**Practical No. 10**

**Government College of Engineering, Jalgaon**

**(An Autonomous Institute of Govt. of Maharashtra)**

**Name of Student:**

**PRN:**   **Batch:**

**Date of Performance: Date of Completion:**

**Subject: CO257U DSFL Subject Teacher:- Miss. Prajakta D. Sawle**

# Aim: Implementation of operations on AVL Trees

# Theory:

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.

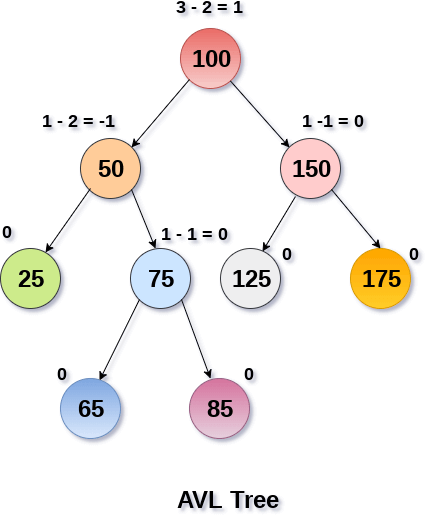
Balance Factor (k) = height (left(k)) - height (right(k))

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.



# 

# 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.

**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.

**Deletion :**Deletion can 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.

**Height of AVL Tree:**

# Height-Balance Property:

# A node in an AVL tree is height-balanced if the heights of its left and right subtrees differ by at most 1.

# The AVL invariant ensures that the tree remains balanced during insertions and deletions.

int getHeight(struct Node\* node) {

if (node == NULL)

return 0;

return node->height;

}

# AVL Rotations

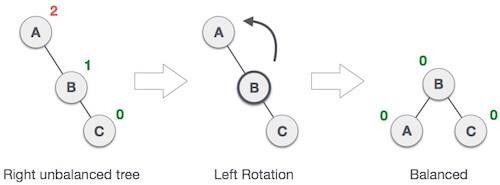
We perform rotation in AVL tree only in case if Balance Factor is other than **-1, 0, and 1**. There are basically four types of rotations which are as follows:

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.

# RR Rotation

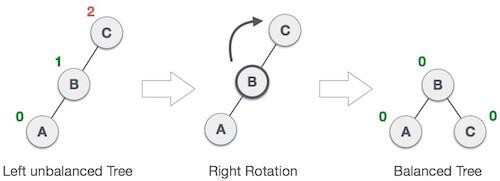
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, [RR rotation](https://www.javatpoint.com/rr-rotation-in-avl-tree) is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



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.

1. 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](https://www.javatpoint.com/ll-rotation-in-avl-tree) 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.

# 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:

|  |  |
| --- | --- |
|  |  |
| AVL Rotations | 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 |
| AVL Rotations | 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**. |
| AVL Rotations | 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** |
| AVL Rotations | 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 |
| AVL Rotations | Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now. |

1. **RL Rotation**

As already discussed, that double rotations are bit tougher than single rotation which has already explained above. [R L rotation](https://www.javatpoint.com/rl-rotation-in-avl-tree) = 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.

|  |  |
| --- | --- |
|  |  |
| AVL Rotations | 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 |
| AVL Rotations | 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**. |
| AVL Rotations | 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. |

|  |  |
| --- | --- |
| AVL Rotations | 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. |
| AVL Rotations | Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now. |

**Conclusion: -** In this way we have executed program in c for AVL Tree.

**Course Teacher**

Prajakta D. Sawle