Title Page

# Project:

## "AVL Tree Implementation"

a. Implement an AVL Tree with Insertion, deletion and rotation to maintain balance.

## Subject:

**Data Structure Lab**

**Submitted to:**

**Mam Arshi Pervaiz**

## Group Members


1. **Muhammad Roman (4118)**
2. **Muhammad Talha (4161)**

# “AVL Tree Implementation”

## Introduction

AVL trees, automatically maintain balance after insertion and deletion using rotations. This project focuses on implementing AVL tree operations such as insertion, deletion, and rotations (left, right, left-right, and right-left) to maintain balance.

## Algorithm for AVL Tree Implementation

This algorithm provides the step-by-step process for inserting, deleting, and balancing nodes in an AVL Tree, ensuring it remains height-balanced.

## 1. Node Structure

Each node consists of:

**key:** The data value of the node.

**height:** The height of the node for balance factor calculation.

**left and right:** Pointers to left and right children.

## 2. Get Height of a Node

**Function:** getHeight(Node* n)

If the node is NULL, return height as 0.

Otherwise, return the height stored in the node.

## 3. Get Balance Factor

**Function:** getBalance(Node* n)

balanceFactor = height(left subtree) - height(right subtree).

If the value is >1 (Left-heavy) or <-1 (Right-heavy), rotations are needed.

## 4. Update Height of a Node

**Function:** updateHeight(Node* n)

Update the height of a node as:

height = 1 + max(height(left subtree), height(right subtree))

## 5. Rotations for Balancing

**A. Right** Rotation (LL Case)

Function: rotateRight(Node* y)

Make y->left the new root.

Move y down and adjust child pointers.

Update heights of affected nodes.

Return the new root.

**B.** Left Rotation (RR Case)

Function: rotateLeft(Node* x)

Make x->right the new root.

Move x down and adjust child pointers.

Update heights of affected nodes.

Return the new root.

**C.** Left-Right Rotation (LR Case)

**Function:**

1. Perform a Left Rotation on root->left.

2. Perform a Right Rotation on root.

D. Right-Left Rotation (RL Case)

**Function:**

1. Perform a Right Rotation on root->right.

2. Perform a Left Rotation on root.

## 6. Insert a Node

**Function:** insert(Node* root, int key)

1. If the tree is empty, create a new node.

2. Recursively insert the node into the left or right subtree based on value comparison.

3. Update the height of the current node.

4. Calculate the balance factor to check for rotation cases:

**LL Case:** Perform Right Rotation.

**RR Case:** Perform Left Rotation.

**LR Case:** Perform Left-Right Rotation.

**RL Case:** Perform Right-Left Rotation.

5. Return the updated root.

## 7. Delete a Node

**Function:** remove(Node* root, int key)

1. Find the node to be deleted:

If the key is smaller, move to the left.

If the key is larger, move to the right.

If found:

Case 1: Node has no child → Delete it.

Case 2: Node has one child → Replace it with its child.

Case 3: Node has two children →

Find the minimum node in the right subtree.

Replace the key with this minimum value.

Recursively delete the minimum node.

2. Update height and balance:

If balance > 1, check LL or LR case.

If balance < -1, check RR or RL case.

Perform necessary rotations.

3. Return the updated root

## 8. Find Minimum Node

Function: findMin(Node* root)

Traverse to the leftmost node.

Return that node.

## 9. Inorder Traversal

Function: inorder(Node* root)

Recursively visit:

1. Left subtree

2. Print the node value

3. Right subtree

## 10. Main Function Execution

Insertion Sequence:

1. Insert 10, 20, 30, 40, 50, 25.

2. The AVL tree self-balances using rotations where needed.

Deletion Operation:

1. Delete 30.

2. The AVL tree rebalances.

# Code:

```
#include <iostream>

using namespace std;

// Node structure

struct Node {

    int key, height;

    Node* left;

    Node* right;

    Node(int k) : key(k), height(1), left(nullptr), right(nullptr) {}

};

// Get height of a node
```

```
int getHeight(Node* n) {
    return n ? n->height : 0;
}
// Get balance factor
int getBalance(Node* n) {
    return n ? getHeight(n->left) - getHeight(n->right) : 0;
}
// Update height of a node
void updateHeight(Node* n) {
    if (n) n->height = 1 + max(getHeight(n->left), getHeight(n->right));
}
// Right rotate
Node* rotateRight(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;
    x->right = y;
    y->left = T2;
    updateHeight(y);
    updateHeight(x);
    return x;
}
// Left rotate
Node* rotateLeft(Node* x) {
```

```
    Node* y = x->right;
    Node* T2 = y->left;
    y->left = x;
    x->right = T2;
    updateHeight(x);
    updateHeight(y);
    return y;
}
// Insert node
Node* insert(Node* root, int key) {
    if (!root) return new Node(key);
    if (key < root->key) root->left = insert(root->left, key);
    else if (key > root->key) root->right = insert(root->right, key);
    else return root; // No duplicates allowed
     updateHeight(root);
    int balance = getBalance(root);
// Left Heavy
    if (balance > 1 && key < root->left->key) return rotateRight(root);
    // Right Heavy
    if (balance < -1 && key > root->right->key) return rotateLeft(root);
    // Left-Right Case
    if (balance > 1 && key > root->left->key) {
        root->left = rotateLeft(root->left);
```

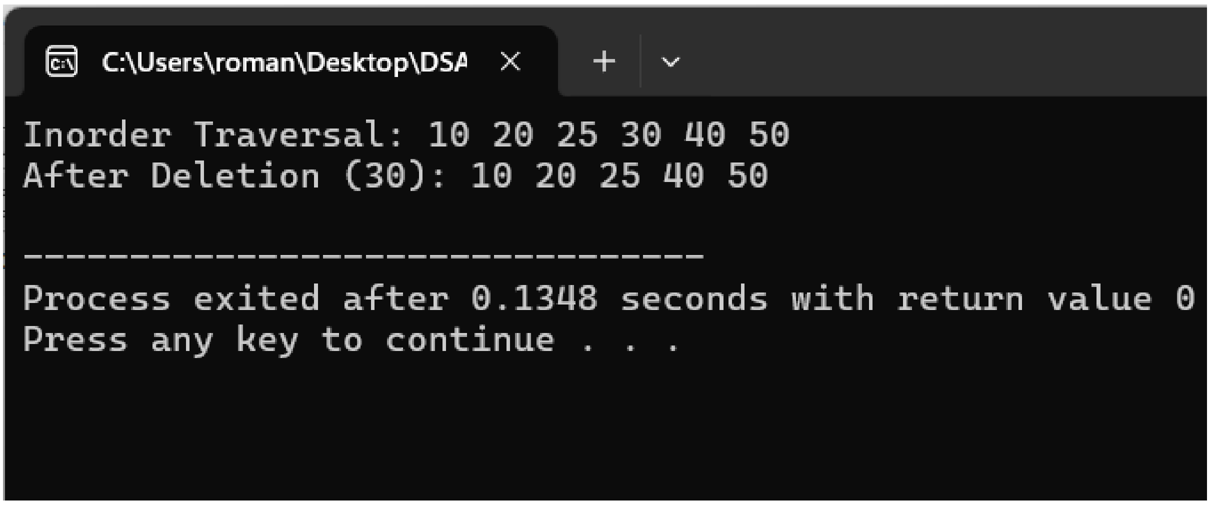
```
        return rotateRight(root);
    }
    // Right-Left Case
    if (balance < -1 && key < root->right->key) {
        root->right = rotateRight(root->right);
        return rotateLeft(root);
    }
 return root;
}
// Find min node
Node* findMin(Node* root) {
    while (root->left) root = root->left;
    return root;
}
// Delete node
Node* remove(Node* root, int key) {
    if (!root) return root;
    if (key < root->key) root->left = remove(root->left, key);
    else if (key > root->key) root->right = remove(root->right, key);
    else {
        if (!root->left || !root->right) {
            Node* temp = root->left ? root->left : root->right;
            delete root;
```

```
            return temp;
        }
        Node* temp = findMin(root->right);
        root->key = temp->key;
        root->right = remove(root->right, temp->key);
    }
 if (!root) return root;
    updateHeight(root);
    int balance = getBalance(root);
 // Balancing cases
    if (balance > 1 && getBalance(root->left) >= 0) return rotateRight(root);
    if (balance > 1 && getBalance(root->left) < 0) {
        root->left = rotateLeft(root->left);
        return rotateRight(root);
    }
    if (balance < -1 && getBalance(root->right) <= 0) return rotateLeft(root);
    if (balance < -1 && getBalance(root->right) > 0) {
        root->right = rotateRight(root->right);
        return rotateLeft(root);
    }
return root;
}
// Inorder Traversal
```

```
void inorder(Node* root) {
    if (root) {
        inorder(root->left);
        cout << root->key << " ";
        inorder(root->right);
    }
}
// Main function
int main() {
    Node* root = nullptr;
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);
cout << "Inorder Traversal: ";
    inorder(root);
    cout << endl;
root = remove(root, 30);
    cout << "After Deletion (30): ";
    inorder(root);
    cout << endl;
```

```
    return 0;
}
```

# OUTPUT:




# Explanation of AVL Tree Code

## Objective:

To implement an AVL Tree with insertion, deletion, and balancing rotations.

To maintain a self-balancing Binary Search Tree (BST) where the height difference of left and right subtrees is at most 1.

To ensure efficient searching, insertion, and deletion with O(log N) complexity.

## Step 1: Node Structure

```
struct Node {
    int key, height;
    Node* left;
```

```
    Node* right;
    Node(int k) : key(k), height(1), left(nullptr), right(nullptr) {}
};
```

# Explanation:

Each node has:

key: Stores the data.

height: Stores the height of the node for balance factor calculations.

left: Pointer to the left child.

right: Pointer to the right child.

A constructor initializes these values when a new node is created.

# Step 2: Get Height of a Node

```
int getHeight(Node* n) {
    return n ? n->height : 0;
}
```

# Explanation:

If the node exists, return its height.

If nullptr, return 0 (base case for empty nodes).

# Step 3: Get Balance Factor

```
int getBalance(Node* n) {
    return n ? getHeight(n->left) - getHeight(n->right) : 0;
}
```

## Explanation:

Balance Factor = Height of Left Subtree - Height of Right Subtree

If Balance Factor:

>1 → Left-heavy (needs right rotation)

<-1 → Right-heavy (needs left rotation)

-1, 0, 1 → Balanced, no rotation needed

## Step 4: Update Node Height

```
void updateHeight(Node* n) {

    if (n) n->height = 1 + max(getHeight(n->left), getHeight(n->right));

}
```

## Explanation:

Height of a node is 1 + max(left subtree height, right subtree height).

This ensures height values remain updated after insertions or deletions.

## Step 5: Rotations for Balancing

Right Rotation (LL Case)

```
Node* rotateRight(Node* y) {

    Node* x = y->left;

    Node* T2 = x->right;

    x->right = y;

    y->left = T2;

    updateHeight(y);

    updateHeight(x);
```

```
    return x;
}
```

# Explanation:

Used when tree is Left-heavy (LL case).

Moves y down and x up.

T2 is repositioned to maintain BST properties.

Updates heights after rotation.

# Left Rotation (RR Case)

```
Node* rotateLeft(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;
    y->left = x;
    x->right = T2;
    updateHeight(x);
    updateHeight(y);
    return y;
}
```

# Explanation:

Used when tree is Right-heavy (RR case).

Moves x down and y up.

T2 is repositioned to maintain BST properties.

Updates heights after rotation.

# Step 6: Insert a Node

```
Node* insert(Node* root, int key) {
    if (!root) return new Node(key);
    if (key < root->key) root->left = insert(root->left, key);
    else if (key > root->key) root->right = insert(root->right, key);
    else return root; // No duplicates allowed
    updateHeight(root);
    int balance = getBalance(root);
   // Left Heavy
    if (balance > 1 && key < root->left->key) return rotateRight(root);
    // Right Heavy
    if (balance < -1 && key > root->right->key) return rotateLeft(root);
    // Left-Right Case
    if (balance > 1 && key > root->left->key) {
        root->left = rotateLeft(root->left);
        return rotateRight(root);
    }
    // Right-Left Case
    if (balance < -1 && key < root->right->key) {
        root->right = rotateRight(root->right);
        return rotateLeft(root);
    }
```

```
    return root;
}
```

# Explanation:

1. If the tree is empty, create a new node.

2. Recursively insert into left or right subtree based on value.

3. Update the height of the node.

4. Check balance factor:

LL Case → Right Rotation

RR Case → Left Rotation

LR Case → Left Rotation + Right Rotation

RL Case → Right Rotation + Left Rotation

5. Return the balanced root.

# Step 7: Find Minimum Node

```
Node* findMin(Node* root) {
    while (root->left) root = root->left;
    return root;
}
```

# Explanation:

Finds the leftmost node, which is the minimum value in a BST.

# Step 8: Delete a Node

```
Node* remove(Node* root, int key) {
    if (!root) return root;
```

```
if (key < root->key) root->left = remove(root->left, key);
else if (key > root->key) root->right = remove(root->right, key);
else {
    if (!root->left || !root->right) {
        Node* temp = root->left ? root->left : root->right;
        delete root;
        return temp;
    }
    Node* temp = findMin(root->right);
    root->key = temp->key;
    root->right = remove(root->right, temp->key);
}
if (!root) return root;
updateHeight(root);
int balance = getBalance(root);
// Balancing cases
if (balance > 1 && getBalance(root->left) >= 0) return rotateRight(root);
if (balance > 1 && getBalance(root->left) < 0) {
    root->left = rotateLeft(root->left);
    return rotateRight(root);
}
if (balance < -1 && getBalance(root->right) <= 0) return rotateLeft(root);
if (balance < -1 && getBalance(root->right) > 0) {
```

```
        root->right = rotateRight(root->right);
        return rotateLeft(root);
    }
  return root;
}
```

# Explanation:

1. Find the node to delete.

2. Handle cases:

No child: Delete directly.

One child: Replace with child.

Two children: Replace with minimum node from right subtree.

3. Update height and balance.

4. Perform necessary rotations.

# Step 9: Inorder Traversal

```
void inorder(Node* root) {
    if (root) {
        inorder(root->left);
        cout << root->key << " ";
        inorder(root->right);
    }
}
```

# Explanation:

Prints nodes in sorted order (Left → Root → Right).

## Step 10: Main Function Execution

```
int main() {
   Node* root = nullptr;
   root = insert(root, 10);
   root = insert(root, 20);
   root = insert(root, 30);
   root = insert(root, 40);
   root = insert(root, 50);
   root = insert(root, 25);
   cout << "Inorder Traversal: ";
   inorder(root);
   cout << endl;
   root = remove(root, 30);
   cout << "After Deletion (30): ";
   inorder(root);
   cout << endl;
  return 0;
}
```

## Explanation:

1. Inserts values 10, 20, 30, 40, 50, 25.

2. Performs balancing operations if needed.

3. Displays inorder traversal (sorted order).

4. Deletes node 30 and prints the tree again.