Data Structure Algorithms & Applications (CT-159)
Lab 10

AVL Tree

Objectives

The objective of the lab is to get students familiar with AVL Tree and rotations.

Tools Required

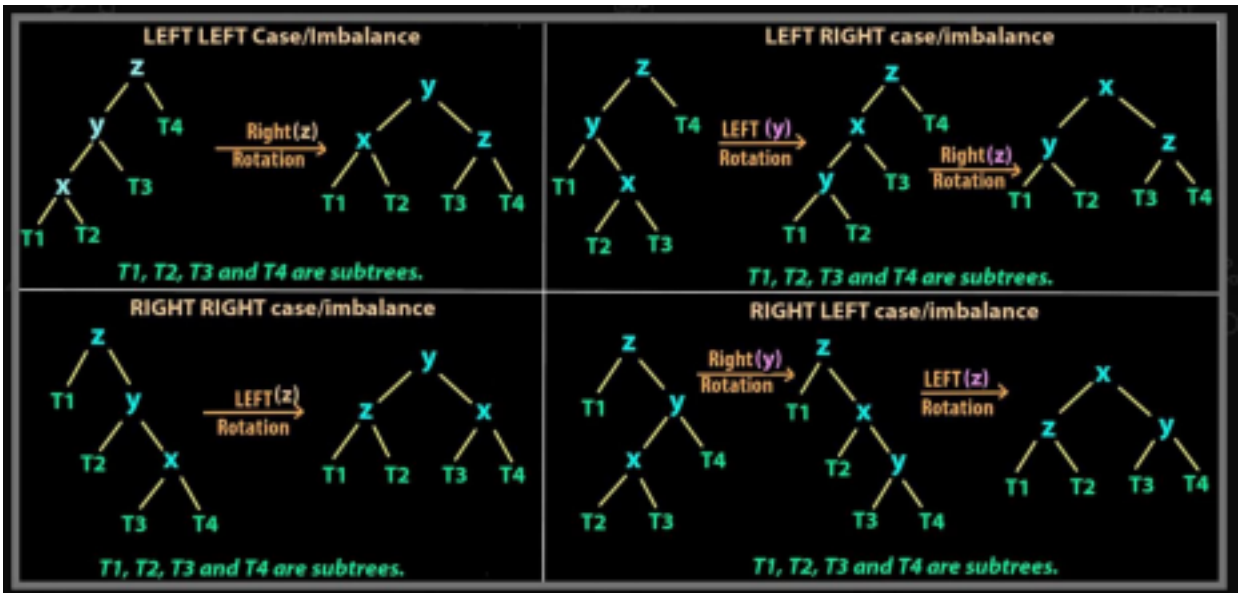
Dev C++ IDE

Course Coordinator –
Course Instructor –
Lab Instructor –
Prepared By Department of Computer Science and Information Technology NED University of Engineering and Technology

AVL Tree

AVL trees (named after their inventors Adelson-Velsky and Landis) are **self-balancing binary search trees**that maintain a balance factor for each node to ensure the tree remains balanced. They provide guaranteed O(log n) time complexity for search, insertion, and deletion operations. Here's how they solve the problems in BSTs:

- **Height-Balanced Property**: In an AVL tree, the difference in the height of the left and right subtrees for any node (called the **balance factor**) is either -1, 0, or 1. Balance factor = Height of the left subtree − Height of the right subtree. Whenever an insertion or deletion operation leads to an imbalance, **rotations** (left, right, or a combination) are performed to restore balance.
- **Self-Balancing**: After every insertion or deletion, the tree checks if it is still balanced by calculating the balance factor for each node. If the balance factor is violated, the tree performs **rotations** to re-balance itself. This prevents the tree from becoming too skewed and ensures that the height of the tree is kept at O(log n).
- **Rotations**: AVL trees use **four types of rotations** to maintain balance: **Right Rotation (Single)**: Performed when a node becomes unbalanced due to too much height on its left subtree (left-left case). **Left Rotation (Single)**: Performed when a node becomes unbalanced due to too much height on its right subtree (right-right case). **Left-Right Rotation (Double)**: Performed when a node's left child is right-heavy (left-right case). **Right-Left Rotation (Double)**: Performed when a node's right child is left-heavy (right-left case). These rotations effectively reduce the height of the tree and restore balance.



Steps for Inserting a Node in an AVL tree

1. Perform standard BST insertion.
2. Update the height of each node.
3. Calculate the balance factor for each node.
4. If unbalanced, apply the appropriate rotation(s).

Steps for Deleting a Node from an AVL tree

1. Perform Standard BST Deletion.
2. Update the height for each node.
3. Check Balance Factor for each node.
4. Perform Rotations (if needed). If the tree is unbalanced, apply one of the four rotations: LL, RR, LR, or RL to restore balance.
5. Repeat Until the Tree is Fully Balanced. Ensure that every node along the path from the deleted node up to the root is balanced. Multiple rotations might be needed.

Example 01: Implementation of an AVL tree.

```
#include <iostream>
using namespace std;

class AVLTree {
private:
 struct Node {
 int key;
 Node* left;
 Node* right;
 int height;

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

 Node* root;
```



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

 int balanceFactor(Node* node) {
 return node ? height(node->left) - height(node->right) : 0;  }

 void updateHeight(Node* node) {
 node->height = 1 + max(height(node->left), height(node->right));  }

 Node* rotateRight(Node* y) {
 Node* x = y->left;
 Node* T2 = x->right;
```

```
x->right = y;
y->left = T2;

updateHeight(y);
updateHeight(x);

return x;
}

Node* rotateLeft(Node* x) {
Node* y = x->right;
Node* T2 = y->left;

y->left = x;
x->right = T2;

updateHeight(x);
updateHeight(y);

return y;
}

Node* balance(Node* node) {
updateHeight(node);

int balance = balanceFactor(node);

// Left heavy
if (balance > 1) {
if (balanceFactor(node->left) < 0) // Left-Right case
node->left = rotateLeft(node->left);
return rotateRight(node); // Left-Left case
}

// Right heavy
if (balance < -1) {
if (balanceFactor(node->right) > 0) // Right-Left case
node->right = rotateRight(node->right);
return rotateLeft(node); // Right-Right case
}

return node; // Balanced
}

Node* minValueNode(Node* node) {
Node* current = node;
while (current && current->left != nullptr)
current = current->left;
return current;
}

Node* insert(Node* node, int key) {
```

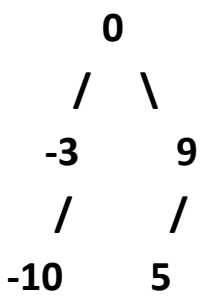


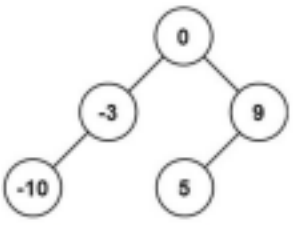
```
//recursive insertion in a bst
// implement the code

return balance(node);
}

Node* deleteNode(Node* root, int key) {
// Perform standard BST delete
// Complete the code

// Balance the node
return balance(root);
}
```

```
void inOrder(Node* node) {
if (!node) return;
inOrder(node->left);
cout << node->key << " ";
inOrder(node->right);
}

public:
AVLTree() : root(nullptr) {}

void insert(int key) {
root = insert(root, key);
}

void deleteKey(int key) {
root = deleteNode(root, key);
}

void inOrder() {
inOrder(root);
cout << endl;
}
};

int main() {
AVLTree tree;
tree.insert(10);
tree.insert(20);
tree.insert(30);
tree.insert(20); // Duplicate insertion
tree.insert(40);
tree.insert(50);
tree.insert(25);

cout << "In-order traversal before deletion: ";
tree.inOrder();

tree.deleteKey(20);
cout << "In-order traversal after deleting 20: ";
tree.inOrder();

return 0;
}
```

Exercise

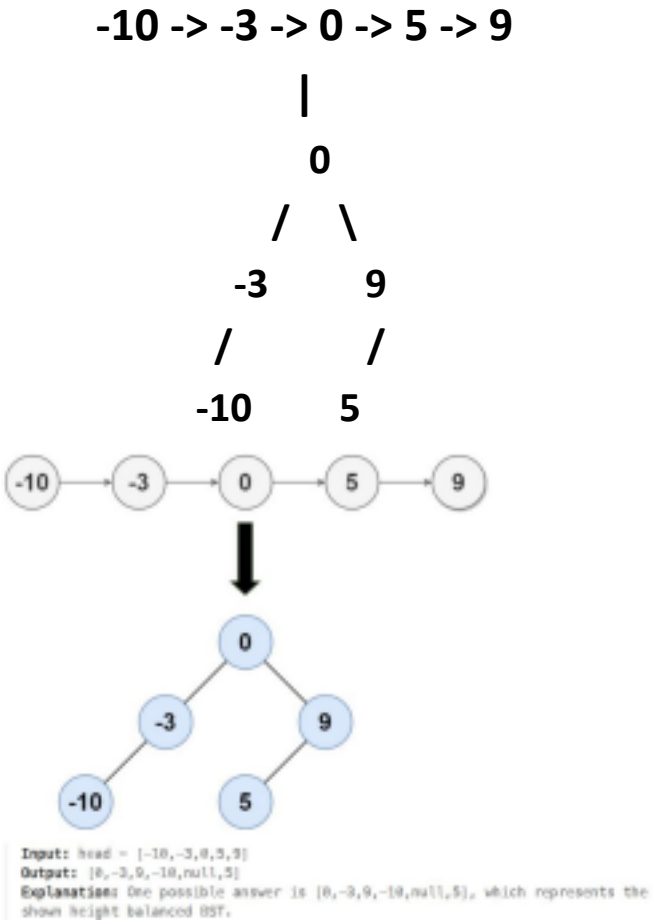
1. Complete implementation of AVL tree given in Example 01 by implementing Insert and Delete methods.
2. Given an integer array nums where the elements are sorted in ascending order, convert it to a height-balanced binary search tree.






```
Input: nums = [-10,-3,0,5,9]
Output: [0,-3,9,-10,null,5]
Explanation: [0,-10,5,null,-3,null,9] is also accepted:
```

3. Given the head of a singly linked list where elements are sorted in ascending order, convert it to a height-balanced binary search tree.




```
Input: head = [-10,-3,0,5,9]
Output: [0,-3,9,-10,null,5]
Explanation: One possible answer is [0,-3,9,-10,null,5], which represents the
shown height balanced BST.
```

4. Given the root of a BST, find its diameter.

Lab 10 Evaluation		
Student Name: Student ID: Date:		
Rubric	**Marks (25)**	**Remarks by teacher in accordance with the rubrics**
R1		
R2		
R3		
R4		
R5		