# ADS LAB ASSINGMENT 5

**Name-Manasvi Mahadev Patil**

**Class- SY IT-C Batch 1**

**PRN-12211105**

**Roll no-03**

Q. Write C/C++ program to perform various operations i.e., insertions, updating height , verifying balance factor and perform rotations on AVL trees. Show final tree is balanced using solved example and given inputs.

# Theory-

An AVL tree (Adelson-Velsky and Landis tree) is a self-balancing binary search tree. It is named after its inventors, Georgy Adelson-Velsky and Evgenii Landis. AVL trees maintain their balance by ensuring that the height of the left and right subtrees of any node (known as the balance factor) does not differ by more than one. When the balance factor of a node violates this property, rotations are performed to restore balance.

Here are some key concepts and theory related to AVL trees:

1. Binary Search Tree (BST): An AVL tree is essentially a binary search tree, which means it maintains the following properties:
   * All nodes in the left subtree of a node have values less than the node's value.
   * All nodes in the right subtree of a node have values greater than the node's value.
   * Both the left and right subtrees are also binary search trees.
2. Balance Factor: The balance factor of a node in an AVL tree is defined as the difference between the height of its left subtree and the height of its right subtree. Mathematically, it's calculated as balanceFactor(node) = height(node->left) - height(node->right).
3. Balanced Tree: An AVL tree is considered balanced if the balance factor of every node in the tree is either -1, 0, or 1. In other words, the tree is balanced when the heights of the left and right subtrees differ by at most one level.
4. Rotations: Rotations are the fundamental operations used to maintain the balance of an AVL tree. There are four types of rotations:
   * Single Left Rotation (LL Rotation): Used to balance a node when its left subtree is taller by 2 or more levels.
   * Single Right Rotation (RR Rotation): Used to balance a node when its right subtree is taller by 2 or more levels.
   * Left-Right Rotation (LR Rotation): A combination of left rotation followed by a right rotation, used to balance a node when its left child's right subtree is taller.
   * Right-Left Rotation (RL Rotation): A combination of right rotation followed by a left rotation, used to balance a node when its right child's left subtree is taller.
5. Insertion: When inserting a new node into an AVL tree, the tree may become unbalanced. After inserting the node, the balance factors of its ancestors are checked, and rotations are performed as needed to maintain balance. The height of each affected node is updated.
6. Deletion: Similarly, when a node is deleted from an AVL tree, it may become unbalanced, and rotations are performed to restore balance. Like insertion, the height of each affected node is updated.
7. Time Complexity: The worst-case time complexity for both insertion and deletion in an AVL tree is O(log n), where n is the number of nodes in the tree. This is because the tree remains balanced, and the maximum height of the tree is logarithmic in the number of nodes.
8. Advantages: AVL trees are particularly useful when you need to maintain a balanced search tree with predictable performance for search, insertion, and deletion operations.

**Programming Lang. Used- C**

**Compiler Used- CodeBlocks**

# CODE-

#include <stdio.h>

#include <stdlib.h>

// Structure for a tree node

struct Node {

int key;

struct Node\* left;

struct Node\* right;

int height;

};

// Function to calculate the height of a node

int height(struct Node\* node) {

if (node == NULL)

return 0;

return node->height;

}

// Function to update the height of a node

int updateHeight(struct Node\* node) {

int leftHeight = height(node->left);

int rightHeight = height(node->right);

return (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;

}

// Function to calculate the balance factor of a node

int balanceFactor(struct Node\* node) {

if (node == NULL)

return 0;

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

}

// Function to perform a right rotation

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 = updateHeight(y);

x->height = updateHeight(x);

return x;

}

// Function to perform a left rotation

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 = updateHeight(x);

y->height = updateHeight(y);

return y;

}

// Function to insert a node into the AVL tree

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

// Standard BST insertion

if (node == NULL) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->key = key;

newNode->left = NULL;

newNode->right = NULL;

newNode->height = 1;

return newNode;

}

if (key < node->key)

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

else if (key > node->key)

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

else

return node; // Duplicate keys not allowed

// Update height of current node

node->height = updateHeight(node);

// Get the balance factor

int balance = balanceFactor(node);

// Perform rotations if needed

if (balance > 1) {

if (key < node->left->key) {

return rightRotate(node);

} else {

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

return rightRotate(node);

}

}

if (balance < -1) {

if (key > node->right->key) {

return leftRotate(node);

} else {

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

return leftRotate(node);

}

}

return node;

}

// Function to print the AVL tree in in-order traversal

void inOrderTraversal(struct Node\* root) {

if (root != NULL) {

inOrderTraversal(root->left);

printf("%d ", root->key);

inOrderTraversal(root->right);

}

}

int main() {

struct Node\* root = NULL;

// Insert some elements into the AVL tree

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 30);

root = insert(root, 40);

root = insert(root, 50);

root = insert(root, 25);

printf("In-order traversal of the AVL tree: ");

inOrderTraversal(root);

return 0;

}

# OUTPUT-

