National University of Computer and Emerging Sciences



Name: Muhammad Suleman

Roll #: 22F-3350

Section: BCS-4B

Lab # 10

# Problem 01

#include <iostream>

using namespace std;

template <typename T>

struct Node

{

T data;

Node<T>\* left;

Node<T>\* right;

int height;

Node(T value)

: data(value), left(nullptr), right(nullptr), height(0) {}

};

template <typename T>

class AVLTree

{

public:

AVLTree()

: root(nullptr) {}

void insert(T value)

{

root = insertNode(root, value);

}

bool search(T value)

{

return searchNode(root, value);

}

void inorderTraversal()

{

inorderTraversalNode(root);

cout << endl;

}

void preorderTraversal()

{

preorderTraversalNode(root);

cout << endl;

}

void postorderTraversal()

{

postorderTraversalNode(root);

cout << endl;

}

T findMin()

{

Node<T>\* minNode = findMinNode(root);

if (minNode)

{

return minNode->data;

}

return T();

}

T findSecondMin()

{

Node<T>\* secondMinNode = findSecondMinNode(root);

if (secondMinNode)

{

return secondMinNode->data;

}

return T();

}

bool isAVL()

{

return isAVLTree(root);

}

private:

Node<T>\* root;

int getHeight(Node<T>\* node)

{

if (node == nullptr)

{

return -1;

}

return node->height;

}

int getMax(int a, int b)

{

if (a > b)

{

return a;

}

return b;

}

int getBalanceFactor(Node<T>\* node)

{

if (node == nullptr)

{

return -1;

}

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

}

Node<T>\* rotateRight(Node<T>\* y)

{

Node<T>\* x = y->left;

Node<T>\* t2 = x->right;

x->right = y;

y->left = t2;

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

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

return x;

}

Node<T>\* rotateLeft(Node<T>\* x)

{

Node<T>\* y = x->right;

Node<T>\* t2 = y->left;

y->left = x;

x->right = t2;

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

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

return y;

}

Node<T>\* insertNode(Node<T>\* node, T value)

{

if (node == nullptr)

{

return new Node<T>(value);

}

if (value < node->data)

{

node->left = insertNode(node->left, value);

}

else if (value > node->data)

{

node->right = insertNode(node->right, value);

}

else

{

return node;

}

node->height = 1 + getMax(getHeight(node->left), getHeight(node->right));

int balanceFactor = getBalanceFactor(node);

if (balanceFactor > 1 && value < node->left->data)

{

return rotateRight(node);

}

if (balanceFactor < -1 && value > node->right->data)

{

return rotateLeft(node);

}

if (balanceFactor > 1 && value > node->left->data)

{

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

return rotateRight(node);

}

if (balanceFactor < -1 && value < node->right->data)

{

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

return rotateLeft(node);

}

return node;

}

bool isAVLTree(Node<T>\* node)

{

if (node == nullptr)

{

return true;

}

int balanceFactor = getBalanceFactor(node);

if (balanceFactor < -1 || balanceFactor > 1)

{

return false;

}

return isAVLTree(node->left) && isAVLTree(node->right);

}

bool searchNode(Node<T>\* node, T value)

{

if (node == nullptr)

{

return false;

}

if (value < node->data)

{

return searchNode(node->left, value);

}

else if (value > node->data)

{

return searchNode(node->right, value);

}

else

{

return true;

}

}

Node<T>\* findMinNode(Node<T>\* node)

{

if (node == nullptr || node->left == nullptr)

{

return node;

}

return findMinNode(node->left);

}

Node<T>\* findSecondMinNode(Node<T>\* node)

{

if (node == nullptr || (node->left == nullptr && node->right == nullptr))

{

return nullptr;

}

Node<T>\* current = node;

Node<T>\* parent = nullptr;

while (current->left != nullptr)

{

parent = current;

current = current->left;

}

if (current->right != nullptr)

{

return findMinNode(current->right);

}

return parent;

}

void inorderTraversalNode(Node<T>\* node)

{

if (node == nullptr)

{

return;

}

inorderTraversalNode(node->left);

cout << node->data << " ";

inorderTraversalNode(node->right);

}

void preorderTraversalNode(Node<T>\* node)

{

if (node == nullptr)

{

return;

}

cout << node->data << " ";

preorderTraversalNode(node->left);

preorderTraversalNode(node->right);

}

void postorderTraversalNode(Node<T>\* node)

{

if (node == nullptr)

{

return;

}

postorderTraversalNode(node->left);

postorderTraversalNode(node->right);

cout << node->data << " ";

}

};

int main()

{

AVLTree<int> avlTree;

for (int i = 1; i <= 15; i++)

{

avlTree.insert(i);

}

cout << "Inorder traversal: ";

avlTree.inorderTraversal();

cout << "Preorder traversal: ";

avlTree.preorderTraversal();

cout << "Postorder traversal: ";

avlTree.postorderTraversal();

cout << "Searching for 25: ";

if (avlTree.search(25))

{

cout << "Found" << endl;

}

else

{

cout << "Not found" << endl;

}

cout << "Searching for 1: ";

if (avlTree.search(1))

{

cout << "Found" << endl;

}

else

{

cout << "Not found" << endl;

}

cout << "Minimum element: " << avlTree.findMin() << endl;

cout << "Second minimum element: " << avlTree.findSecondMin() << endl;

cout << "IsAVL tree: ";

if (avlTree.isAVL())

{

cout << "AVL tree" << endl;

}

else

{

cout << "Not AVL tree" << endl;

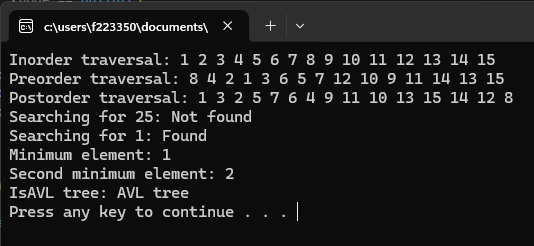
}

system("pause");

return 0;

}

## Output



# Problem 02

#include <iostream>

using namespace std;

template <typename T>

struct Node

{

T data;

Node<T>\* left;

Node<T>\* right;

int height;

Node(T value)

: data(value), left(nullptr), right(nullptr), height(0) {}

};

template <typename T>

class AVLTree

{

public:

AVLTree()

: root(nullptr) {}

void insert(T value)

{

root = insertNode(root, value);

}

bool search(T value)

{

return searchNode(root, value);

}

void inorderTraversal()

{

inorderTraversalNode(root);

cout << endl;

}

void preorderTraversal()

{

preorderTraversalNode(root);

cout << endl;

}

void postorderTraversal()

{

postorderTraversalNode(root);

cout << endl;

}

T findMin()

{

Node<T>\* minNode = findMinNode(root);

if (minNode)

{

return minNode->data;

}

return T();

}

T findSecondMin()

{

Node<T>\* secondMinNode = findSecondMinNode(root);

if (secondMinNode)

{

return secondMinNode->data;

}

return T();

}

bool isAVL()

{

return isAVLTree(root);

}

private:

Node<T>\* root;

int getHeight(Node<T>\* node)

{

if (node == nullptr)

{

return -1;

}

return node->height;

}

int getMax(int a, int b)

{

if (a > b)

{

return a;

}

return b;

}

int getBalanceFactor(Node<T>\* node)

{

if (node == nullptr)

{

return -1;

}

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

}

Node<T>\* rotateRight(Node<T>\* y)

{

Node<T>\* x = y->left;

Node<T>\* t2 = x->right;

x->right = y;

y->left = t2;

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

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

return x;

}

Node<T>\* rotateLeft(Node<T>\* x)

{

Node<T>\* y = x->right;

Node<T>\* t2 = y->left;

y->left = x;

x->right = t2;

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

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

return y;

}

Node<T>\* insertNode(Node<T>\* node, T value)

{

if (node == nullptr)

{

return new Node<T>(value);

}

if (value < node->data)

{

node->left = insertNode(node->left, value);

}

else if (value > node->data)

{

node->right = insertNode(node->right, value);

}

else

{

return node;

}

node->height = 1 + getMax(getHeight(node->left), getHeight(node->right));

int balanceFactor = getBalanceFactor(node);

if (balanceFactor > 1 && value < node->left->data)

{

return rotateRight(node);

}

if (balanceFactor < -1 && value > node->right->data)

{

return rotateLeft(node);

}

if (balanceFactor > 1 && value > node->left->data)

{

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

return rotateRight(node);

}

if (balanceFactor < -1 && value < node->right->data)

{

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

return rotateLeft(node);

}

return node;

}

bool isAVLTree(Node<T>\* node)

{

if (node == nullptr)

{

return true;

}

int balanceFactor = getBalanceFactor(node);

if (balanceFactor < -1 || balanceFactor > 1)

{

return false;

}

return isAVLTree(node->left) && isAVLTree(node->right);

}

bool searchNode(Node<T>\* node, T value)

{

if (node == nullptr)

{

return false;

}

if (value < node->data)

{

return searchNode(node->left, value);

}

else if (value > node->data)

{

return searchNode(node->right, value);

}

else

{

return true;

}

}

Node<T>\* findMinNode(Node<T>\* node)

{

if (node == nullptr || node->left == nullptr)

{

return node;

}

return findMinNode(node->left);

}

Node<T>\* findSecondMinNode(Node<T>\* node)

{

if (node == nullptr || (node->left == nullptr && node->right == nullptr))

{

return nullptr;

}

Node<T>\* current = node;

Node<T>\* parent = nullptr;

while (current->left != nullptr)

{

parent = current;

current = current->left;

}

if (current->right != nullptr)

{

return findMinNode(current->right);

}

return parent;

}

void inorderTraversalNode(Node<T>\* node)

{

if (node == nullptr)

{

return;

}

inorderTraversalNode(node->left);

cout << node->data << " ";

inorderTraversalNode(node->right);

}

void preorderTraversalNode(Node<T>\* node)

{

if (node == nullptr)

{

return;

}

cout << node->data << " ";

preorderTraversalNode(node->left);

preorderTraversalNode(node->right);

}

void postorderTraversalNode(Node<T>\* node)

{

if (node == nullptr)

{

return;

}

postorderTraversalNode(node->left);

postorderTraversalNode(node->right);

cout << node->data << " ";

}

};

int main()

{

AVLTree<int> avlTree;

avlTree.insert(13);

avlTree.insert(8);

avlTree.insert(5);

avlTree.insert(9);

avlTree.insert(4);

avlTree.insert(6);

avlTree.insert(12);

avlTree.insert(2);

avlTree.insert(1);

avlTree.insert(3);

cout << "Inorder traversal: ";

avlTree.inorderTraversal();

cout << "Preorder traversal: ";

avlTree.preorderTraversal();

cout << "Postorder traversal: ";

avlTree.postorderTraversal();

cout << "Searching for 25: ";

if (avlTree.search(25))

{

cout << "Found" << endl;

}

else

{

cout << "Not found" << endl;

}

cout << "Searching for 1: ";

if (avlTree.search(1))

{

cout << "Found" << endl;

}

else

{

cout << "Not found" << endl;

}

cout << "Minimum element: " << avlTree.findMin() << endl;

cout << "Second minimum element: " << avlTree.findSecondMin() << endl;

cout << "IsAVL tree: ";

if (avlTree.isAVL())

{

cout << "AVL tree" << endl;

}

else

{

cout << "Not AVL tree" << endl;

}

system("pause");

return 0;

}

## Output

