Advantages of Threaded Binary Tree

- In this Tree it enables linear traversal of elements.
- It eliminates the use of stack as it perform linear traversal.
- Enables to find parent node without explicit use of parent pointer
- Threaded tree give forward and backward traversal of nodes by in-order fashion
- Nodes contain pointers to in-order predecessor and successor

Binary Search Tree:

In this article, we will discuss the Binary search tree. This article will be very helpful and informative to the students with technical background as it is an important topic of their course. Before moving directly to the binary search tree, let's first see a brief description of the tree.

What is a tree?

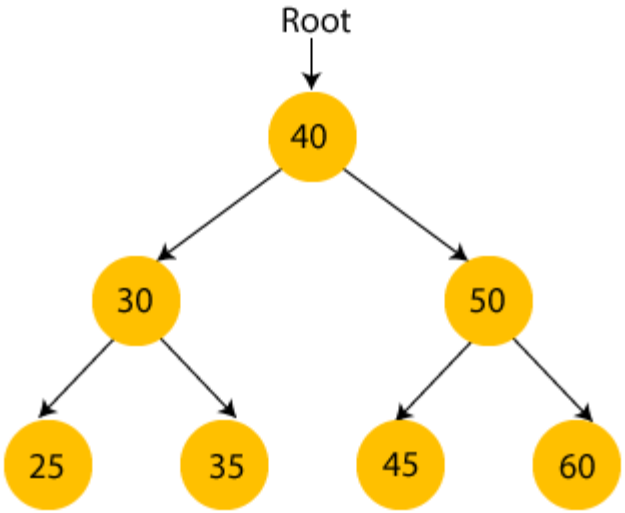
A tree is a kind of data structure that is used to represent the data in hierarchical form. It can be defined as a collection of objects or entities called as nodes that are linked together to simulate a hierarchy. Tree is a non-linear data structure as the data in a tree is not stored linearly or sequentially.

Now, let's start the topic, the Binary Search tree.

What is a Binary Search tree?

A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.

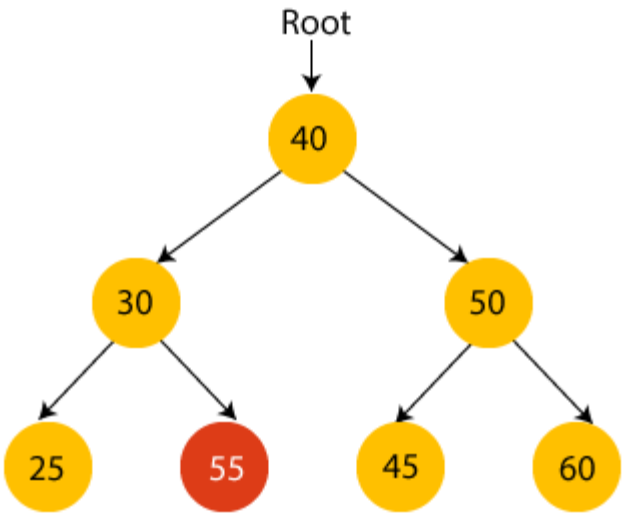
Let's understand the concept of Binary search tree with an example.



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

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

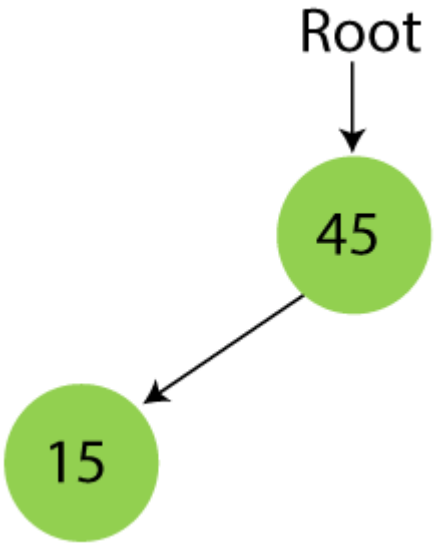
Suppose the data elements are **- 45, 15, 79, 90, 10, 55, 12, 20, 50**

- 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.

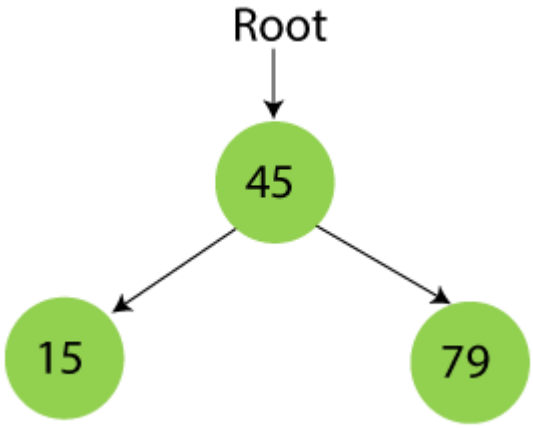
Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below -

Step 1 - Insert 45.

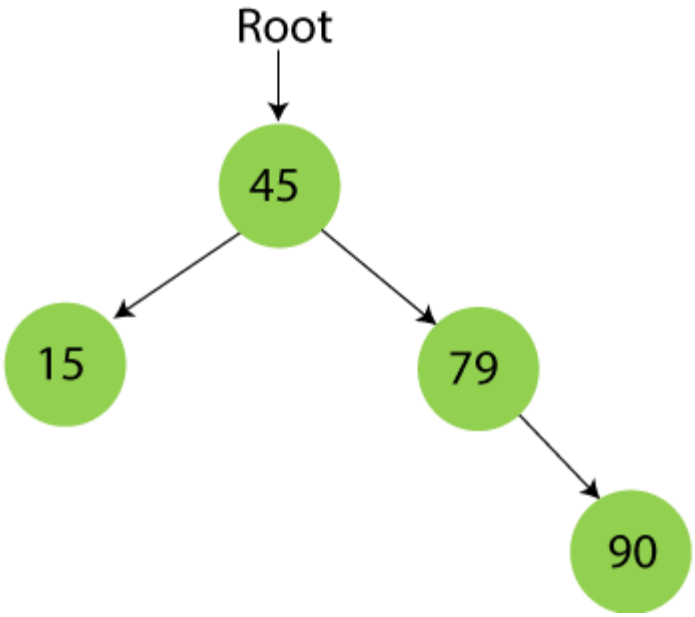
Step 2 - Insert 15.
As 15 is smaller than 45, so insert it as the root node of the left subtree.



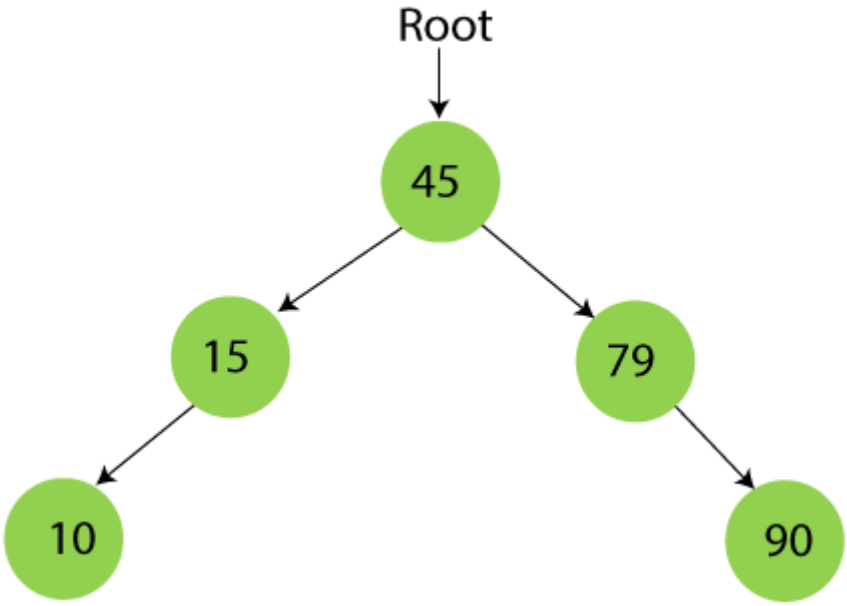
Step 3 - Insert 79.
As 79 is greater than 45, so insert it as the root node of the right subtree.



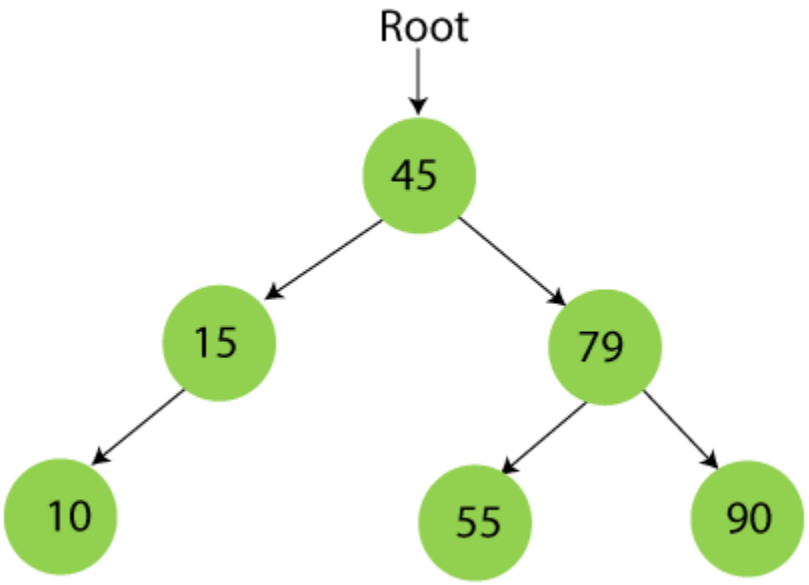
Step 4 - Insert 90.
90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



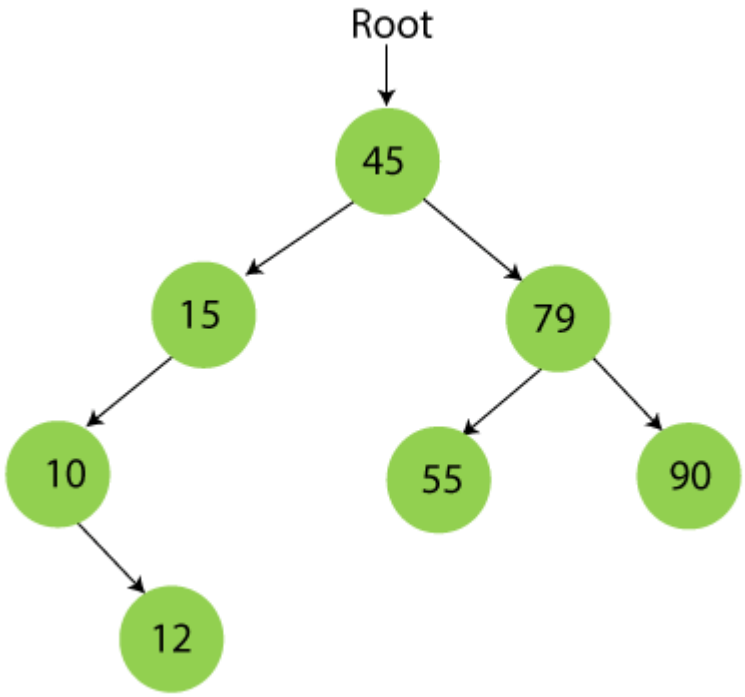
Step 5 - Insert 10.
10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



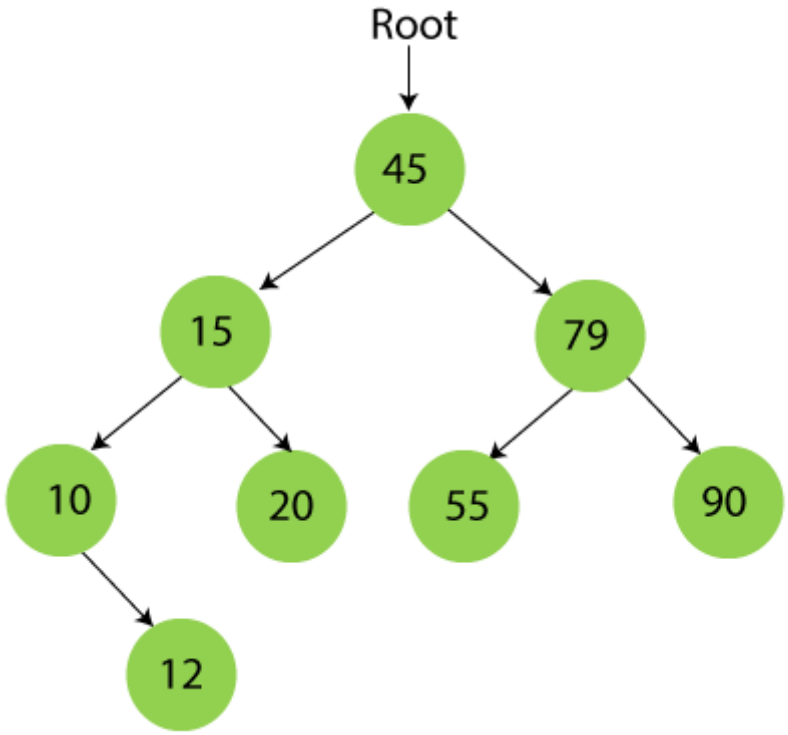
Step 6 - Insert 55.
55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



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.

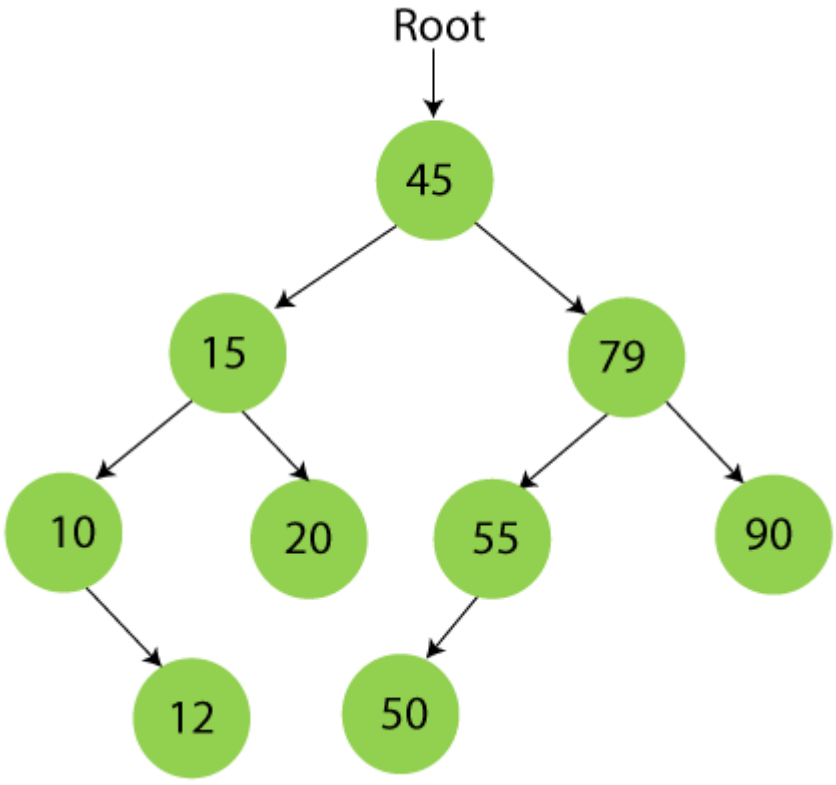


Step 8 - Insert 20.
20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



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.

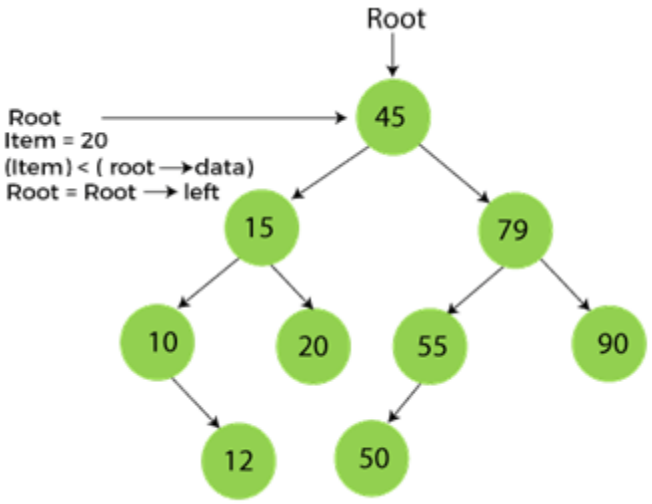
Searching in Binary search tree

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 -

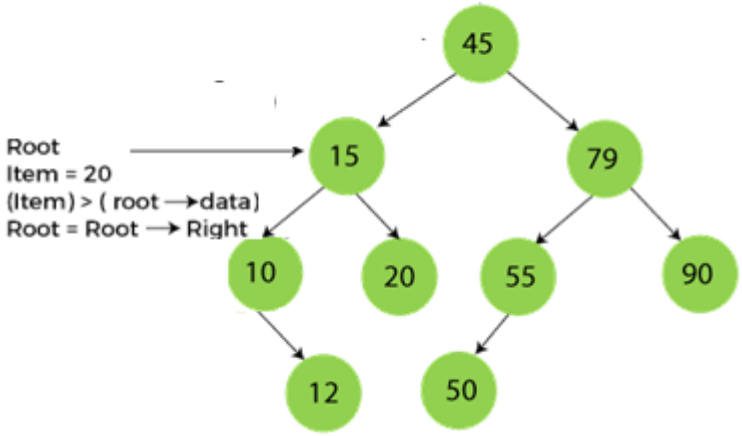
1. First, compare the element to be searched with the root element of the tree.
2. If root is matched with the target element, then return the node's location.
3. If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the 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:


Root
Root
Item = 20
(Item) < (root →data)
Root = Root → left
45
15
79
10
20
55
90
12
50

Step2:


45
Root
Item = 20
(Item) > (root →data)
Root = Root → Right
15
79
10
20
55
90
12
50

Step3:


45
15
79
Root
Item = 20
(Item) = (root →data)
return Root
10
20
55
90
12
50

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

Algorithm to search an element in Binary search tree

1. Search (root, item)
2. Step 1 - if (item = root → data) or (root = NULL)
3. return root
4. else if (item < **root** → data)
5. return Search(root → left, item)
6. else
7. return Search(root → right, item)
8. END if
9. Step 2 - END

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

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

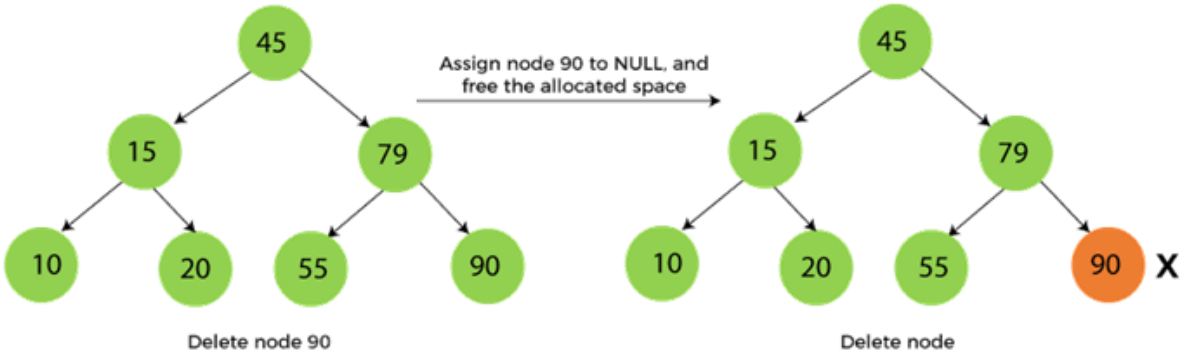
We will understand the situations listed above in detail.

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.

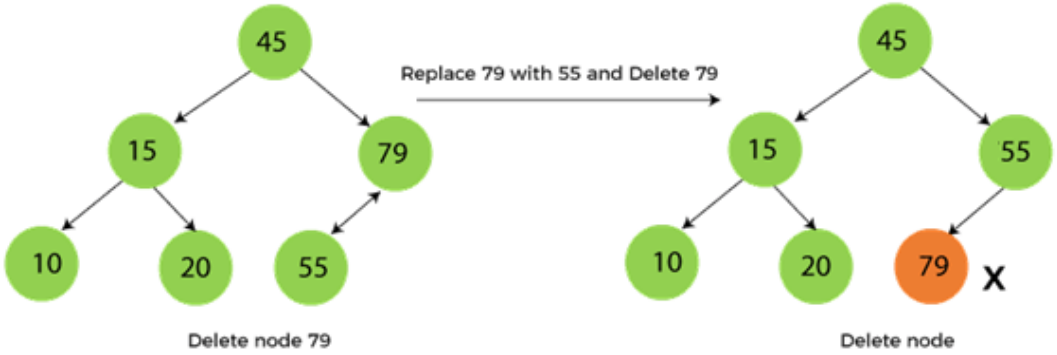


When the node to be deleted has only one child

In this case, we have to replace the target node 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.

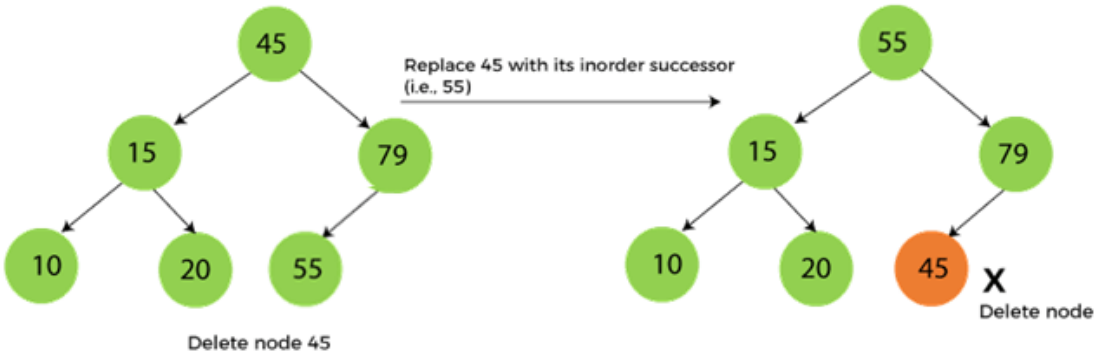


When the node to be deleted has two children

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.



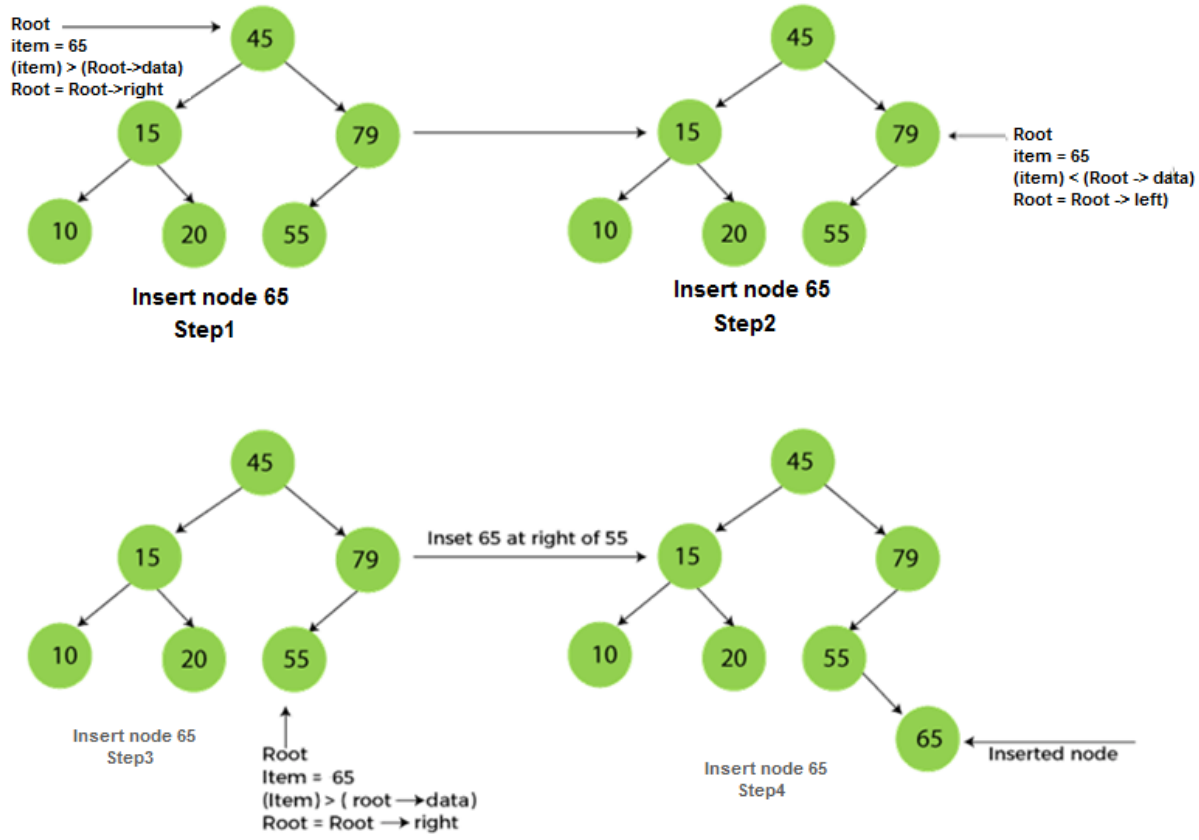
Now let's understand how insertion is performed on a binary search tree.

Insertion in Binary Search tree

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)

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
Now, let's see the program to implement the operations of Binary Search tree.
Program: Write a program to perform operations of Binary Search tree in C++.
In this program, we will see the implementation of the operations of binary search tree. Here, we will see the creation, inorder traversal, insertion, and deletion operations of tree.
Here, we will see the inorder traversal of the tree to check whether the nodes of the tree are in their proper location or not. We know that the inorder traversal always gives us the data in ascending order. So, after performing the insertion and deletion operations, we perform the inorder traversal, and after traversing, if we get data in ascending order, then it is clear that the nodes are in their proper location.

```
1. #include <iostream>
2. using namespace std;
3. struct Node {
4.     int data;
5.     Node *left;
6.     Node *right;
7. };
8. Node* create(int item)
9. {
10.         Node* node = new Node;
11.         node->data = item;
```

```
12.         node->left = node->right = NULL;
13.         return node;
14.     }
15.     /*Inorder traversal of the tree formed*/
16.     void inorder(Node *root)
17.     {
18.         if (root == NULL)
19.             return;
20.         inorder(root->left); //traverse left subtree
21.         cout<< root->data << "   "; //traverse root node
22.         inorder(root->right); //traverse right subtree
23.     }
24.     Node* findMinimum(Node* cur) /*To find the
    inorder successor*/
25.     {
26.         while(cur->left != NULL) {
27.             cur = cur->left;
28.         }
29.         return cur;
30.     }
31.     Node* insertion(Node* root, int item) /*Insert a
    node*/
32.     {
33.         if (root == NULL)
34.             return create(item); /*return new node if tree is
    empty*/
35.         if (item < root->data)
36.             root->left = insertion(root->left, item);
37.         else
38.             root->right = insertion(root->right, item);
39.         return root;
40.     }
41.     void search(Node* &cur, int item, Node* &parent)
```

```
42.     {
43.         while (cur != NULL && cur->data != item)
44.         {
45.             parent = cur;
46.             if (item < cur->data)
47.                 cur = cur->left;
48.             else
49.                 cur = cur->right;
50.         }
51.     }
52.     void deletion(Node*& root, int item) /*function to
    delete a node*/
53.     {
54.         Node* parent = NULL;
55.         Node* cur = root;
56.         search(cur, item, parent); /*find the node to be
    deleted*/
57.         if (cur == NULL)
58.             return;
59.         if (cur->left == NULL && cur->right == NULL)
    /*When node has no children*/
60.         {
61.             if (cur != root)
62.             {
63.                 if (parent->left == cur)
64.                     parent->left = NULL;
65.                 else
66.                     parent->right = NULL;
67.             }
68.             else
69.                 root = NULL;
70.             free(cur);
71.         }
```

```
72.         else if (cur->left && cur->right)
73.         {
74.             Node* succ  = findMinimum(cur->right);
75.             int val = succ->data;
76.             deletion(root, succ->data);
77.             cur->data = val;
78.         }
79.         else
80.         {
81.             Node* child = (cur->left)? cur->left: cur->right;
82.             if (cur != root)
83.             {
84.                 if (cur == parent->left)
85.                     parent->left = child;
86.                 else
87.                     parent->right = child;
88.             }
89.             else
90.                 root = child;
91.             free(cur);
92.         }
93.     }
94.     int main()
95.     {
96.       Node* root = NULL;
97.       root = insertion(root, 45);
98.       root = insertion(root, 30);
99.       root = insertion(root, 50);
100.      root = insertion(root, 25);
101.      root = insertion(root, 35);
102.      root = insertion(root, 45);
103.      root = insertion(root, 60);
```

```
104.    root = insertion(root, 4);
105.    printf("The inorder traversal of the given binary tree
        is - \n");
106.    inorder(root);
107.    deletion(root, 25);
108.    printf("\nAfter deleting node 25, the inorder
        traversal of the given binary tree is - \n");
109.    inorder(root);
110.    insertion(root, 2);
111.    printf("\nAfter inserting node 2, the inorder
        traversal of the given binary tree is - \n");
112.    inorder(root);
113.    return 0;
114.  }
```

Output

After the execution of the above code, the output will be -

```
The inorder traversal of the given binary tree is -
4       25      30      35      45      45      50      60
After deleting node 25, the inorder traversal of the given binary tree is -
4       30      35      45      45      50      60
After inserting node 2, the inorder traversal of the given binary tree is -
2       4       30      35      45      45      50      60
```

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.

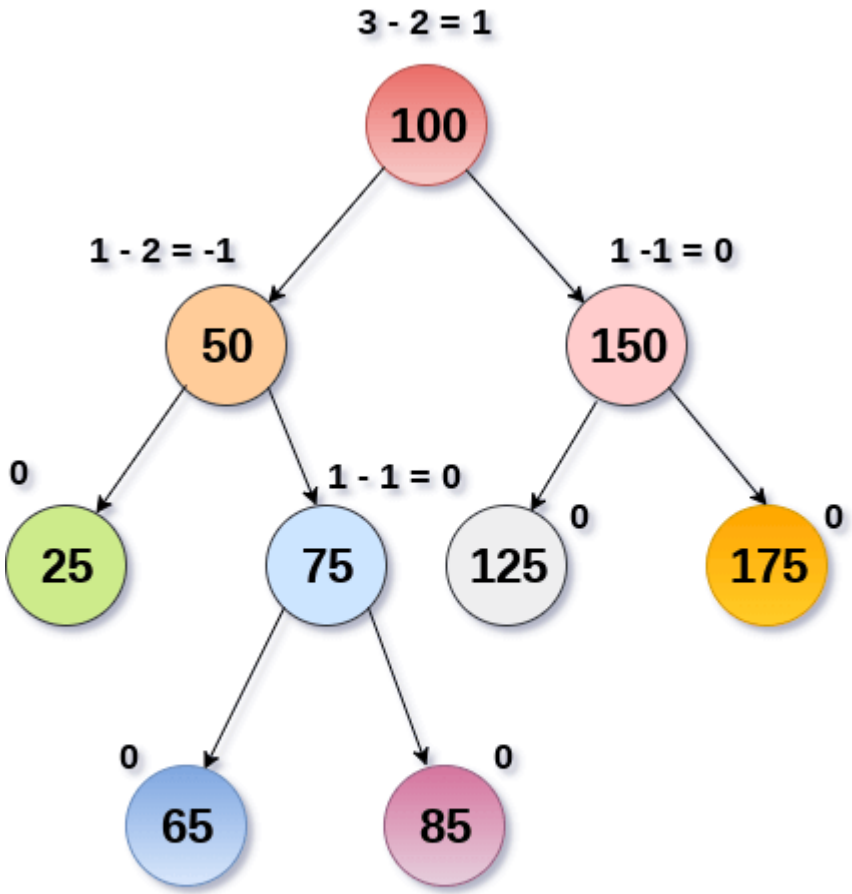
Balance Factor (k) = height (left(k)) - height (right(k))

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

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

Complexity

Algorithm	Average case	Worst case
Space	o(n)	o(n)
Search	o(log n)	o(log n)
Insert	o(log n)	o(log n)
Delete	o(log n)	o(log n)

Operations on AVL tree

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

SN	Operation	Description

1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. 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.
2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the 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:

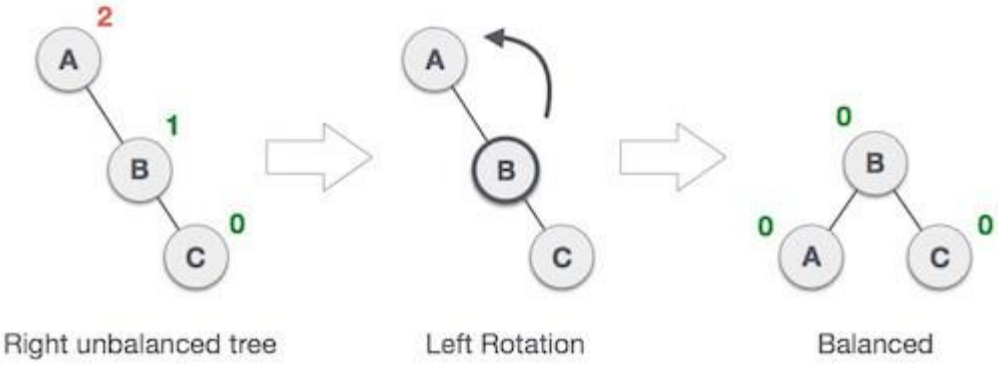
1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation : Inserted node is in the right subtree of right subtree of A
3. L R rotation : Inserted node is in the right subtree of left subtree of A
4. R L rotation : Inserted node is in the left subtree of right subtree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

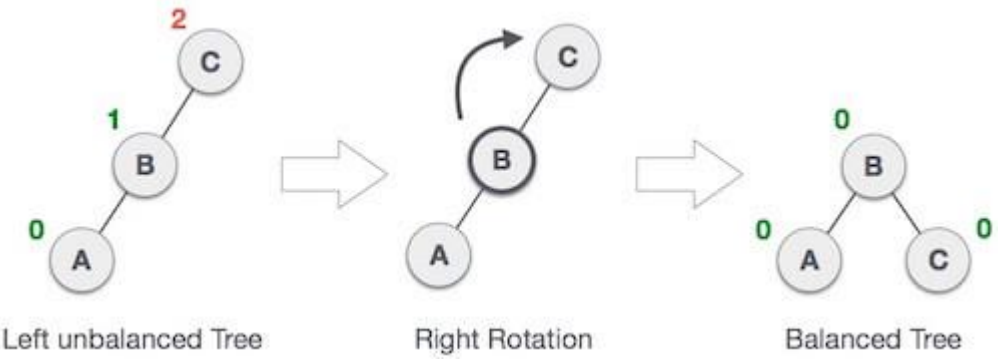
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



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.

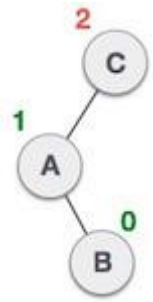
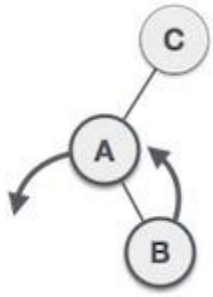


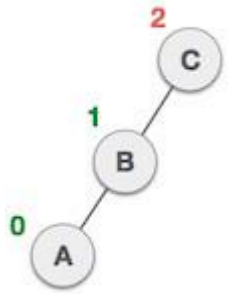
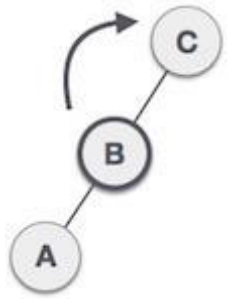
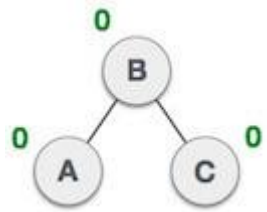
In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

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.

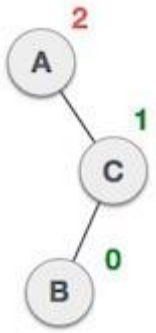
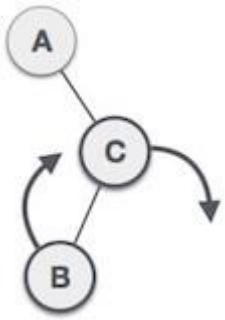
Let us understand each and every step very clearly:

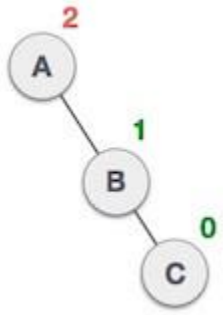
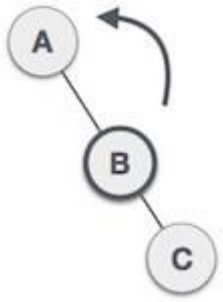
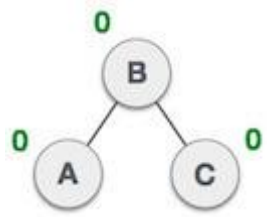
State	Action
	A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C
	As LR rotation = RR + LL rotation, hence RR (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.

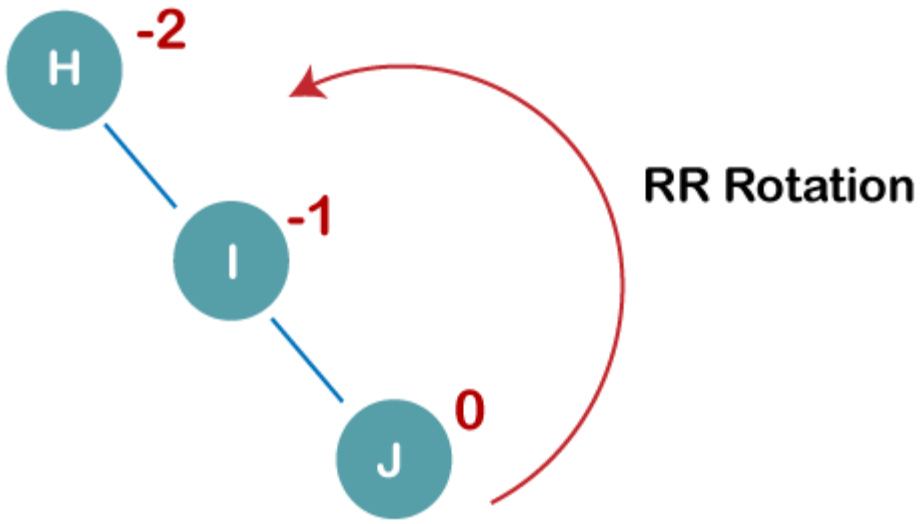
4. RL Rotation

As already discussed, that double rotations are bit tougher than single rotation which has already explained above. 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**.

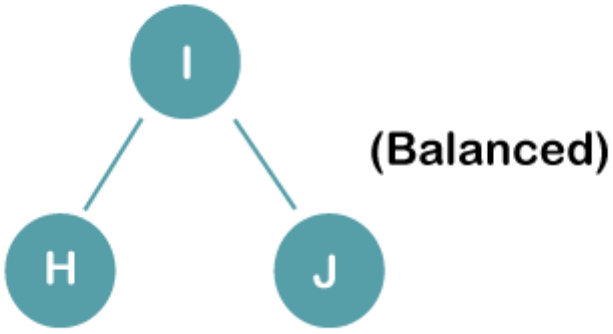
	After performing LL rotation, node **A** is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.
	Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node **C** has now become the right subtree of node B, and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

Q: 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

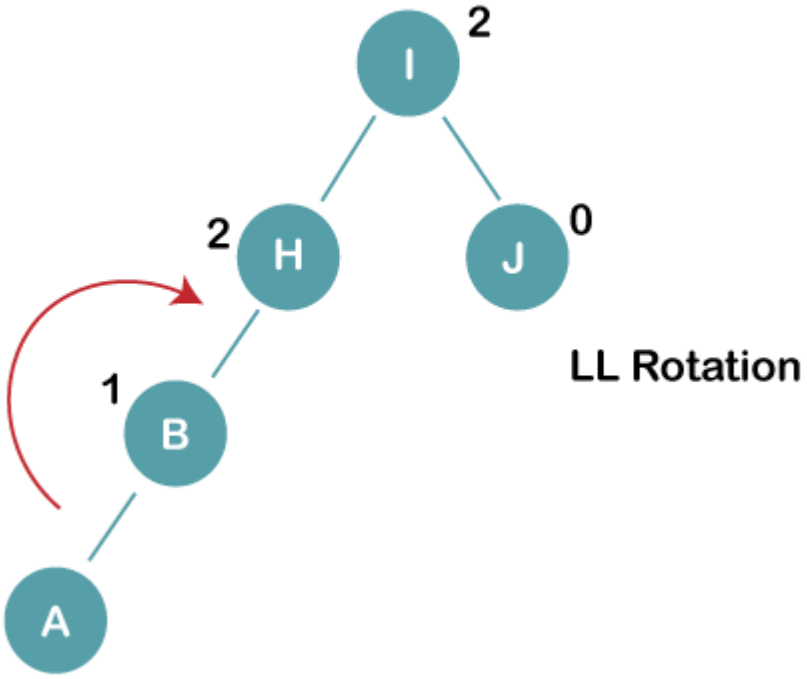


On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2. Since the BST is right-skewed, we will perform RR Rotation on node H.

The resultant balance tree is:

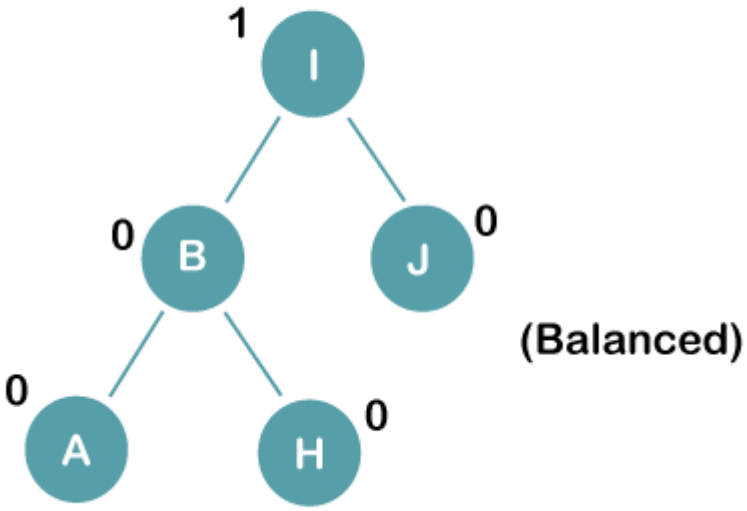


2. Insert B, A

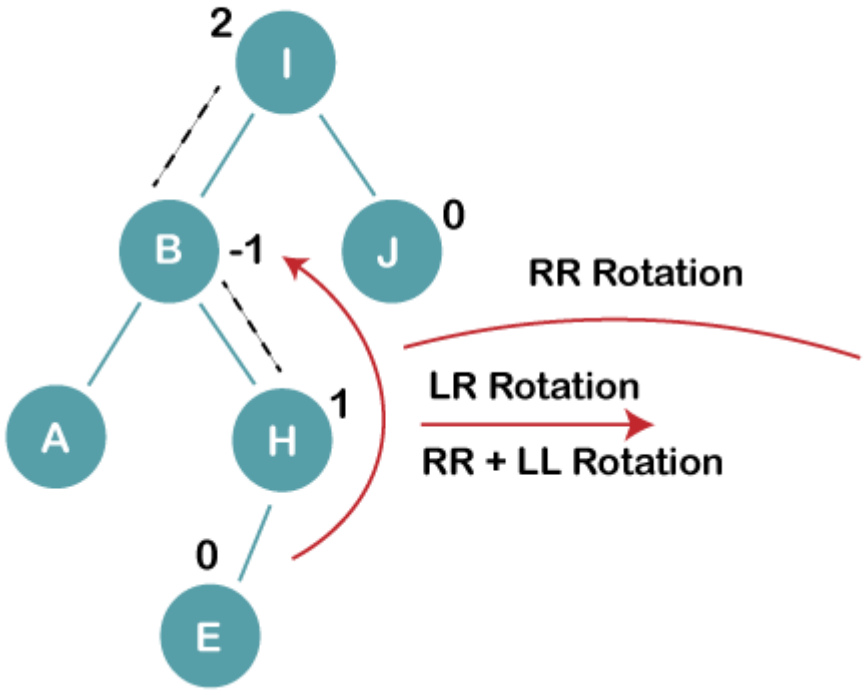


On inserting the above elements, especially in case of A, the BST becomes unbalanced as the Balance Factor of H and I is 2, we consider the first node from the last inserted node i.e. H. Since the BST from H is left-skewed, we will perform LL Rotation on node H.

The resultant balance tree is:



3. Insert E

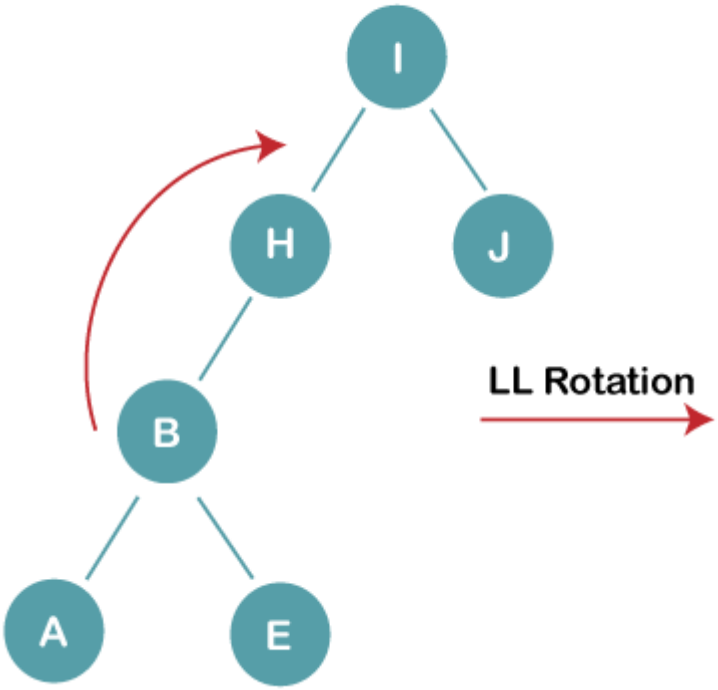


On inserting E, BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I we find that it is inserted

in the left subtree of right subtree of I, we will perform LR Rotation on node I. LR = RR + LL rotation

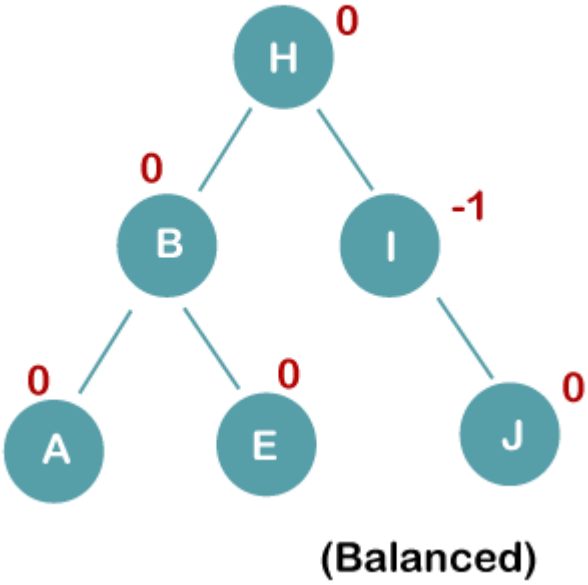
3 a) We first perform RR rotation on node B

The resultant tree after RR rotation is:



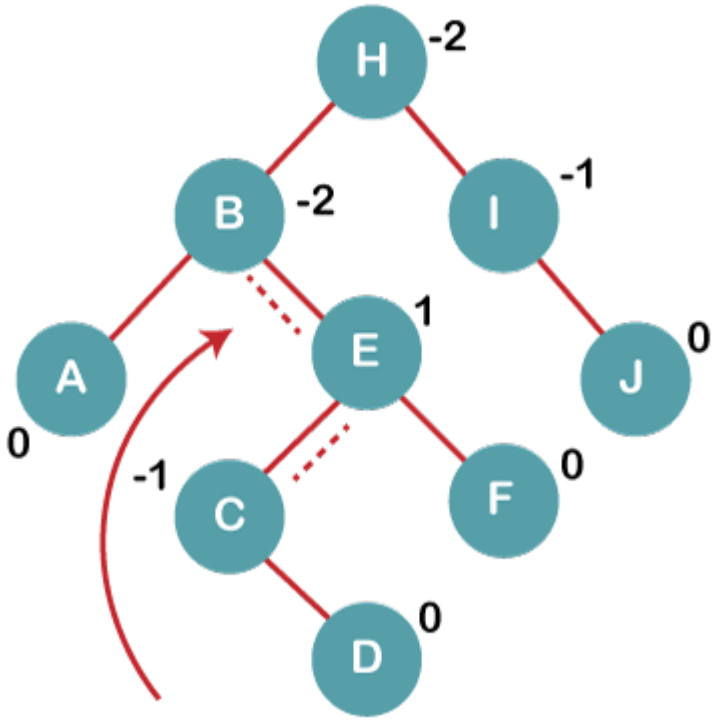
3b) We first perform LL rotation on the node I

The resultant balanced tree after LL rotation is:



(Balanced)

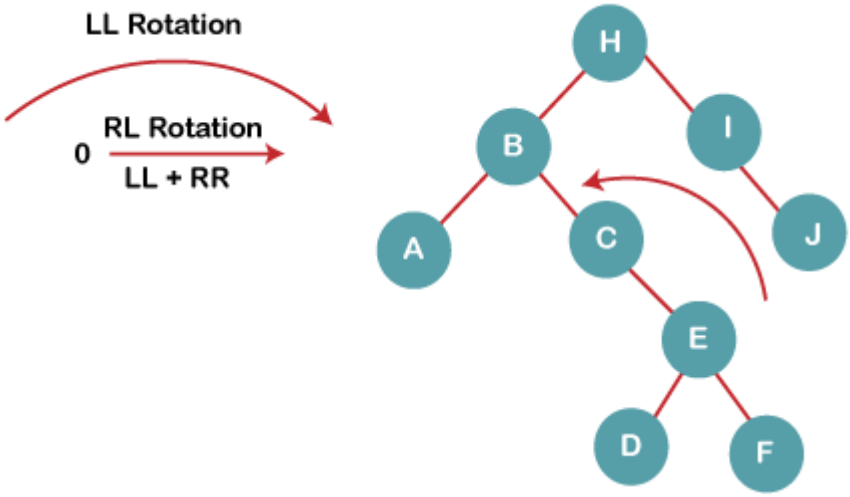
4. Insert C, F, D



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.

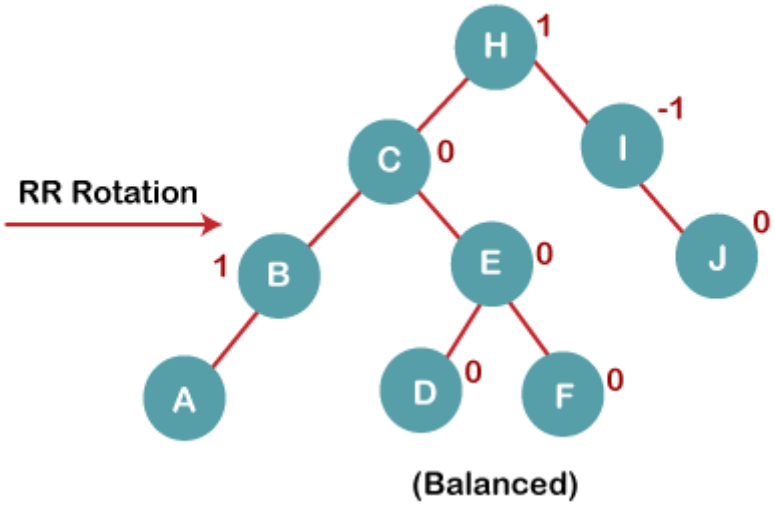
4a) We first perform LL rotation on node E

The resultant tree after LL rotation is:

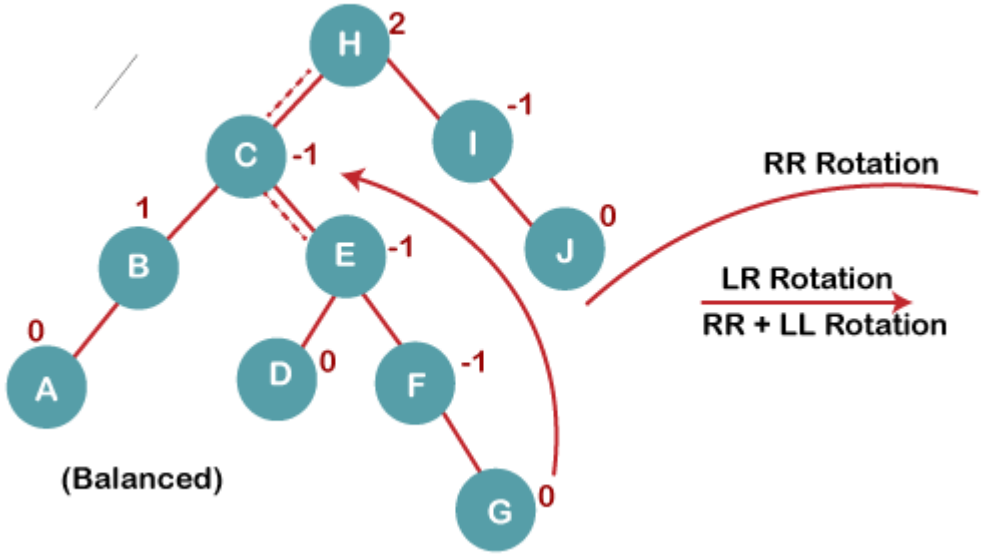


4b) We then perform RR rotation on node B

The resultant balanced tree after RR rotation is:



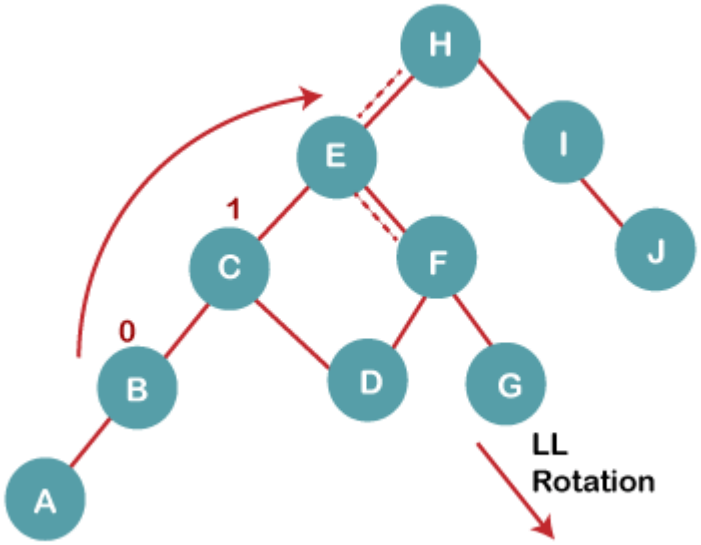
5. Insert G



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.

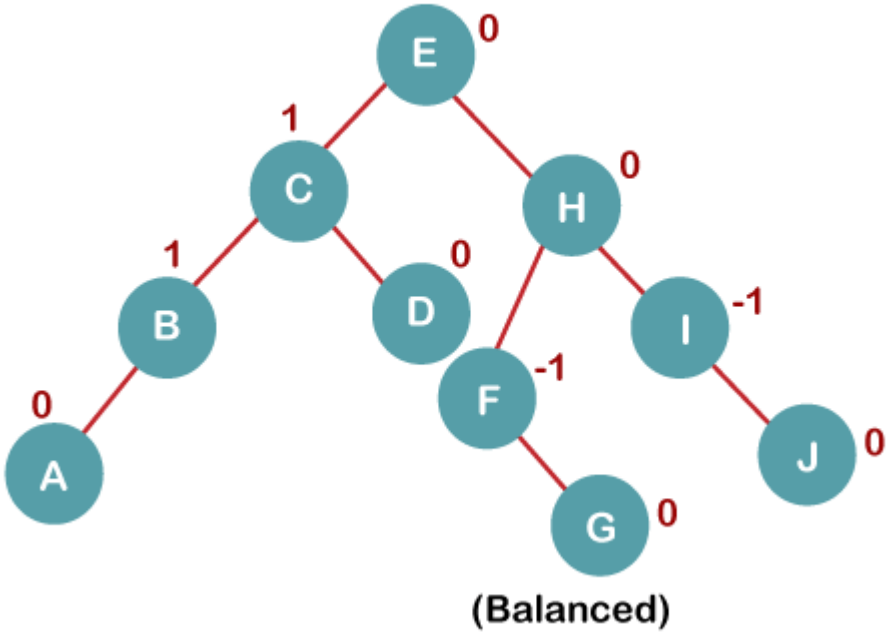
5 a) We first perform RR rotation on node C

The resultant tree after RR rotation is:



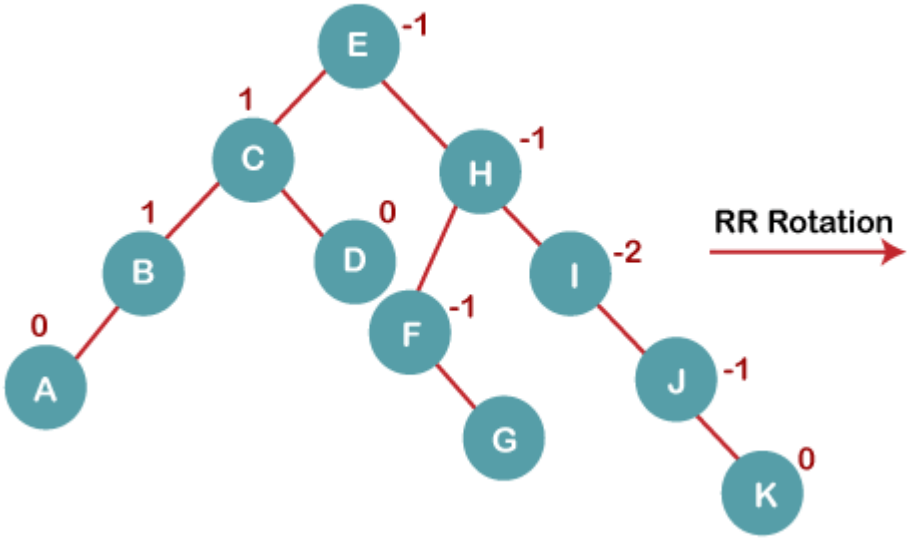
5 b) We then perform LL rotation on node H

The resultant balanced tree after LL rotation is:



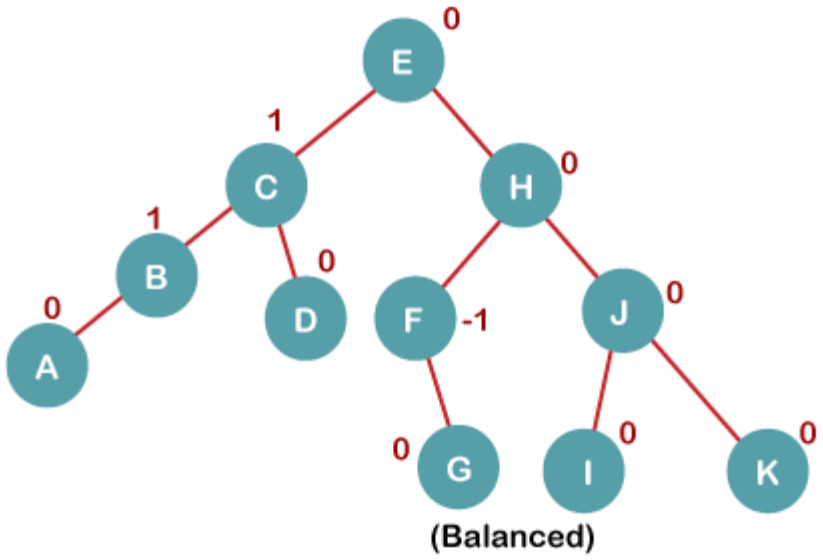
(Balanced)

6. Insert K



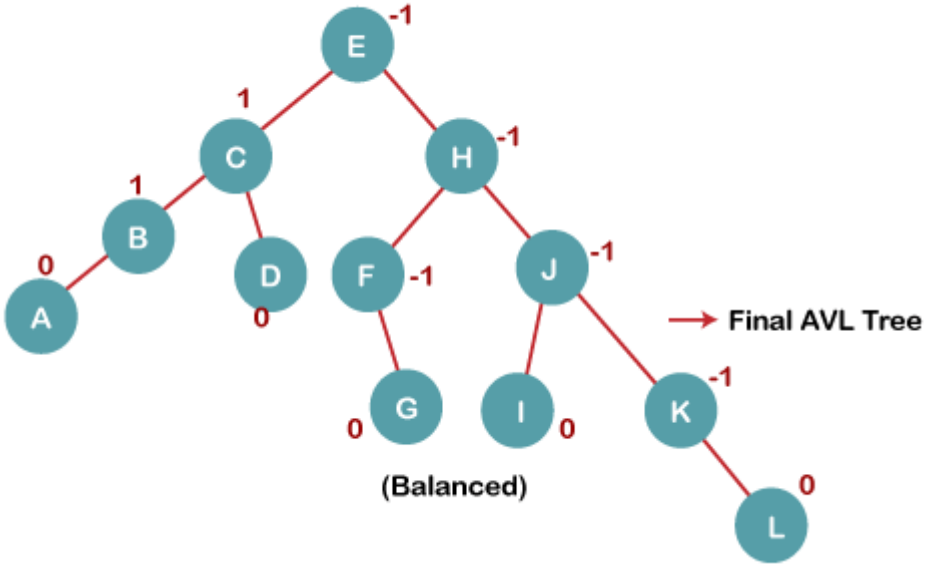
On inserting K, BST becomes unbalanced as the Balance Factor of I is -2. Since the BST is right-skewed from I to K, hence we will perform RR Rotation on the node I.

The resultant balanced tree after RR rotation is:



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



Tree operations on each of the trees and their algorithms with complexity analysis:

Basic operations of a tree

Here are a few basic operations you can perform on a tree: