**Raahat Arora (Week 5)**

**230957216**

**Roll no – 72**

**Q1-**

#include <iostream>

using namespace std;

// Definition for a binary tree node

struct TreeNode {

int val;

TreeNode\* left;

TreeNode\* right;

TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

};

// Function to insert a node in the BST

TreeNode\* insert(TreeNode\* root, int val) {

if (root == nullptr) {

return new TreeNode(val);

}

if (val < root->val) {

root->left = insert(root->left, val);

} else {

root->right = insert(root->right, val);

}

return root;

}

// Function to calculate the height of the BST

int height(TreeNode\* root) {

// Base case: If the tree is empty, its height is -1

if (root == nullptr) {

return -1;

}

// Recursively find the height of left and right subtrees

int leftHeight = height(root->left);

int rightHeight = height(root->right);

// The height of the current tree is the max of left and right subtree heights plus 1

return max(leftHeight, rightHeight) + 1;

}

int main() {

// Create an empty binary search tree

TreeNode\* root = nullptr;

// Insert nodes into the BST

root = insert(root, 50);

root = insert(root, 30);

root = insert(root, 20);

root = insert(root, 40);

root = insert(root, 70);

root = insert(root, 60);

root = insert(root, 80);

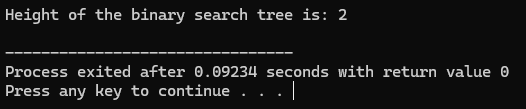
// Calculate and print the height of the BST

cout << "Height of the binary search tree is: " << height(root) << endl;

return 0;

}

**Output-**

****

**Q2-**

#include <iostream>

#include <queue>

using namespace std;

struct TreeNode {

int val;

TreeNode\* left;

TreeNode\* right;

TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

};

TreeNode\* insert(TreeNode\* root, int val) {

if (root == nullptr) {

return new TreeNode(val);

}

if (val < root->val) {

root->left = insert(root->left, val);

} else {

root->right = insert(root->right, val);

}

return root;

}

// In-Order Traversal (Left, Root, Right)

void inorder(TreeNode\* root) {

if (root == nullptr) return;

inorder(root->left);

cout << root->val << " ";

inorder(root->right);

}

// Pre-Order Traversal (Root, Left, Right)

void preorder(TreeNode\* root) {

if (root == nullptr) return;

cout << root->val << " ";

preorder(root->left);

preorder(root->right);

}

// Post-Order Traversal (Left, Right, Root)

void postorder(TreeNode\* root) {

if (root == nullptr) return;

postorder(root->left);

postorder(root->right);

cout << root->val << " ";

}

// Level-Order Traversal (Breadth-First Traversal)

void levelOrder(TreeNode\* root) {

if (root == nullptr) return;

queue<TreeNode\*> q;

q.push(root);

while (!q.empty()) {

TreeNode\* node = q.front();

cout << node->val << " ";

q.pop();

if (node->left) q.push(node->left);

if (node->right) q.push(node->right);

}

}

// Function to print the tree elements

void printTree(TreeNode\* root) {

cout << "In-Order Traversal: ";

inorder(root);

cout << endl;

cout << "Pre-Order Traversal: ";

preorder(root);

cout << endl;

cout << "Post-Order Traversal: ";

postorder(root);

cout << endl;

cout << "Level-Order Traversal: ";

levelOrder(root);

cout << endl;

}

int main() {

// Create an empty binary search tree

TreeNode\* root = nullptr;

// Insert nodes into the BST

root = insert(root, 50);

root = insert(root, 30);

root = insert(root, 20);

root = insert(root, 40);

root = insert(root, 70);

root = insert(root, 60);

root = insert(root, 80);

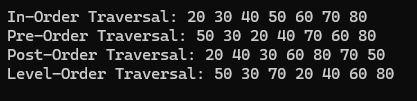
// Print the tree using all traversal methods

printTree(root);

return 0;

}

**Output-**

****

**Q3-**

#include <iostream>

#include <algorithm>

using namespace std;

// Definition for a binary tree node

struct TreeNode {

int val;

TreeNode\* left;

TreeNode\* right;

int height;

TreeNode(int x) : val(x), left(nullptr), right(nullptr), height(1) {}

};

// Function to get the height of a node

int height(TreeNode\* node) {

if (node == nullptr) return 0;

return node->height;

}

// Function to get the balance factor of a node

int getBalance(TreeNode\* node) {

if (node == nullptr) return 0;

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

}

// Right Rotation

TreeNode\* rightRotate(TreeNode\* y) {

TreeNode\* x = y->left;

TreeNode\* T2 = x->right;

// Perform rotation

x->right = y;

y->left = T2;

// Update heights

y->height = max(height(y->left), height(y->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

return x; // Return the new root

}

// Left Rotation

TreeNode\* leftRotate(TreeNode\* x) {

TreeNode\* y = x->right;

TreeNode\* T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

// Update heights

x->height = max(height(x->left), height(x->right)) + 1;

y->height = max(height(y->left), height(y->right)) + 1;

return y; // Return the new root

}

// Function to insert a node into the AVL tree

TreeNode\* insert(TreeNode\* root, int val) {

// 1. Perform the normal BST insertion

if (root == nullptr) {

return new TreeNode(val);

}

if (val < root->val) {

root->left = insert(root->left, val);

} else if (val > root->val) {

root->right = insert(root->right, val);

} else { // Duplicate values are not allowed in BST

return root;

}

// 2. Update the height of this ancestor node

root->height = max(height(root->left), height(root->right)) + 1;

// 3. Get the balance factor to check whether this node became unbalanced

int balance = getBalance(root);

// 4. If the node becomes unbalanced, then there are 4 cases

// Left Left Case

if (balance > 1 && val < root->left->val) {

return rightRotate(root);

}

// Right Right Case

if (balance < -1 && val > root->right->val) {

return leftRotate(root);

}

// Left Right Case

if (balance > 1 && val > root->left->val) {

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

return rightRotate(root);

}

// Right Left Case

if (balance < -1 && val < root->right->val) {

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

return leftRotate(root);

}

// Return the (unchanged) node pointer

return root;

}

// Function to perform in-order traversal and print the tree

void inorder(TreeNode\* root) {

if (root == nullptr) return;

inorder(root->left);

cout << root->val << " ";

inorder(root->right);

}

int main() {

TreeNode\* root = nullptr;

// Inserting 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);

cout << "In-order traversal of the AVL tree: ";

inorder(root);

cout << endl;

// Insert element 6 into the AVL tree

root = insert(root, 6);

cout << "In-order traversal after inserting 6: ";

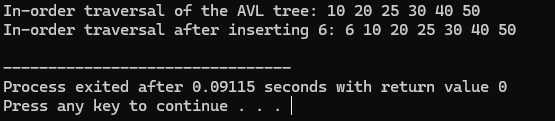
inorder(root);

cout << endl;

return 0;

}

**Output-**



**Q4-**

#include <iostream>  
using namespace std;  
// Define a structure for a 2-3 Tree Node  
struct Node {  
    int keys[2];  // A node can store up to 2 keys  
    Node\* children[3];  // A node can have up to 3 children  
    int numKeys;  // Number of keys in the node (1 or 2)  
    Node() {  
        numKeys = 0;  
        for (int i = 0; i < 3; i++) {  
            children[i] = NULL;  
        }  
    }  
};  
// 2-3 Tree class  
class TwoThreeTree {  
private:  
    Node\* root;  
    // Helper function to split a node if it's full (2 keys)  
    void splitNode(Node\* parent, int index, Node\* child) {  
        Node\* newChild = new Node();  
        int midKey = child->keys[1];  
        newChild->keys[0] = midKey;  
        newChild->children[0] = child->children[1];  
        newChild->children[1] = child->children[2];  
        child->numKeys = 1;  
        parent->children[index] = child;  
        parent->keys[index] = midKey;  
        parent->children[index + 1] = newChild;  
        parent->numKeys++;  
    }  
    // Insert a key into a non-full node  
    void insertNonFull(Node\* node, int key) {  
        int i = node->numKeys - 1;  
        // If it's a leaf node, insert the key into the correct position  
        if (node->children[0] == NULL) {  
            while (i >= 0 && key < node->keys[i]) {  
                node->keys[i + 1] = node->keys[i];  
                i--;  
            }  
            node->keys[i + 1] = key;  
            node->numKeys++;  
        } else {  
            // If it's not a leaf node, find the appropriate child to insert the key  
            while (i >= 0 && key < node->keys[i]) {  
                i--;  
            }  
            // Check if the child is full and needs to be split  
            if (node->children[i + 1]->numKeys == 2) {  
                splitNode(node, i + 1, node->children[i + 1]);  
                if (key > node->keys[i + 1]) {  
                    i++;  
                }  
            }  
            insertNonFull(node->children[i + 1], key);  
        }  
    }  
    // Function to recursively print the tree (in-order traversal)  
    void inorderTraversal(Node\* node) {  
        if (node == NULL) return;  
        for (int i = 0; i < node->numKeys; i++) {  
            inorderTraversal(node->children[i]);  
            cout << node->keys[i] << " ";  
        }  
        inorderTraversal(node->children[node->numKeys]);  
    }  
public:  
    // Constructor to initialize the tree  
    TwoThreeTree() {  
        root = new Node();  
    }  
    // Insert a key into the 2-3 tree  
    void insert(int key) {  
        Node\* rootNode = root;  
        if (rootNode->numKeys == 2) {  
            Node\* newRoot = new Node();  
            newRoot->children[0] = rootNode;  
            splitNode(newRoot, 0, rootNode);  
            root = newRoot;  
        }  
        insertNonFull(root, key);  
    }  
    // Wrapper function for inorder traversal  
    void printTree() {  
        inorderTraversal(root);  
        cout << endl;  
    }  
};  
int main() {  
    TwoThreeTree tree;  
    // Insert elements into the 2-3 tree  
    tree.insert(10);  
    tree.insert(20);  
    tree.insert(5);  
    tree.insert(15);  
    tree.insert(25);  
    tree.insert(30);  
    // Print the tree before inserting 6  
    cout << "Inorder Traversal before inserting 6: ";  
    tree.printTree();  
    // Insert the element 6  
    tree.insert(6);  
    // Print the tree after inserting 6  
    cout << "Inorder Traversal after inserting 6: ";  
    tree.printTree();  
    return 0;  
}

**Output-**

