# Data Structures Lab

***Session 10***

**Course:** Data Structures (CS2001) **Semester:** Fall 2021

# Instructor: T.A: N/A

**Note:**

* Lab manual covers the following topics

**{**AVL & its Rotations, basic utility functions **}**

* Maintain discipline during the lab.
* Just raise your hand if you have any problems.
* Completing all tasks of each lab is compulsory.
* Get your lab checked at the end of the session.

# 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. The valid BF range lies in

{ -1, 0, +1. In an AVL tree, every node maintains extra information known as the balance factor.

**class AVLTree { public:**

**TreeNode \* root;**

**AVLTree() { root = NULL;**

**}**

**bool isTreeEmpty() { if (root == NULL) {**

**return true;**

**} else {**

**return false;**

**}**

**}**

**class TreeNode { public:**

**int value; TreeNode \* left; TreeNode \* right;**

**TreeNode() {**

**value = 0; left = NULL; right = NULL;**

**}**

**TreeNode(int v) { value = v; left = NULL; right = NULL;**

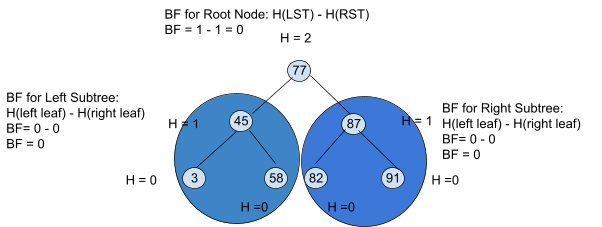
**}**

**}**

# Balance Factor:

The balance factor of a node is the diﬀerence between the heights of the left and right subtrees of that node. The balance factor of a node is calculated by subtracting the height of the right subtree from the height of the left subtree (OR) height of the right subtree - the height of the left subtree.

In the following explanation, we calculate balance factorof the tree as follows:



To calculate the balance factor for the AVL tree, we ﬁrst need to calculate the height of each subtree.

**Get Height**

**int height(TreeNode \* root) { if (root == NULL)**

**return -1; else {**

**/\* compute the height of each subtree \*/**

**int lheight = height(root’s left);**

**int rheight = height(root’s right);**

**// Get Balance factor of node N int getBalanceFactor( TreeNode**

* **node) {**

**if (n == NULL)**

**return -1;//showing no node is present in the tree or leaf node**

**return height(node’s left) - height(node’s right);**

**}**

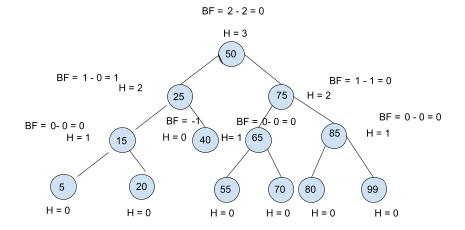
**/\* use the larger one \*/ if (lheight > rheight)**

**return (lheight + 1); else return (rheight + 1);**

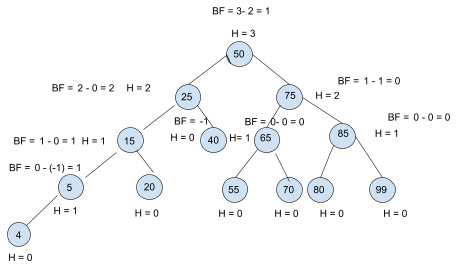
**}**

**}**

Consider the Following Binary Search Tree now:



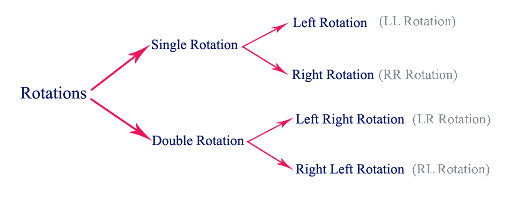
The balance factors for all of the subtrees and complete tree lie in the range of { 1, 0, -1 }

If in case we insert a new node: 4 into the left of node 1: 50, then:

Although the overall balance factor of BST is balanced, however, if you see, the left subtree starting from 25, it has a balance factor of +2, which means it is quite left heavy. To make it balanced, we have to perform some form of rotation:

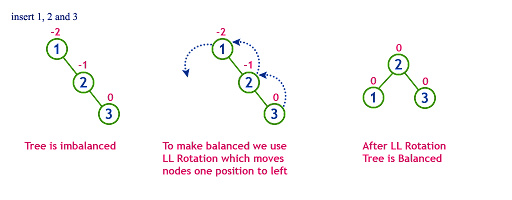
# The AVL Tree Rotations :

In the AVL tree, after performing operations like insertion and deletion we need to check the balance factor of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation we use rotation operations to make the tree balanced. Rotation operations are used to make the tree balanced. There are four rotations and they are classified into two types.



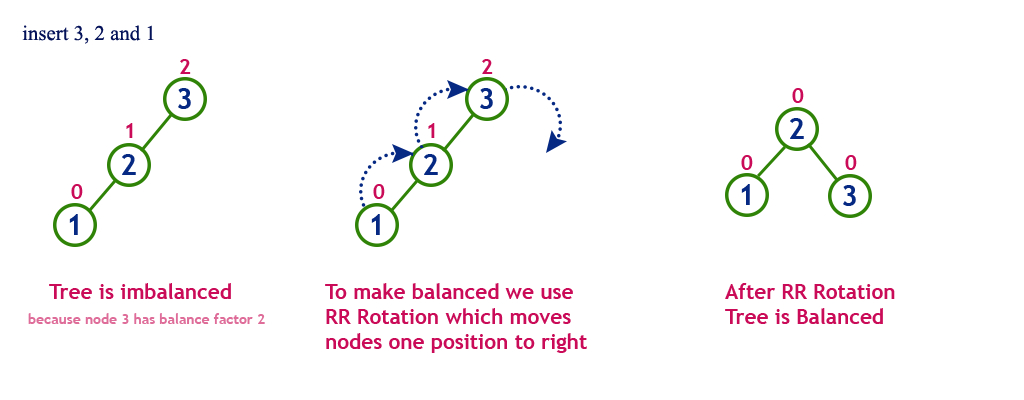
# Single Left Rotation (LL Rotation):

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



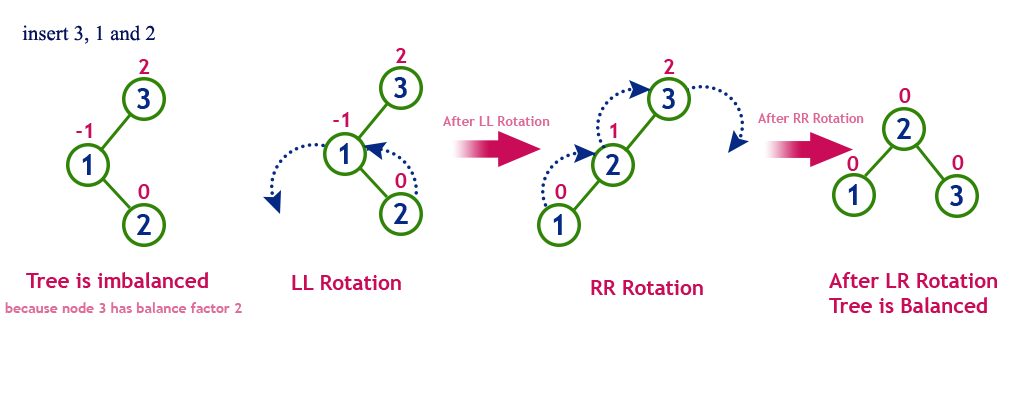
# Single Right Rotation (RR Rotation)

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



# Left Right Rotation (LR Rotation)

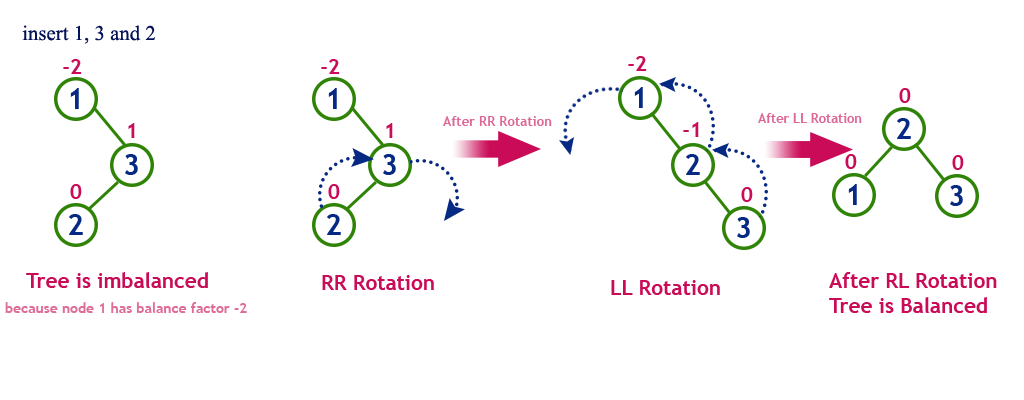
The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at ﬁrst, every node moves one position to the left and one position to the right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree.

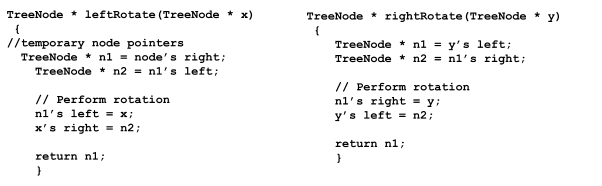


# Right Left Rotation (RL Rotation)

The RL Rotation is a sequence of single right rotation followed by a single left rotation. In

RL Rotation, at ﬁrst, every node moves one position to the right and one position to the left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree





# Operations on an AVL Tree

The following operations are performed on the AVL tree.

* 1. Insertion 25
  2. Search
  3. Deletion

# Insertion:

In an AVL tree, the insertion operation is performed with O(log n) time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows :

**Step 1** - Insert the new element into the tree using Binary Search Tree insertion logic.

**Step 2** - After insertion, check the Balance Factor of every node.

**Step 3** - If the Balance Factor of every node is 0 or 1 or -1 then go for the next operation.

**Step 4** - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for the next operation

**Task-01:** Create an AVL Tree using insertion sequence as { 55, 66, 77, 11, 33, 22, 35, 25, 44, 88,99 }, the program should print the height of tree too.