

|  |
| --- |
| **LAB 11** of DSA LAB |

**Submitted by:**

**Name:** Muhammad shuraim

**Roll no:** SU92-S24-BSSEM-103

**Section:** 3A

**Submitted to**

Sir Rasikh Ali

**Question 1**

**AVL(INSERT AND TRANSVERSE)**

#include <iostream>

using namespace std;

struct TreeNode {

int value;

TreeNode\* leftChild;

TreeNode\* rightChild;

TreeNode(int data) {

value = data;

leftChild = rightChild = nullptr;

}

};

TreeNode\* insertNode(TreeNode\* currentRoot, int data) {

if (currentRoot == nullptr)

return new TreeNode(data);

if (data < currentRoot->value)

currentRoot->leftChild = insertNode(currentRoot->leftChild, data);

else

currentRoot->rightChild = insertNode(currentRoot->rightChild, data);

return currentRoot;

}

// Inorder traversal (Left, Root, Right)

void traverseInorder(TreeNode\* currentNode) {

if (currentNode != nullptr) {

traverseInorder(currentNode->leftChild);

cout << currentNode->value << " ";

traverseInorder(currentNode->rightChild);

}

}

// Preorder traversal (Root, Left, Right)

void traversePreorder(TreeNode\* currentNode) {

if (currentNode != nullptr) {

cout << currentNode->value << " ";

traversePreorder(currentNode->leftChild);

traversePreorder(currentNode->rightChild);

}

}

// Postorder traversal (Left, Right, Root)

void traversePostorder(TreeNode\* currentNode) {

if (currentNode != nullptr) {

traversePostorder(currentNode->leftChild);

traversePostorder(currentNode->rightChild);

cout << currentNode->value << " ";

}

}

int main() {

TreeNode\* bstRoot = nullptr;

int inputValues[] = { 50, 30, 70, 20, 40, 60, 80 };

for (int item : inputValues) {

bstRoot = insertNode(bstRoot, item);

}

cout << "Inorder traversal: ";

traverseInorder(bstRoot);

cout << "\n";

cout << "Preorder traversal: ";

traversePreorder(bstRoot);

cout << "\n";

cout << "Postorder traversal: ";

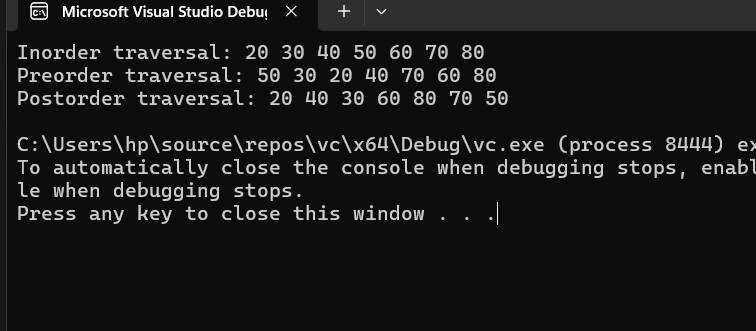
traversePostorder(bstRoot);

cout << "\n";

return 0;

}

**OUTPUT**

****

**AVL INSERT AND TRANSVERSE**

#include <iostream>

using namespace std;

struct AVLNode {

int key;

AVLNode\* leftChild;

AVLNode\* rightChild;

int nodeHeight;

AVLNode(int value) {

key = value;

leftChild = rightChild = nullptr;

nodeHeight = 1;

}

};

int getHeight(AVLNode\* currentNode) {

if (currentNode == nullptr)

return 0;

return currentNode->nodeHeight;

}

int getBalanceFactor(AVLNode\* currentNode) {

if (currentNode == nullptr)

return 0;

return getHeight(currentNode->leftChild) - getHeight(currentNode->rightChild);

}

void updateNodeHeight(AVLNode\* currentNode) {

currentNode->nodeHeight = 1 + max(getHeight(currentNode->leftChild), getHeight(currentNode->rightChild));

}

AVLNode\* performRightRotation(AVLNode\* yNode) {

AVLNode\* xNode = yNode->leftChild;

AVLNode\* subtree = xNode->rightChild;

xNode->rightChild = yNode;

yNode->leftChild = subtree;

updateNodeHeight(yNode);

updateNodeHeight(xNode);

return xNode;

}

AVLNode\* performLeftRotation(AVLNode\* xNode) {

AVLNode\* yNode = xNode->rightChild;

AVLNode\* subtree = yNode->leftChild;

yNode->leftChild = xNode;

xNode->rightChild = subtree;

updateNodeHeight(xNode);

updateNodeHeight(yNode);

return yNode;

}

AVLNode\* insertIntoAVL(AVLNode\* currentRoot, int newValue) {

if (currentRoot == nullptr)

return new AVLNode(newValue);

if (newValue < currentRoot->key)

currentRoot->leftChild = insertIntoAVL(currentRoot->leftChild, newValue);

else if (newValue > currentRoot->key)

currentRoot->rightChild = insertIntoAVL(currentRoot->rightChild, newValue);

else

return currentRoot;

updateNodeHeight(currentRoot);

int balance = getBalanceFactor(currentRoot);

// Left Left Case

if (balance > 1 && newValue < currentRoot->leftChild->key)

return performRightRotation(currentRoot);

// Right Right Case

if (balance < -1 && newValue > currentRoot->rightChild->key)

return performLeftRotation(currentRoot);

// Left Right Case

if (balance > 1 && newValue > currentRoot->leftChild->key) {

currentRoot->leftChild = performLeftRotation(currentRoot->leftChild);

return performRightRotation(currentRoot);

}

// Right Left Case

if (balance < -1 && newValue < currentRoot->rightChild->key) {

currentRoot->rightChild = performRightRotation(currentRoot->rightChild);

return performLeftRotation(currentRoot);

}

return currentRoot;

}

// Inorder traversal

void printInorder(AVLNode\* rootNode) {

if (rootNode != nullptr) {

printInorder(rootNode->leftChild);

cout << rootNode->key << " ";

printInorder(rootNode->rightChild);

}

}

// Preorder traversal

void printPreorder(AVLNode\* rootNode) {

if (rootNode != nullptr) {

cout << rootNode->key << " ";

printPreorder(rootNode->leftChild);

printPreorder(rootNode->rightChild);

}

}

// Postorder traversal

void printPostorder(AVLNode\* rootNode) {

if (rootNode != nullptr) {

printPostorder(rootNode->leftChild);

printPostorder(rootNode->rightChild);

cout << rootNode->key << " ";

}

}

int main() {

AVLNode\* avlRoot = nullptr;

int elements[] = { 10, 20, 30, 40, 50, 25 };

for (int element : elements) {

avlRoot = insertIntoAVL(avlRoot, element);

}

cout << "Inorder traversal: ";

printInorder(avlRoot);

cout << "\n";

cout << "Preorder traversal: ";

printPreorder(avlRoot);

cout << "\n";

cout << "Postorder traversal: ";

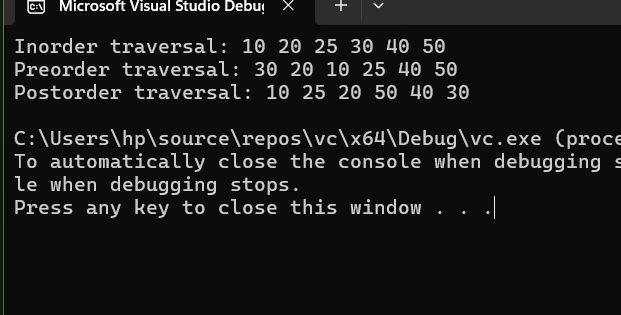
printPostorder(avlRoot);

cout << "\n";

return 0;

}

Output



EXPLANATION OF BOTH

**Binary Search Tree (BST)**

**✅ What is BST?**

A **Binary Search Tree** is a tree data structure where:

* Each node contains a value.
* Left child contains values **less than** the parent.
* Right child contains values **greater than** the parent.

**✅ Insertion in BST**

* Start from the root.
* If the value is smaller, go to the left subtree.
* If greater, go to the right.
* Repeat until the correct position is found and insert the node.

**✅ Traversal in BST**

1. **Inorder** (Left → Root → Right) — gives sorted output.
2. **Preorder** (Root → Left → Right) — used to copy the tree.
3. **Postorder** (Left → Right → Root) — used to delete the tree.

**🟩 AVL Tree (Adelson-Velsky and Landis Tree)**

**✅ What is AVL Tree?**

An **AVL Tree** is a **self-balancing BST** where the **balance factor** (difference of heights between left and right subtrees) of every node is **-1, 0, or +1**.

**✅ Insertion in AVL**

* Insert like a BST.
* After insertion, check balance factor.
* If unbalanced, apply **rotations** to restore balance:
  1. **LL Rotation** (Right Rotate)
  2. **RR Rotation** (Left Rotate)
  3. **LR Rotation** (Left-Right Rotate)
  4. **RL Rotation** (Right-Left Rotate)

**✅ Traversal in AVL**

Same as BST:

1. **Inorder**
2. **Preorder**
3. **Postorder**