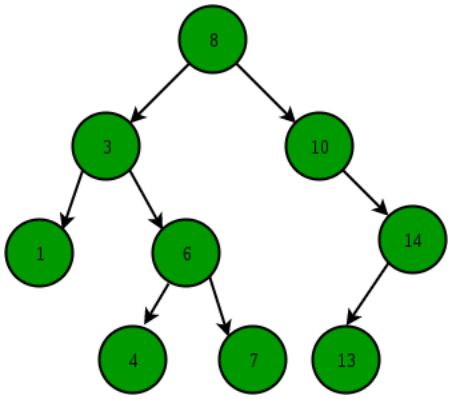
Introduction to Binary Search Trees

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than or equal to the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.
  There must be no duplicate nodes.

Sample Binary Search Tree:



The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast in comparison to normal Binary Trees. If there is no ordering, then we may have to compare every key to search a given key.

Searching a Key

Using the property of Binary Search Tree, we can search for an element in O(h) time complexity where **h** is the height of the given BST.

To search a given key in Binary Search Tree, first compare it with root, if the key is present at root, return root. If the key is greater than the root's key, we recur for the right subtree of the root node. Otherwise, we recur for the left subtree.

Implementation:

C++

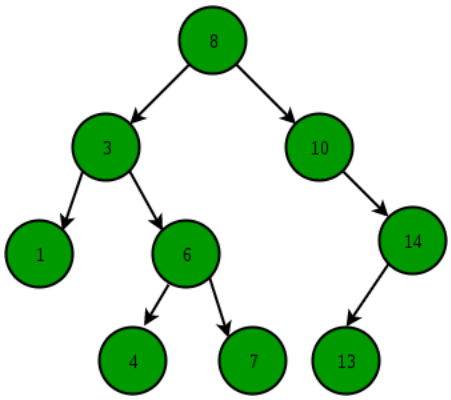
```
1
2 // C++ function to search a given key in a given BST
3
```

```
/* Node Structure:

   struct Node
   {
       int key;
       Node* left, right;
   };
*/

struct node* search(struct node* root, int key)
{
    // Base Cases: root is null or key is present at root
    if (root == NULL || root->key == key)
       return root;

    // Key is greater than root's key
    if (root->key < key)
       return search(root->right, key);

    // Key is smaller than root's key
    return search(root->left, key);
}

```

Java

Illustration to search a key 6 in the below BST:

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

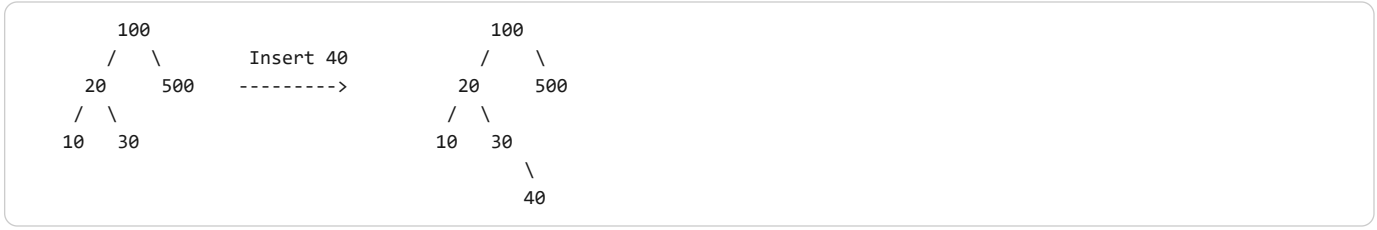


Step 1: Compare 6 with 8.
Since 6 is less than 8,
move to left subtree.
Step 2: Compare 6 with 3.
Since 6 is greater than 3,
move to its right subtree.
Step 3: Comapre 6 with 6.
Node Found.

Insertion of a Key

Inserting a new node in the Binary Search Tree is always done at the leaf nodes to maintain the order of nodes in the Tree. The idea is to start searching the given node to be inserted from the root node till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

For Example:



Implementation:

C++

```cpp

// C++ program to demonstrate insert
// operation in binary search tree

#include<bits/stdc++.h>
using namespace std;

// Binary Search Tree node
struct node
{
    int key;
    struct node *left, *right;
};

// A utility function to create a new BST node
struct node *newNode(int item)
{
    node *temp = new node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        cout<<root->key<<" ";
```

Run

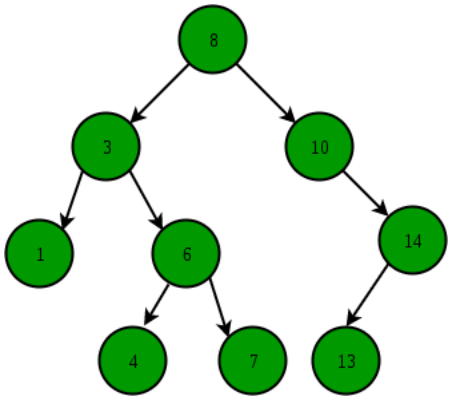
Java

Output:

```
20 30 40 50 60 70 80
```

Illustration to insert 2 in the below tree:

1. Start from root.
2. Compare the element to be inserted with root, if it is less than root, then recur for left-subtree, else recur for right-subtree.
3. After reaching end,just insert that node at left(if less than current leaf node) else right.



Element to be inserted: 2

Step 1: Compare 2 with 8.
Since 2 is less than 8,
move to left subtree.
Step 2: Compare 2 with 3.
Since 2 is less than 3,
move to its left subtree.
Step 3: Comapre 2 with 1.
Since 2 is greater than 1 and also 1 is a leaf node.
Insert 2 as its right child of node 1.

Time Complexity: The worst case time complexity of search and insert operations is O(h) where **h** is height of Binary Search Tree. In the worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of search and insert operation may become O(n).

− Deletion in a Binary Search Tree

We have discussed the Search and Insert operations in BST in the previous post. In this post, delete operation is discussed.

That is, given a Binary Search Tree and a node to be deleted. The task is to search that node in the given BST and delete it from the BST if it is present.

When we delete a node, three cases may arise:

1. ***Node to be deleted is leaf:*** Simply remove from the tree.

```
      50                            50
   /     \         delete(20)      /   \
  30      70       --------->    30     70
 /  \    /  \                     \    /  \
20   40  60   80                   40  60   80
```

2. ***Node to be deleted has only one child:*** Copy the child to the node and delete the child.

```
    50                            50
 /     \         delete(30)      /   \
30      70       --------->    40     70
  \    /  \                           /  \
  40  60   80                       60   80
```

3. ***Node to be deleted has two children:*** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

Note: *The inorder successor can be obtained by finding the minimum value in the right child of the node.*
Below is the implementation of the above three cases:

C++

```
// C++ program to demonstrate delete
// operation in binary search tree
#include<bits/stdc++.h>
using namespace std;

// BST Node
struct node
{
    int key;
    struct node *left, *right;
};

// A utility function to create a new BST node
struct node *newNode(int item)
{
    node *temp = new node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        cout<<root->key<<" ";
        inorder(root->right);
```

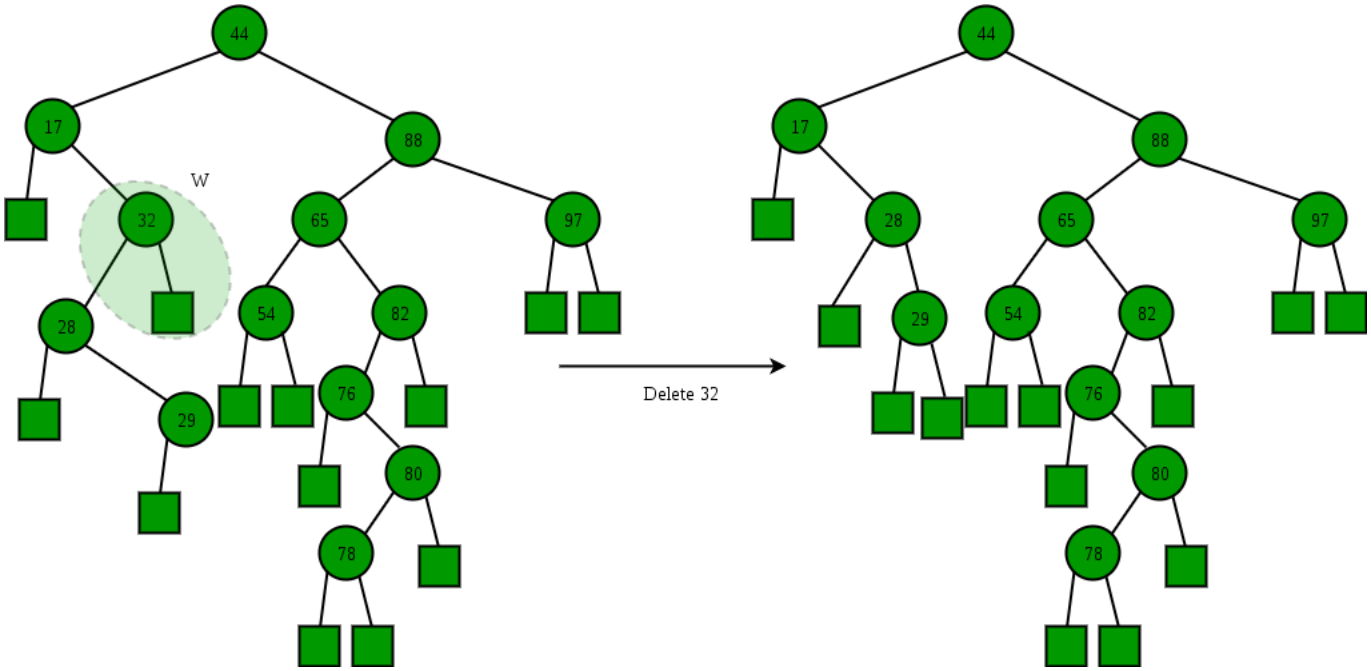
Run

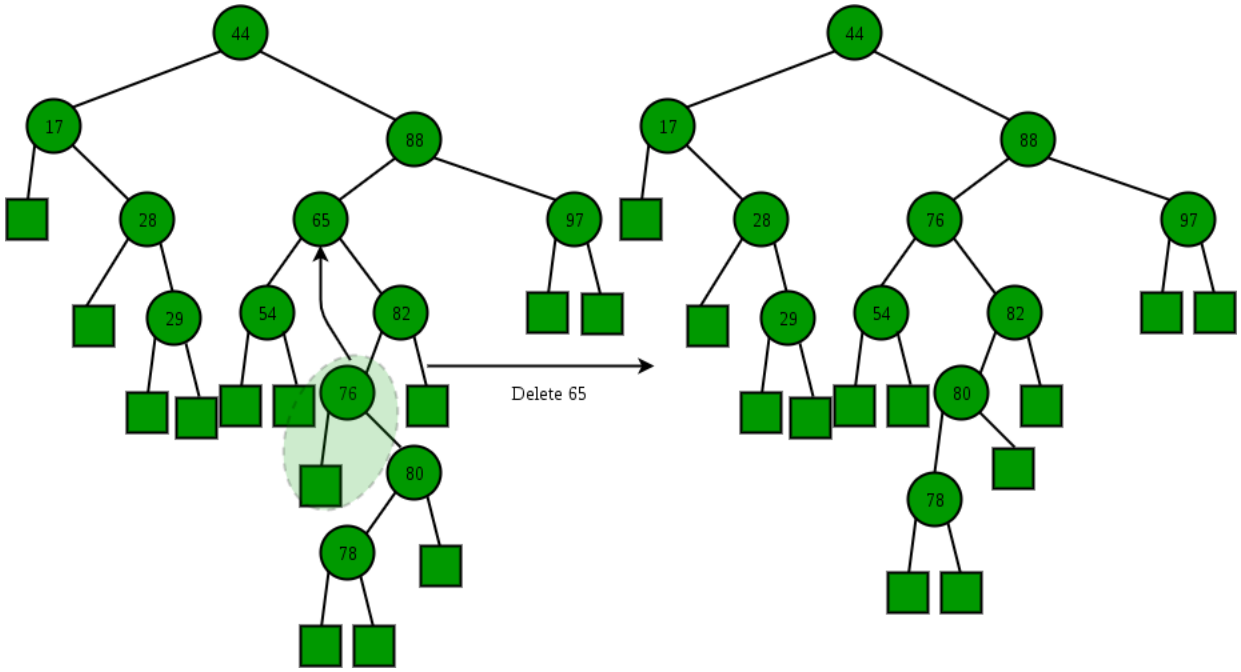
Java

Output:

```
Inorder traversal of the given tree
20 30 40 50 60 70 80
Delete 20
Inorder traversal of the modified tree
30 40 50 60 70 80
Delete 30
Inorder traversal of the modified tree
40 50 60 70 80
Delete 50
Inorder traversal of the modified tree
40 60 70 80
```

Illustration:





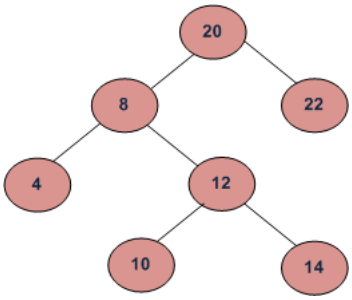
- Finding LCA in Binary Search Tree

We have already seen the definition of LCA and finding LCA of any two nodes in a Binary Tree in the previous post.

Let's look at the process of finding LCA in a Binary Search Tree.

Problem: Given values of two values n1 and n2 in a Binary Search Tree, find the **L**owest **C**ommon **A**ncestor (LCA). For Simplicity, you may assume that both the values exist in the tree.

Consider the below BST:



In the above BST:

```
LCA of 10 and 14 is 12
LCA of 14 and 8 is 8
LCA of 10 and 22 is 20
```

> *The **LCA** or **Lowest Common Ancestor** of any two nodes N1 and N2 is defined as the common ancestor of both the nodes which is closest to them. That is the distance of the common ancestor from the nodes N1 and N2 should be least possible.*

Finding LCA

Since Binary Search Tree is also a Binary Tree, we can apply the same process of finding LCA of two nodes in BST as that of binary trees. But finding LCA in a Binary Tree takes O(N) time complexity.

However, we can solve this problem using BST properties. We can **recursively traverse** the BST from root. The main idea of the solution is, while traversing from top to bottom, the first node **n** we encounter with value between **n1** and **n2**, i.e., **n1 <= n <= n2**, is LCA of n1 and n2 (assuming that n1 < n2). So just recursively traverse the BST, if node's value is greater than both n1 and n2 then our LCA lies in the left subtree of the node, if it's is smaller than both n1 and n2, then LCA lies on the right subtree. Otherwise root is LCA (assuming that both n1 and n2 are present in BST).

Below is the implementation of this approach:

C++

```
// A recursive CPP program to find
// LCA of two nodes n1 and n2.

#include <bits/stdc++.h>
using namespace std;

class node
{
    public:
    int data;
    node* left, *right;
};

// Function to find LCA of n1 and n2.
// The function assumes that both
// n1 and n2 are present in BST
node *lca(node* root, int n1, int n2)
{
    if (root == NULL) return NULL;

    // If both n1 and n2 are smaller
    // than root, then LCA lies in left
    if (root->data > n1 && root->data > n2)
        return lca(root->left, n1, n2);

    // If both n1 and n2 are greater than
    // root, then LCA lies in right
    if (root->data < n1 && root->data < n2)
        return lca(root->right, n1, n2);
```

Run

Java

Output:

```
LCA of 10 and 14 is 12
LCA of 14 and 8 is 8
LCA of 10 and 22 is 20
```

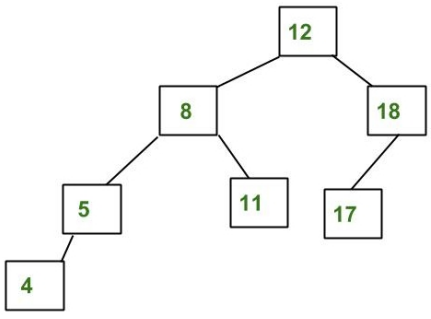
The time complexity of the above solution is O(h) where h is the height of the tree. Also, the above solution requires O(h) extra space in function call stack for recursive function calls. We can avoid extra space using **iterative solution**.

```
/* Function to find LCA of n1 and n2.
   The function assumes that both
   n1 and n2 are present in BST */
struct node *lca(struct node* root, int n1, int n2)
{
    while (root != NULL)
    {
        // If both n1 and n2 are smaller than root,
        // then LCA lies in left subtree
        if (root->data > n1 && root->data > n2)
           root = root->left;

        // If both n1 and n2 are greater than root,
        // then LCA lies in the right subtree
        else if (root->data < n1 && root->data < n2)
           root = root->right;

        else break;
    }

    return root;
}

```

− Introduction to AVL Tree

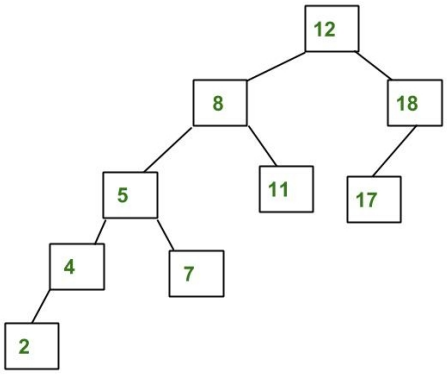
AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

An Example Tree that is an AVL Tree:



The above tree is AVL because differences between heights of left and right subtrees for every node is less than or equal to 1.

An Example Tree that is NOT an AVL Tree:



The above tree is not AVL because differences between heights of left and right subtrees for 8 and 18 is greater than 1.

Why AVL 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(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree.

Insertion

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

1. Left Rotation
2. Right Rotation

```
T1, T2 and T3 are subtrees of the tree
rooted with y (on the left side) or x (on
the right side)
```

```
     y                               x
    / \     Right Rotation          /  \
   x   T3   - - - - - - - >       T1   y
  / \       < - - - - - - -           / \
 T1  T2     Left Rotation            T2  T3
```

```
Keys in both of the above trees follow the
following order:
 keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)

So BST property is not violated anywhere.
```

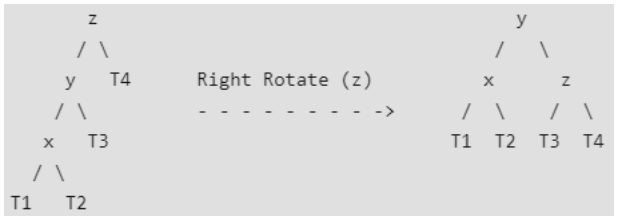
Steps to follow for insertion: Let the newly inserted node be **w**.

1. Perform standard BST insert for w.
2. Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.
3. Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that need to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:
    - y is left child of z and x is left child of y (Left Left Case)
    - y is left child of z and x is right child of y (Left Right Case)
    - y is right child of z and x is right child of y (Right Right Case)
    - y is right child of z and x is left child of y (Right Left Case)

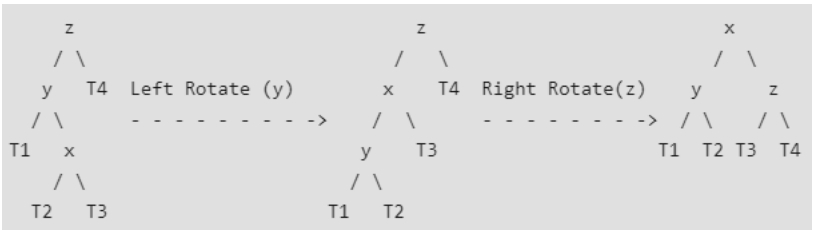
Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re-balance the subtree rooted with z and the complete tree becomes balanced as the height of subtree (After appropriate rotations) rooted with z becomes same as it was before insertion. (See this (http://www.youtube.com/watch?v=TbvhGcf6UJU)video lecture for proof)

a) Left Left Case

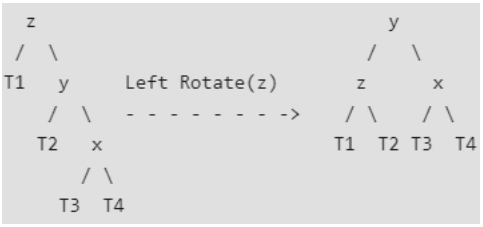
T1, T2, T3 and T4 are subtrees.



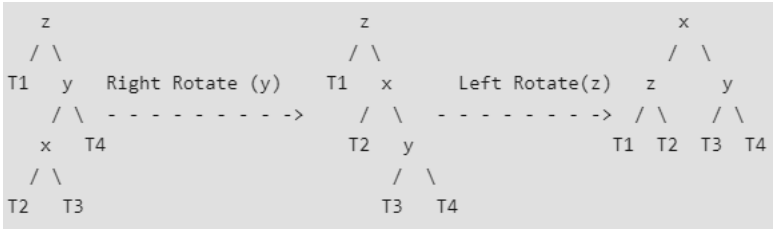
b) Left Right Case



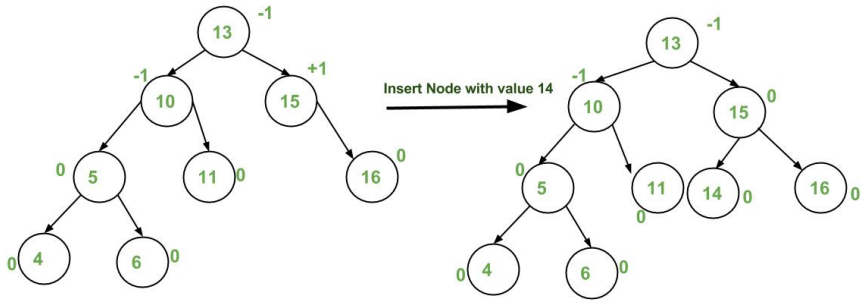
c) Right Right Case



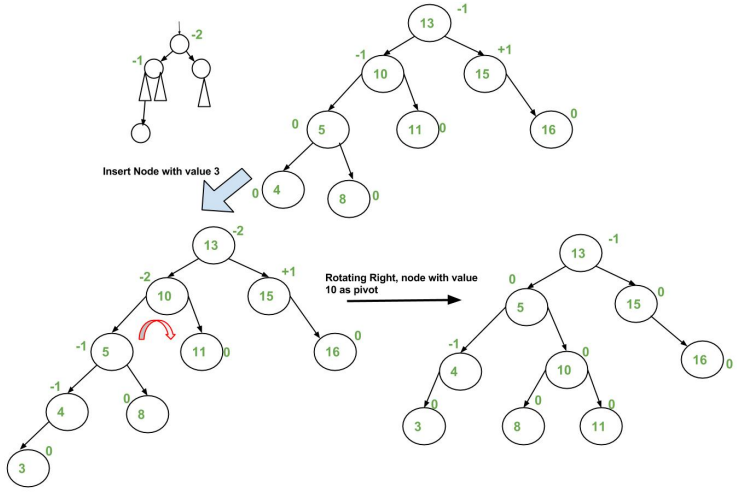
d) Right Left Case



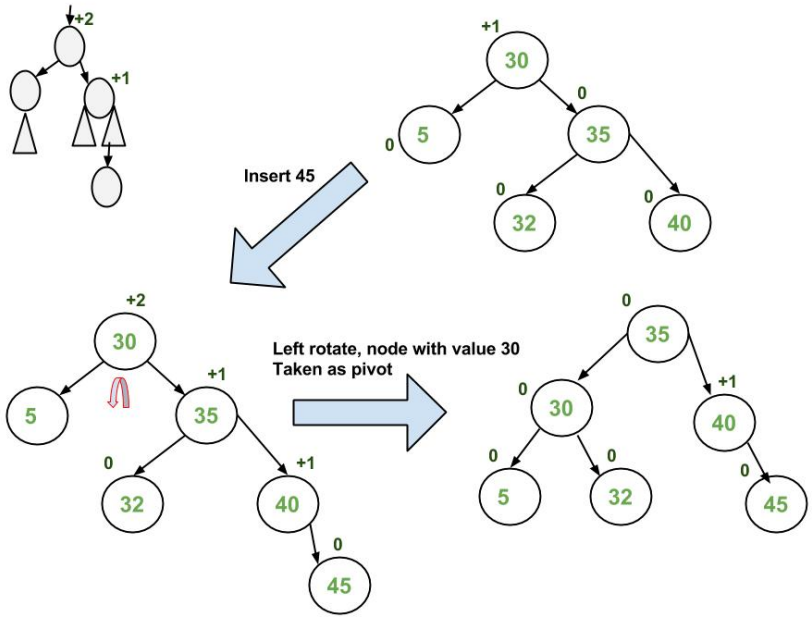
Insertion Examples:



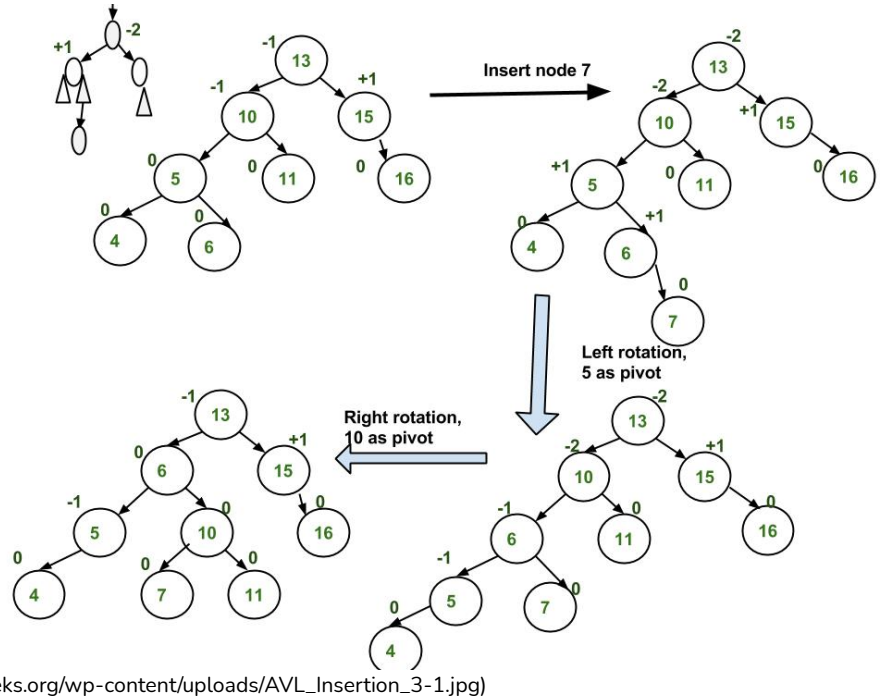
(https://media.geeksforgeeks.org/wp-content/uploads/AVL-Insertion-1.jpg)



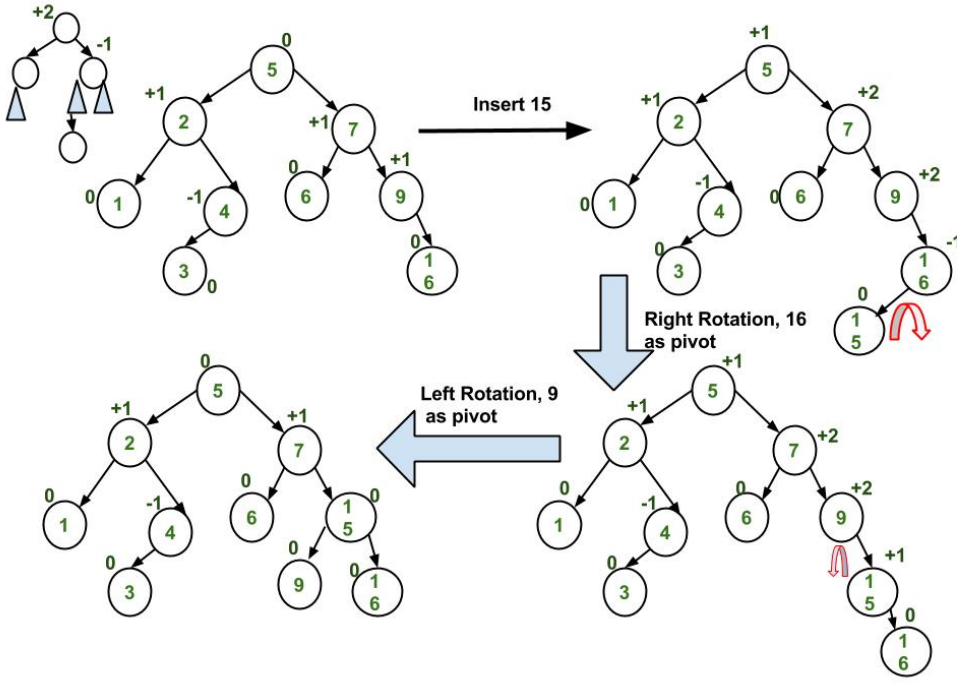
(https://media.geeksforgeeks.org/wp-content/uploads/AVL-Insertion1-1.jpg)



(https://media.geeksforgeeks.org/wp-content/uploads/AVL_INSERTION2-1.jpg)



(https://media.geeksforgeeks.org/wp-content/uploads/AVL_Insertion_3-1.jpg)



(https://media.geeksforgeeks.org/wp-content/uploads/AVL_Tree_4-1.jpg)

Time Complexity: The rotation operations (left and right rotate) take constant time as only a few pointers are being changed there. Updating the height and getting the balance factor also takes constant time. So the time complexity of AVL insert remains same as BST insert which is O(h) where h is the height of the tree. Since the AVL tree is balanced, the height is O(Logn). So time complexity of AVL insert is O(Logn).

5th Floor, A-118,
Sector-136, Noida, Uttar Pradesh - 201305

feedback@geeksforgeeks.org (mailto:feedback@geeksforgeeks.org)

LIVE BATCHES

(https://www.facebook.com/geeksforgeeks.org/)(https://www.instagram.com/geeks_for_geeks/)(https://in.linkedin.com/company/geeksforgeeks)(htt

Company

About Us (https://www.geeksforgeeks.org/about/)

Careers (https://www.geeksforgeeks.org/careers/)

Privacy Policy (https://www.geeksforgeeks.org/privacy-policy/)

Contact Us (https://www.geeksforgeeks.org/about/contact-us/)

Terms of Service (https://practice.geeksforgeeks.org/terms-of-service/)

Learn

Algorithms (https://www.geeksforgeeks.org/fundamentals-of-algorithms/)

Data Structures (https://www.geeksforgeeks.org/data-structures/)

Languages (https://www.geeksforgeeks.org/category/program-output/)

CS Subjects (https://www.geeksforgeeks.org/articles-on-computer-science-subjects-gq/)

Video Tutorials (https://www.youtube.com/geeksforgeeksvideos/)

Practice

Courses (https://practice.geeksforgeeks.org/courses/)

Company-wise (https://practice.geeksforgeeks.org/company-tags/)

Topic-wise (https://practice.geeksforgeeks.org/topic-tags/)

How to begin? (https://practice.geeksforgeeks.org/faq.php)

Contribute

Write an Article (https://www.geeksforgeeks.org/contribute/)

Write Interview Experience (https://www.geeksforgeeks.org/write-interview-experience/)

Internships (https://www.geeksforgeeks.org/internship/)

Videos (https://www.geeksforgeeks.org/how-to-contribute-videos-to-geeksforgeeks/)