In **C++** the **stack** memory is where local variables get stored/constructed. The **stack** is also used to hold parameters passed to functions. ... Generally speaking in **C++** objects allocated with new, or blocks of memory allocated with the likes of malloc ends up on the **heap**.

An AVL tree is balanced, so its height is O(log N) where N is the number of nodes.

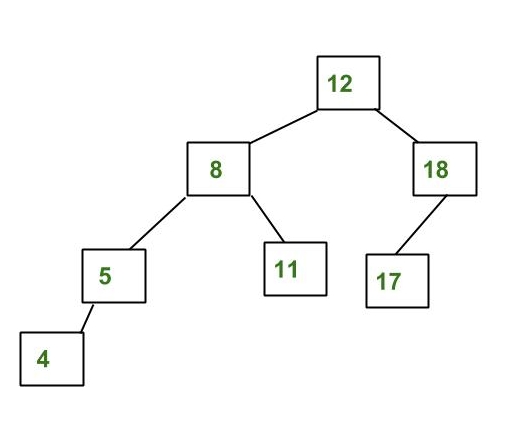
The rotation routines are all themselves O(1), so they don't significantly impact the insert operation complexity, which is still O(k) where k is the height of the tree. But as noted before, this height is O(log N), so insertion into an AVL tree has a worst case O(log N).

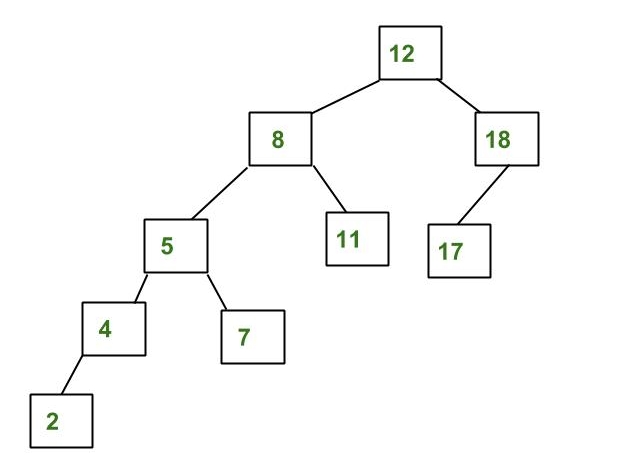
Searching an AVL tree is completely unchanged from BST's, and so also takes time proportional to the height of the tree, making O(log N).

Removing nodes from a binary tree also requires rotations, but remains O(log N) as well.

AVL Tree | Set 1 (Insertion)

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 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 needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:  
a) y is left child of z and x is left child of y (Left Left Case)  
b) y is left child of z and x is right child of y (Left Right Case)  
c) y is right child of z and x is right child of y (Right Right Case)  
d) 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.

**a) Left Left Case**

T1, T2, T3 and T4 are subtrees.

z y

/ \ / \

y T4 Right Rotate (z) x z

/ \ - - - - - - - - -> / \ / \

x T3 T1 T2 T3 T4

/ \

T1 T2

**b) Left Right Case**

z z x

/ \ / \ / \

y T4 Left Rotate (y) x T4 Right Rotate(z) y z

/ \ - - - - - - - - -> / \ - - - - - - - -> / \ / \

T1 x y T3 T1 T2 T3 T4

/ \ / \

T2 T3 T1 T2

**c) Right Right Case**

z y

/ \ / \

T1 y Left Rotate(z) z x

/ \ - - - - - - - -> / \ / \

T2 x T1 T2 T3 T4

/ \

T3 T4

**d) Right Left Case**

z z x

/ \ / \ / \

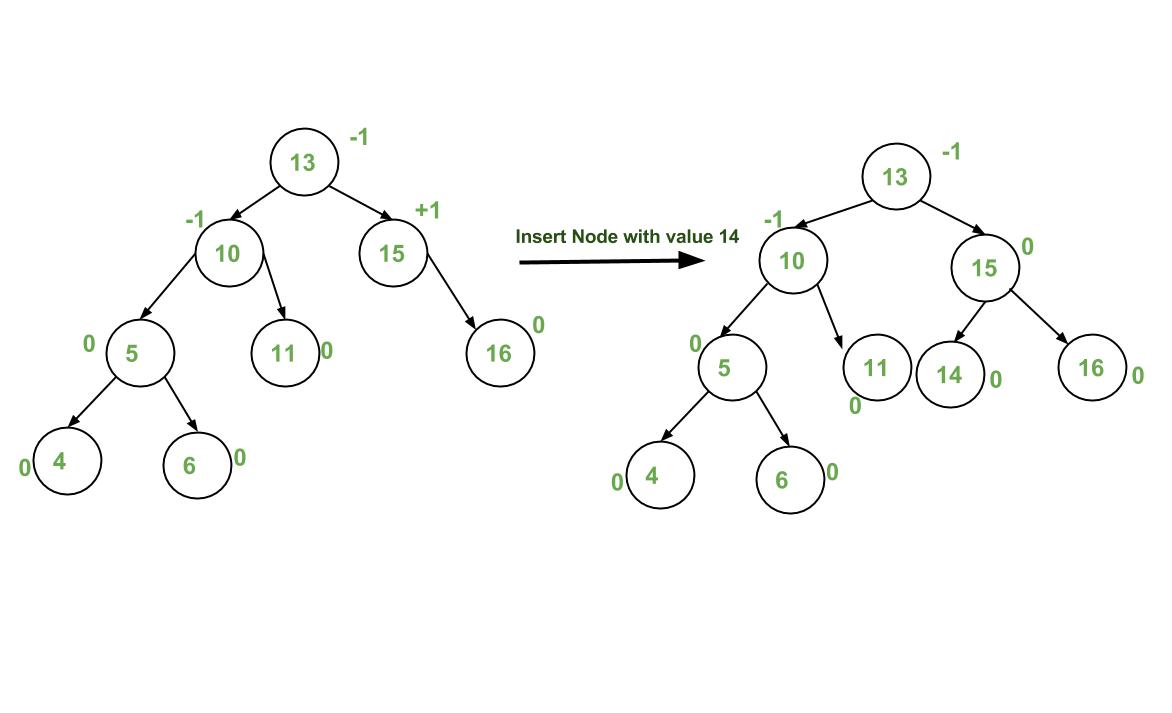
T1 y Right Rotate (y) T1 x Left Rotate(z) z y

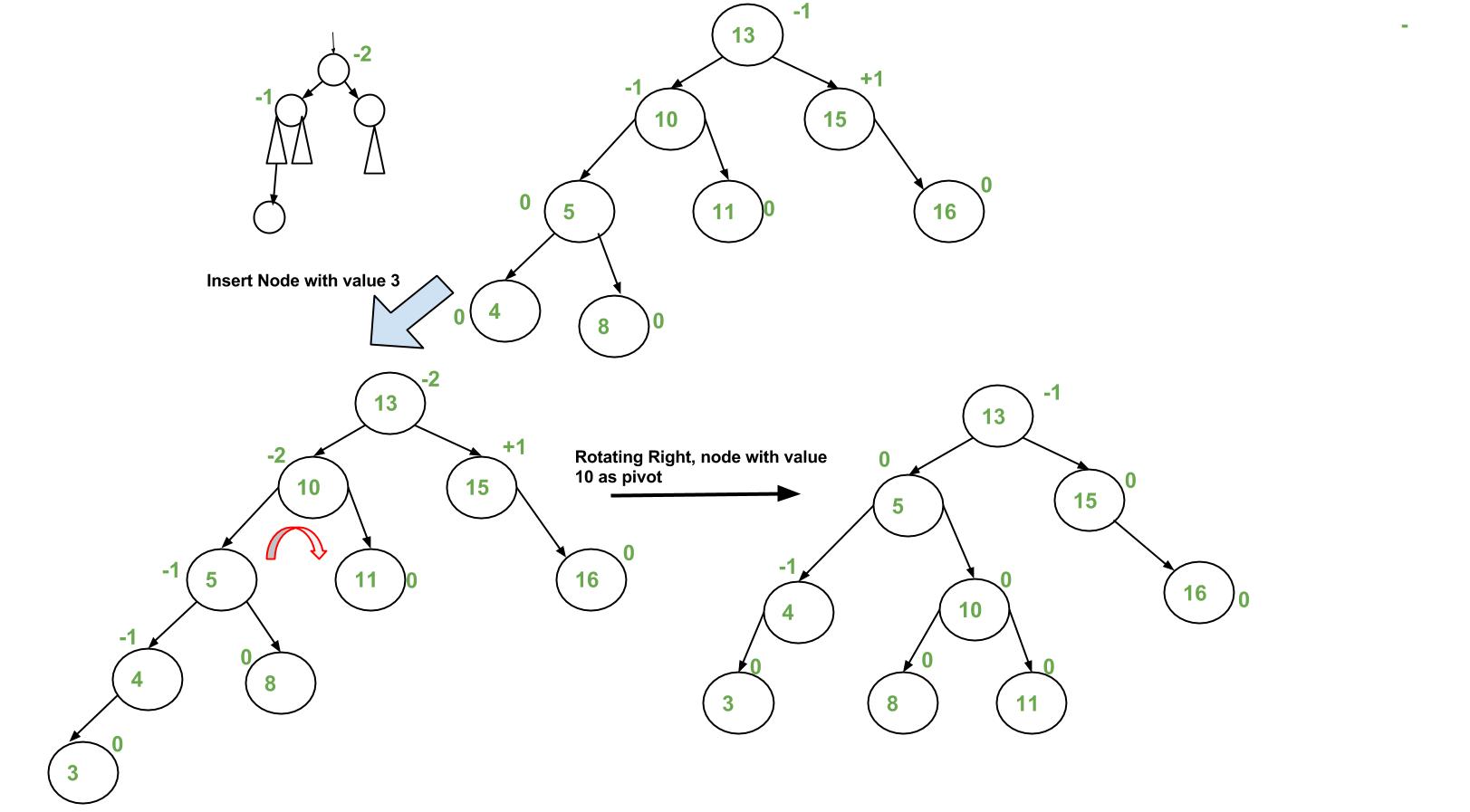
/ \ - - - - - - - - -> / \ - - - - - - - -> / \ / \

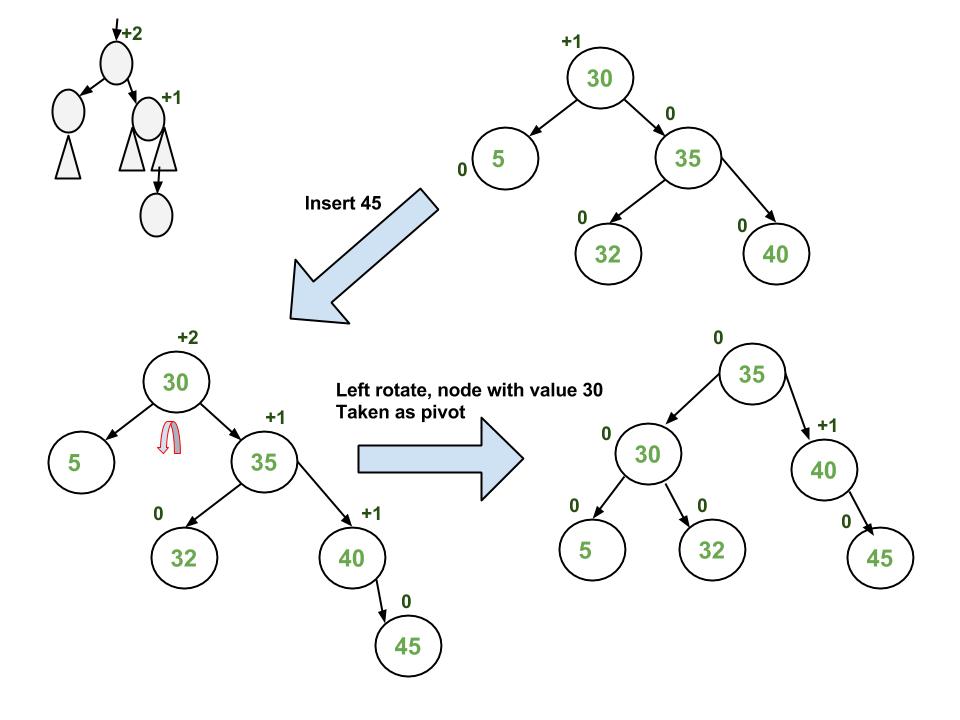
x T4 T2 y T1 T2 T3 T4

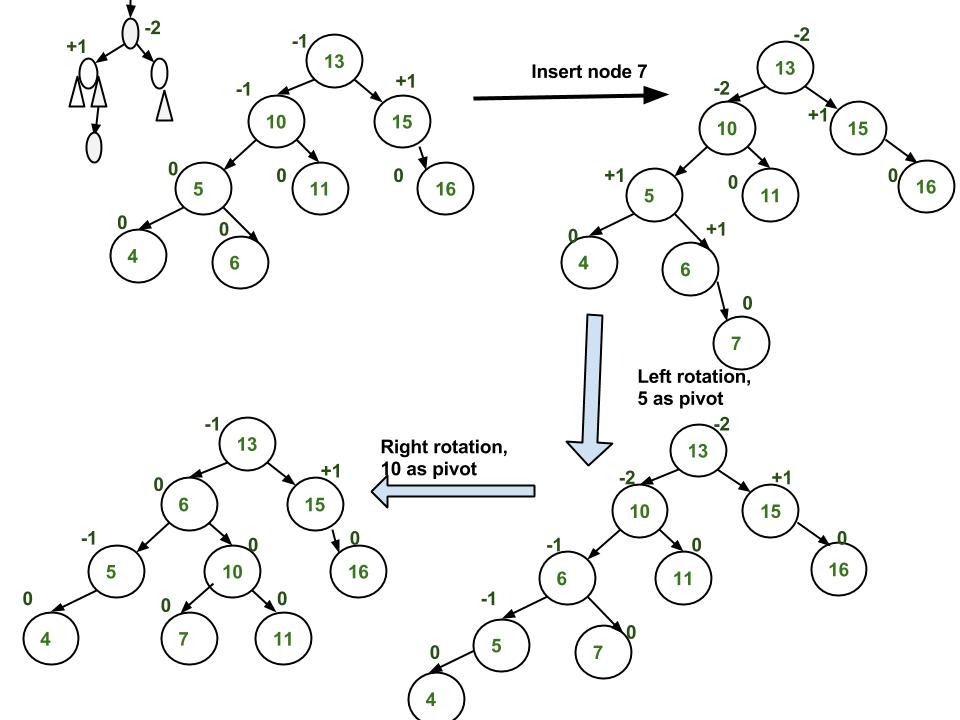
/ \ / \

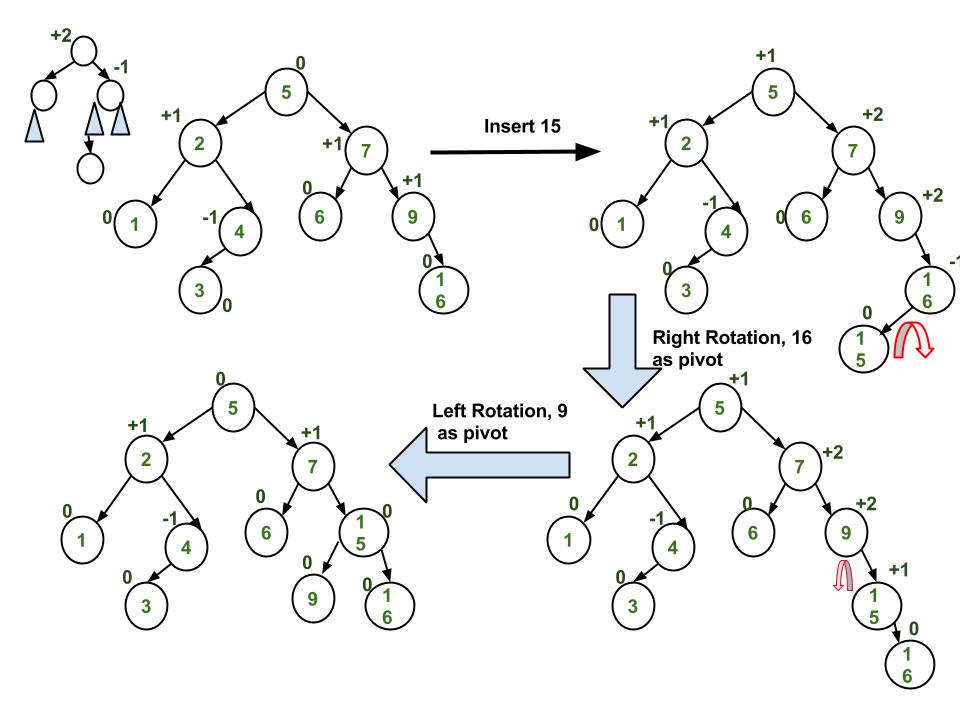
T2 T3 T3 T4

**Insertion Examples:**  
[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/AVL-Insertion-1.jpg)

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

[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/AVL_INSERTION2-1.jpg)

[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/AVL_Insertion_3-1.jpg)



**implementation**  
Following is the implementation for AVL Tree Insertion. The following implementation uses the recursive BST insert to insert a new node. In the recursive BST insert, after insertion, we get pointers to all ancestors one by one in a bottom-up manner. So we don’t need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.  
1) Perform the normal BST insertion.  
2) The current node must be one of the ancestors of the newly inserted node. Update the height of the current node.  
3) Get the balance factor (left subtree height – right subtree height) of the current node.  
4) If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or left Right case. To check whether it is left left case or not, compare the newly inserted key with the key in left subtree root.  
5) If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right-Left case. To check whether it is Right Right case or not, compare the newly inserted key with the key in right subtree root.

// C program to insert a node in AVL tree

#include<stdio.h>

#include<stdlib.h>

// An AVL tree node

struct Node

{

    int key;

    struct Node \*left;

    struct Node \*right;

    int height;

};

// A utility function to get maximum of two integers

int max(int a, int b);

// A utility function to get the height of the tree

int height(struct Node \*N)

{

    if (N == NULL)

        return 0;

    return N->height;

}

// A utility function to get maximum of two integers

int max(int a, int b)

{

    return (a > b)? a : b;

}

/\* Helper function that allocates a new node with the given key and

    NULL left and right pointers. \*/

struct Node\* newNode(int key)

{

    struct Node\* node = (struct Node\*)

                        malloc(sizeof(struct Node));

    node->key   = key;

    node->left   = NULL;

    node->right  = NULL;

    node->height = 1;  // new node is initially added at leaf

    return(node);

}

// A utility function to right rotate subtree rooted with y

// See the diagram given above.

struct Node \*rightRotate(struct Node \*y)

{

    struct Node \*x = y->left;

    struct Node \*T2 = x->right;

    // Perform rotation

    x->right = y;

    y->left = T2;

    // Update heights

    y->height = max(height(y->left), height(y->right))+1;

    x->height = max(height(x->left), height(x->right))+1;

    // Return new root

    return x;

}

// A utility function to left rotate subtree rooted with x

// See the diagram given above.

struct Node \*leftRotate(struct Node \*x)

{

    struct Node \*y = x->right;

    struct Node \*T2 = y->left;

    // Perform rotation

    y->left = x;

    x->right = T2;

    //  Update heights

    x->height = max(height(x->left), height(x->right))+1;

    y->height = max(height(y->left), height(y->right))+1;

    // Return new root

    return y;

}

// Get Balance factor of node N

int getBalance(struct Node \*N)

{

    if (N == NULL)

        return 0;

    return height(N->left) - height(N->right);

}

// Recursive function to insert a key in the subtree rooted

// with node and returns the new root of the subtree.

struct Node\* insert(struct Node\* node, int key)

{

    /\* 1.  Perform the normal BST insertion \*/

    if (node == NULL)

        return(newNode(key));

    if (key < node->key)

        node->left  = insert(node->left, key);

    else if (key > node->key)

        node->right = insert(node->right, key);

    else // Equal keys are not allowed in BST

        return node;

    /\* 2. Update height of this ancestor node \*/

    node->height = 1 + max(height(node->left),

                           height(node->right));

    /\* 3. Get the balance factor of this ancestor

          node to check whether this node became

          unbalanced \*/

    int balance = getBalance(node);

    // If this node becomes unbalanced, then

    // there are 4 cases

    // Left Left Case

    if (balance > 1 && key < node->left->key)

        return rightRotate(node);

    // Right Right Case

    if (balance < -1 && key > node->right->key)

        return leftRotate(node);

    // Left Right Case

    if (balance > 1 && key > node->left->key)

    {

        node->left =  leftRotate(node->left);

        return rightRotate(node);

    }

    // Right Left Case

    if (balance < -1 && key < node->right->key)

    {

        node->right = rightRotate(node->right);

        return leftRotate(node);

    }

    /\* return the (unchanged) node pointer \*/

    return node;

}

// A utility function to print preorder traversal

// of the tree.

// The function also prints height of every node

void preOrder(struct Node \*root)

{

    if(root != NULL)

    {

        printf("%d ", root->key);

        preOrder(root->left);

        preOrder(root->right);

    }

}

/\* Drier program to test above function\*/

int main()

{

  struct Node \*root = NULL;

  /\* Constructing tree given in the above figure \*/

  root = insert(root, 10);

  root = insert(root, 20);

  root = insert(root, 30);

  root = insert(root, 40);

  root = insert(root, 50);

  root = insert(root, 25);

  /\* The constructed AVL Tree would be

            30

           /  \

         20   40

        /  \     \

       10  25    50

  \*/

  printf("Preorder traversal of the constructed AVL"

         " tree is \n");

  preOrder(root);

  return 0;

}

**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 AVL tree is balanced, the height is O(Logn). So time complexity of AVL insert is O(Logn).

**Comparison with Red Black Tree**  
The AVL tree and other self-balancing search trees like Red Black are useful to get all basic operations done in O(log n) time. 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 many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is the more frequent operation,

# Red-Black Tree | Set 1 (Introduction)

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules.  
[](https://www.geeksforgeeks.org/wp-content/uploads/RedBlackTree.png)  
**1)**Every node has a color either red or black.

**2)**Root of 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 root to a NULL node has same number of black nodes.

**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 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 a Red-Black tree is always O(Logn) where n is the number of nodes in the tree.

**Comparison with**[**AVL Tree**](https://www.geeksforgeeks.org/avl-tree-set-1-insertion/)  
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 many 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.

**How does a Red-Black Tree ensure balance?**  
A simple example to understand balancing is, a chain of 3 nodes is not possible in the red-black tree. We can try any combination of colours and see all of them violate Red-Black tree property.

A chain of 3 nodes is nodes is not possible in Red-Black Trees.

Following are **NOT** Red-Black Trees

**30** **30** **30**

/ \ / \ / \

**20** NIL **20 N**IL **20** NIL

/ \ / \ / \

**10** NIL **10** NIL **10** NIL

Violates Violates Violates

Property 4. Property 4 Property 3

Following are different possible Red-Black Trees with above 3 keys

**20**  **20**

/ \ / \

**10 30** **10 30**

/ \ / \ / \ / \

NIL NIL NIL NIL NIL NIL NIL NIL

From the above examples, we get some idea how Red-Black trees ensure balance. Following is an important fact about balancing in Red-Black Trees.

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

***Every Red Black Tree with n nodes has height <=***2Log2(n+1)

This can be proved using following facts:  
1) For a general Binary Tree, let **k** be the minimum number of nodes on all root to NULL paths, then n >= 2k – 1 (Ex. If k is 3, then n is atleast 7). This expression can also be written as k <= 2Log2(n+1)

2) From property 4 of Red-Black trees and above claim, we can say in a Red-Black Tree with n nodes, there is a root to leaf path with at-most Log2(n+1) black nodes.

3) From property 3 of Red-Black trees, we can claim that the number black nodes in a Red-Black tree is at least ⌊ n/2 ⌋ where n is the total number of nodes.

From above 2 points, we can conclude the fact that Red Black Tree with **n** nodes has height <= 2Log2(n+1)

In this post, we introduced Red-Black trees and discussed how balance is ensured. The hard part is to maintain balance when keys are added and removed. We will soon be discussing insertion and deletion operations in coming posts on the Red-Black tree.