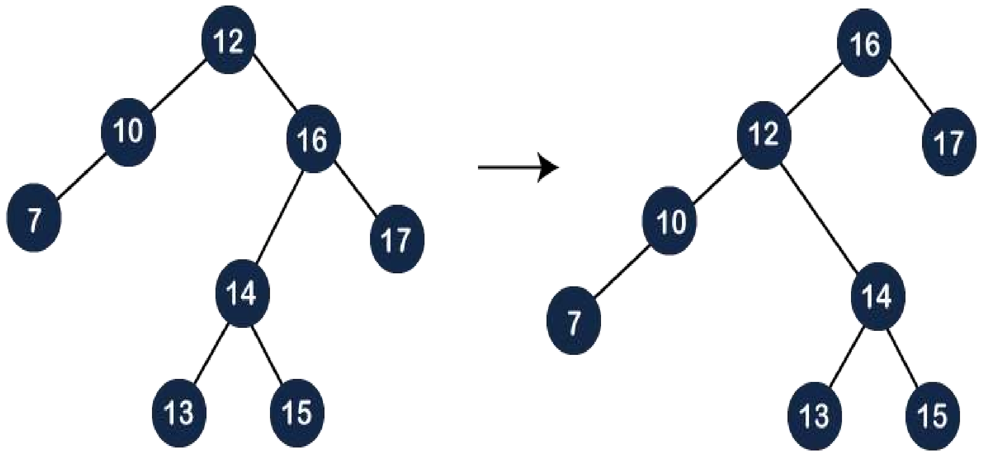
Suppose we want to delete 16 from the tree which is shown below:



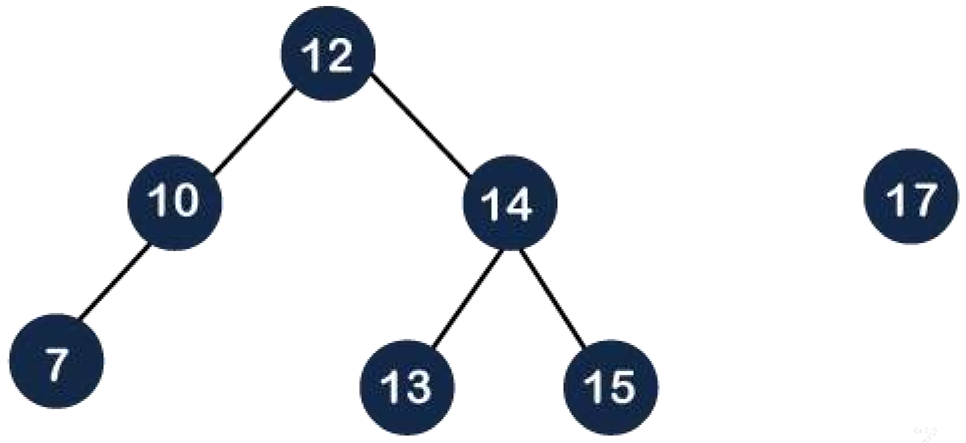
Step 1: In top-down splaying, first we perform splaying on the node 16. The node 16 has both parent as well as grandparent. The node 16 is at the right of its parent and the parent node is also at the right of its parent, so this is a zag zag situation. In this case, first, we will perform the left rotation on node 13 and then 14 as shown below:



The node 16 is still not a root node, and it is a right child of the root node, so we need to perform left rotation on the node 12 to make node 16 as a root node.

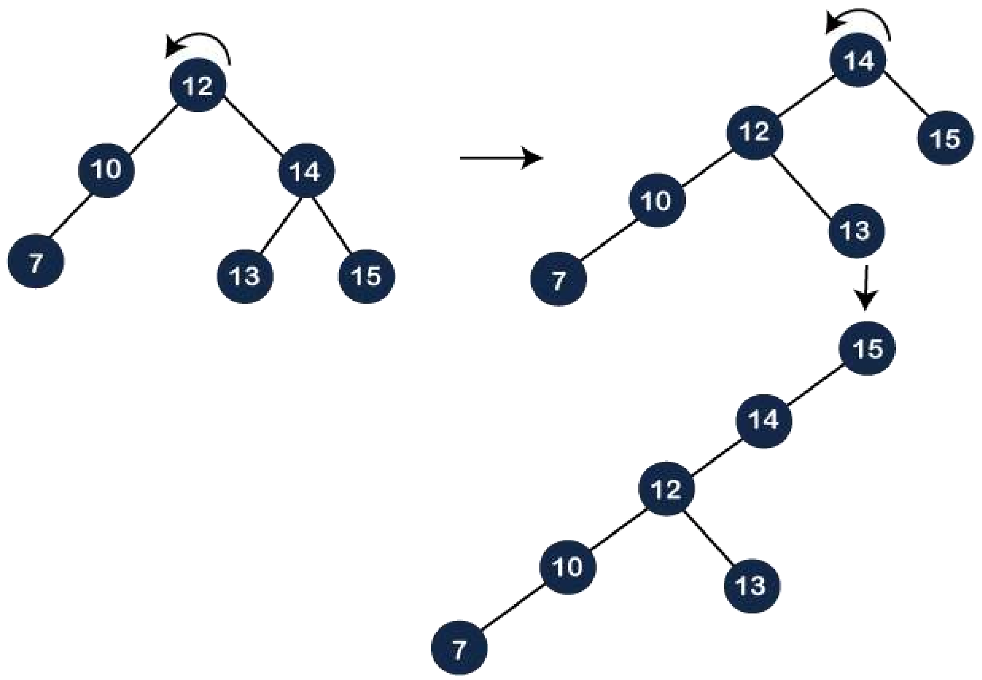


Once the node 16 becomes a root node, we will delete the node 16 and we will get two different trees, i.e., left subtree and right subtree as shown below:

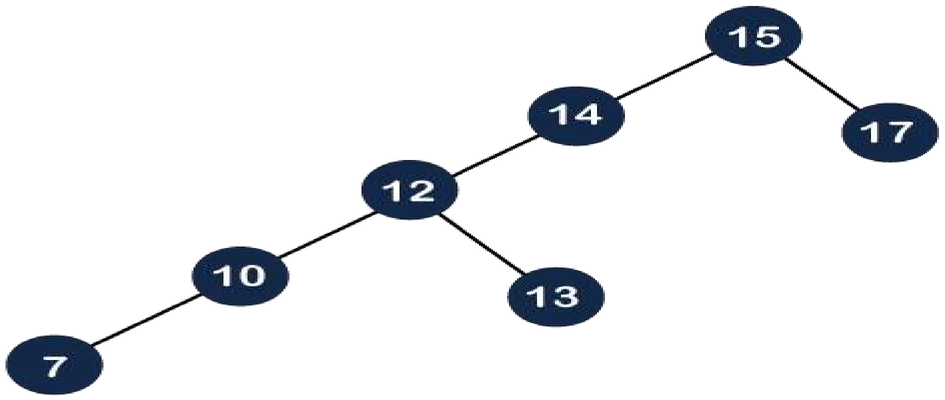


As we know that the values of the left subtree are always lesser than the values of the right subtree. The root of the left subtree is 12 and the root of the right subtree is 17. The first step is to find the maximum element in the left subtree. In the left subtree, the maximum element is 15, and then we need to perform splaying operation on 15.

As we can observe in the above tree that the element 15 is having a parent as well as a grandparent. A node is right of its parent, and the parent node is also right of its parent, so we need to perform two left rotations to make node 15 a root node as shown below:



After performing two rotations on the tree, node 15 becomes the root node. As we can see, the right child of the 15 is NULL, so we attach node 17 at the right part of the 15 as shown below, and this operation is known as a ***join*** operation.


15
14
17
12
10
13
7