# Applied Programming



# Name: M. Hassan Sattar

# Roll No: 22F-3773

**Section: MS-1A**

# Submitted to: Sir. Muhammad Yousaf

**TASK-1**

**Task # 1: AVL Tree ADT.**

#include <iostream>

using namespace std;

template <class T>

struct Node

{

T value;

Node\* left, \* right;

int height;

};

template <class T>

class BST

{

private:

Node<T>\* tree;

void preorderTraversal(Node<T>\* node)

{

if (node == NULL)

return;

cout << node->value << ' ';

preorderTraversal(node->left);

preorderTraversal(node->right);

}

void inorderTraversal(Node<T>\* node)

{

if (node == NULL)

return;

preorderTraversal(node->left);

cout << node->value << ' ';

preorderTraversal(node->right);

}

void postorderTraversal(Node<T>\* node)

{

if (node == NULL)

return;

preorderTraversal(node->left);

preorderTraversal(node->right);

cout << node->value << ' ';

}

int findMax(Node<T>\* node)

{

if (node->right == NULL)

return node->value;

findMax(node->right);

}

int findMin(Node<T>\* node)

{

if (node->left == NULL)

return node->value;

findMin(node->left);

}

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

{

Node<T>\* temp = node;

while (temp->left != NULL)

{

temp = temp->left;

}

return temp;

}

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

{

if (node == NULL)

{

return node;

}

if (value < node->value)

{

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

}

else if (value > node->value)

{

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

}

else

{

if ((node->left == NULL) || (node->right == NULL))

{

Node<T>\* temp = node->left ? node->left : node->right;

if (temp == NULL)

{

temp = node;

node = NULL;

}

else

{

node = temp;

}

delete temp;

}

else

{

Node<T>\* temp = minValueNode(node->right);

node->value = temp->value;

node->right = DeleteKey(node->right, temp->value);

}

}

if (node == NULL)

{

return node;

}

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

int balance = getBalance(node);

if (balance > 1 && getBalance(node->left) >= 0)

return rightRotate(node);

if (balance > 1 && getBalance(node->left) < 0)

{

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

return rightRotate(node);

}

if (balance < -1 && getBalance(node->right) <= 0)

return leftRotate(node);

if (balance < -1 && getBalance(node->right) > 0)

{

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

return leftRotate(node);

}

return node;

}

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

{

if (node == NULL)

return false;

if (value > node->value)

return SearchKey(node->right, value);

else if (value < node->value)

return SearchKey(node->left, value);

else

return true;

}

int treeHeight(Node<T>\* node)

{

if (node == NULL)

return 0;

else

{

int left\_height = treeHeight(node->left);

int right\_height = treeHeight(node->right);

if (left\_height > right\_height)

return left\_height + 1;

else

return right\_height + 1;

}

}

int treeNodeCount(Node<T>\* node)

{

if (node == NULL)

return 0;

return 1 + treeNodeCount(node->left) + treeNodeCount(node->right);

}

int treeLeavesCount(Node<T>\* node)

{

if (node == NULL)

return 0;

if (node->left == NULL && node->right == NULL)

return 1;

else

return treeLeavesCount(node->left) + treeLeavesCount(node->right);

}

void printNodeOnEachLevel(Node<T>\* node, int level)

{

if (node == NULL)

return;

if (level == 1)

cout << node->value << " ";

else if (level > 1)

{

printNodeOnEachLevel(node->left, level - 1);

printNodeOnEachLevel(node->right, level - 1);

}

}

void printNodeLevel(Node<T>\* node)

{

int height = treeHeight(node);

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

{

printNodeOnEachLevel(node, i);

cout << endl;

}

}

// AVL Functions

int height(Node<T>\* node)

{

if (node == NULL)

return 0;

return node->height;

}

int max(int a, int b)

{

return (a > b) ? a : b;

}

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

{

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

Node<T>\* T2 = x->right;

x->right = node;

node->left = T2;

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

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

return x;

}

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

{

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

Node<T>\* T2 = y->left;

y->left = node;

node->right = T2;

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

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

return y;

}

int getBalance(Node<T>\* N)

{

if (N == NULL)

return 0;

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

}

Node<T>\* Insert(Node<T>\* node, T key)

{

if (node == NULL)

{

Node<T>\* temp = new Node<T>();

temp->value = key;

temp->left = NULL;

temp->right = NULL;

temp->height = 1;

return temp;

}

if (key < node->value)

node->left = Insert(node->left, key);

else if (key > node->value)

node->right = Insert(node->right, key);

else

return node;

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

int balance = getBalance(node);

if (balance > 1 && key < node->left->value)

return rightRotate(node);

if (balance < -1 && key > node->right->value)

return leftRotate(node);

if (balance > 1 && key > node->left->value)

{

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

return rightRotate(node);

}

if (balance < -1 && key < node->right->value)

{

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

return leftRotate(node);

}

return node;

}

public:

BST() : tree(NULL) {}

void Insert(T value)

{

tree = Insert(tree, value);

}

int findMax()

{

int result = findMax(tree);

return result;

}

int findMin()

{

int result = findMin(tree);

return result;

}

void preorderTraversal()

{

preorderTraversal(tree);

}

void inorderTraversal()

{

inorderTraversal(tree);

}

void postorderTraversal()

{

postorderTraversal(tree);

}

void DeleteKey(T value)

{

DeleteKey(tree, value);

}

void SearchKey(T value)

{

if (SearchKey(tree, value) == true)

{

cout << "Node with value[" << value << "], has been found" << endl;

}

else

{

cout << "Node with value[" << value << "], not found" << endl;

}

}

int treeHeight()

{

int height = treeHeight(tree);

return height;

}

int treeNodeCount()

{

int nodes = treeNodeCount(tree);

return nodes;

}

int treeLeavesCount()

{

int leaves = treeLeavesCount(tree);

return leaves;

}

void printNodeLevel()

{

printNodeLevel(tree);

}

};

int main()

{

BST<int> bst; // 65 55 22 44 61 19 90 10 78 52

bst.Insert(65);

bst.Insert(55);

bst.Insert(22);

bst.Insert(44);

bst.Insert(61);

bst.Insert(19);

bst.Insert(90);

bst.Insert(10);

bst.Insert(78);

bst.Insert(52);

bst.DeleteKey(10);

bst.DeleteKey(65);

bst.DeleteKey(100); // This value doesn't exist

bst.SearchKey(55);

bst.SearchKey(90);

bst.SearchKey(100); // This value doesn't exist

cout << "findMax(): " << bst.findMax() << endl;

cout << "findMin(): " << bst.findMin() << endl;

cout << "Pre Order: ";

bst.preorderTraversal();

cout << endl;

cout << "In Order: ";

bst.inorderTraversal();

cout << endl;

cout << "Post Order: ";

bst.postorderTraversal();

cout << endl;

cout << "treeHeight(): " << bst.treeHeight() << endl;

cout << "treeNodeCount(): " << bst.treeNodeCount() << endl;

cout << "treeLeavesCount(): " << bst.treeLeavesCount() << endl;

cout << "printNodeLevel(): " << endl;

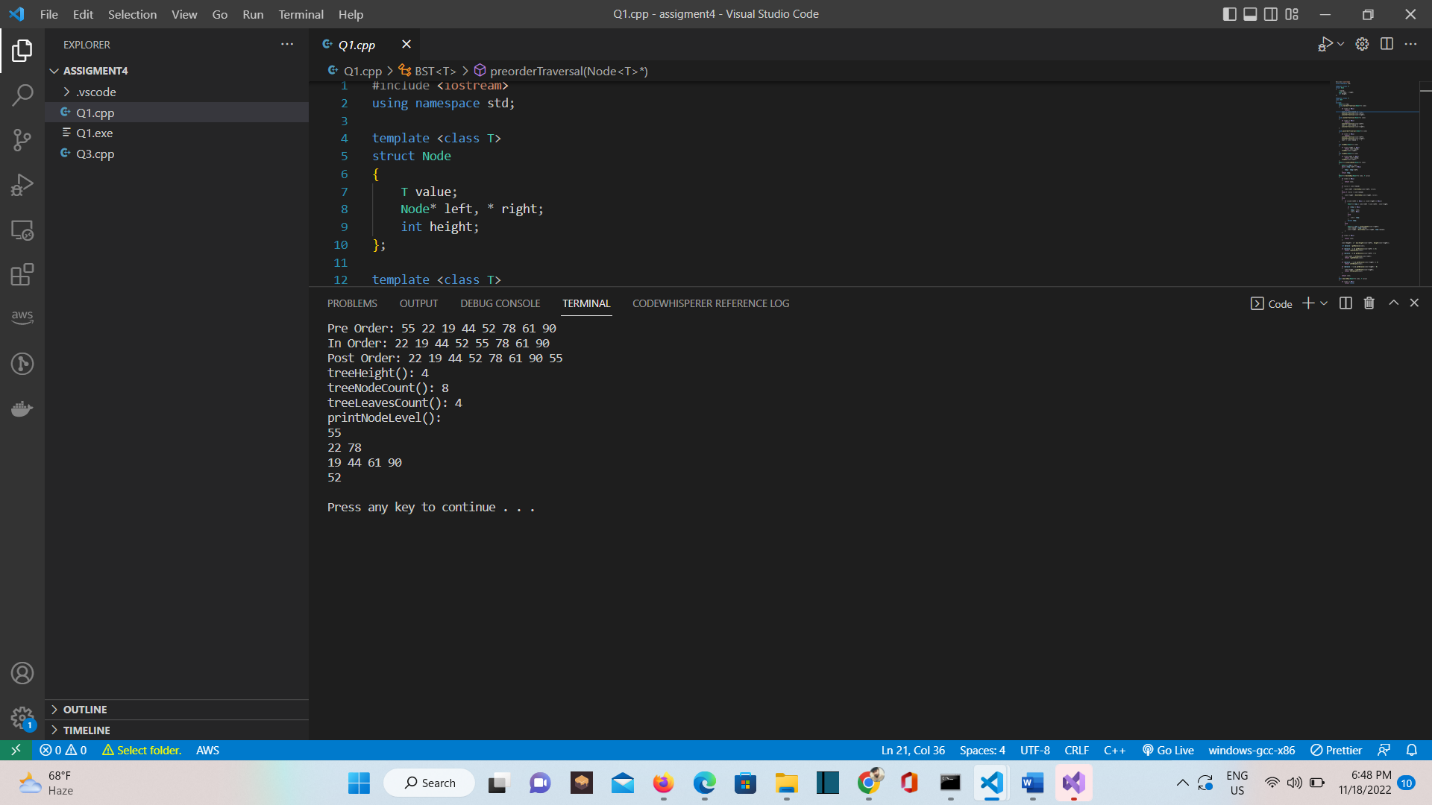
bst.printNodeLevel();

cout << endl;

system("PAUSE");

return 0;

}



**Task # 2**

**Task # 2: BST with Minimal Height.**

#include <iostream>

using namespace std;

struct Node

{

int data;

Node\* left;

Node\* right;

};

Node\* ADD\_NODE(int data);

class MirrorTree

{

public:

void MINIMNAL\_HEIGHT(int arr[], int start, int end);

void PRE\_ORDER(Node\* node);

};

Node\* MINIMNAL\_HEIGHT(int arr[], int start, int end)

{

if (start > end)

return NULL;

int mid = (start + end) / 2;

Node\* root = ADD\_NODE(arr[mid]);

root->left = MINIMNAL\_HEIGHT(arr, start, mid - 1);

root->right = MINIMNAL\_HEIGHT(arr, mid + 1, end);

return root;

}

Node\* ADD\_NODE(int data)

{

Node\* node = new Node();

node->data = data;

node->left = NULL;

node->right = NULL;

return node;

}

void PRE\_ORDER(Node\* node)

{

if (node == NULL)

return;

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

PRE\_ORDER(node->left);

PRE\_ORDER(node->right);

}

int main()

{

int arr[] = { 5, 6, 7, 8, 9, 10, 11 };

int n = sizeof(arr) / sizeof(arr[0]);

Node\* root = MINIMNAL\_HEIGHT(arr, 0, n - 1);

cout << "PreOrder of Minimal Height \n";

PRE\_ORDER(root);

return 0;

}

A screenshot of a computer

Description automatically generated

**Task # 3**

**Task # 3: Sum Pairs.**

#include <iostream>

using namespace std;

template <class T>

struct Node

{

T value;

Node\* left, \* right;

int height;

};

template <class T>

class BST

{

private:

Node<T>\* tree;

void preorderTraversal(Node<T>\* node)

{

if (node == NULL)

return;

cout << node->value << ' ';

preorderTraversal(node->left);

preorderTraversal(node->right);

}

void inorderTraversal(Node<T>\* node)

{

if (node == NULL)

return;

preorderTraversal(node->left);

cout << node->value << ' ';

preorderTraversal(node->right);

}

void postorderTraversal(Node<T>\* node)

{

if (node == NULL)

return;

preorderTraversal(node->left);

preorderTraversal(node->right);

cout << node->value << ' ';

}

int findMax(Node<T>\* node)

{

if (node->right == NULL)

return node->value;

findMax(node->right);

}

int findMin(Node<T>\* node)

{

if (node->left == NULL)

return node->value;

findMin(node->left);

}

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

{

Node<T>\* temp = node;

while (temp->left != NULL)

{

temp = temp->left;

}

return temp;

}

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

{

if (node == NULL)

{

return node;

}

if (value < node->value)

{

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

}

else if (value > node->value)

{

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

}

else

{

if ((node->left == NULL) || (node->right == NULL))

{

Node<T>\* temp = node->left ? node->left : node->right;

if (temp == NULL)

{

temp = node;

node = NULL;

}

else

{

node = temp;

}

delete temp;

}

else

{

Node<T>\* temp = minValueNode(node->right);

node->value = temp->value;

node->right = DeleteKey(node->right, temp->value);

}

}

if (node == NULL)

{

return node;

}

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

int balance = getBalance(node);

if (balance > 1 && getBalance(node->left) >= 0)

return rightRotate(node);

if (balance > 1 && getBalance(node->left) < 0)

{

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

return rightRotate(node);

}

if (balance < -1 && getBalance(node->right) <= 0)

return leftRotate(node);

if (balance < -1 && getBalance(node->right) > 0)

{

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

return leftRotate(node);

}

return node;

}

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

{

if (node == NULL)

return false;

if (value > node->value)

return SearchKey(node->right, value);

else if (value < node->value)

return SearchKey(node->left, value);

else

return true;

}

int treeHeight(Node<T>\* node)

{

if (node == NULL)

return 0;

else

{

int left\_height = treeHeight(node->left);

int right\_height = treeHeight(node->right);

if (left\_height > right\_height)

return left\_height + 1;

else

return right\_height + 1;

}

}

int treeNodeCount(Node<T>\* node)

{

if (node == NULL)

return 0;

return 1 + treeNodeCount(node->left) + treeNodeCount(node->right);

}

int treeLeavesCount(Node<T>\* node)

{

if (node == NULL)

return 0;

if (node->left == NULL && node->right == NULL)

return 1;

else

return treeLeavesCount(node->left) + treeLeavesCount(node->right);

}

void printNodeOnEachLevel(Node<T>\* node, int level)

{

if (node == NULL)

return;

if (level == 1)

cout << node->value << " ";

else if (level > 1)

{

printNodeOnEachLevel(node->left, level - 1);

printNodeOnEachLevel(node->right, level - 1);

}

}

void printNodeLevel(Node<T>\* node)

{

int height = treeHeight(node);

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

{

printNodeOnEachLevel(node, i);

cout << endl;

}

}

// AVL Functions

int height(Node<T>\* node)

{

if (node == NULL)

return 0;

return node->height;

}

int max(int a, int b)

{

return (a > b) ? a : b;

}

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

{

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

Node<T>\* T2 = x->right;

x->right = node;

node->left = T2;

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

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

return x;

}

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

{

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

Node<T>\* T2 = y->left;

y->left = node;

node->right = T2;

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

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

return y;

}

int getBalance(Node<T>\* N)

{

if (N == NULL)

return 0;

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

}

Node<T>\* Insert(Node<T>\* node, T key)

{

if (node == NULL)

{

Node<T>\* temp = new Node<T>();

temp->value = key;

temp->left = NULL;

temp->right = NULL;

temp->height = 1;

return temp;

}

if (key < node->value)

node->left = Insert(node->left, key);

else if (key > node->value)

node->right = Insert(node->right, key);

else

return node;

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

int balance = getBalance(node);

if (balance > 1 && key < node->left->value)

return rightRotate(node);

if (balance < -1 && key > node->right->value)

return leftRotate(node);

if (balance > 1 && key > node->left->value)

{

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

return rightRotate(node);

}

if (balance < -1 && key < node->right->value)

{

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

return leftRotate(node);

}

return node;

}

// Pair

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

{

if (tree == NULL)

{

return false;

}

Node<T>\* head\_ptr = tree;

bool flag = false;

while (head\_ptr != NULL && flag != true)

{

if (head\_ptr->value == value && node != head\_ptr)

{

cout << "Pair Found: " << head\_ptr->value << " + " << node->value << endl;

flag = true;

break;

}

else if (value < head\_ptr->value)

{

head\_ptr = head\_ptr->left;

}

else

{

head\_ptr = head\_ptr->right;

}

}

return flag;

}

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

{

if (node == NULL)

{

return false;

}

return SearchForPair(node, value - node->value) || isPairPresent(node->left, value) || isPairPresent(node->right, value);

}

public:

BST() : tree(NULL) {}

void Insert(T value)

{

tree = Insert(tree, value);

}

bool isPairPresent(T value)

{

bool result = isPairPresent(tree, value);

return result;

}

int findMax()

{

int result = findMax(tree);

return result;

}

int findMin()

{

int result = findMin(tree);

return result;

}

void preorderTraversal()

{

preorderTraversal(tree);

}

void inorderTraversal()

{

inorderTraversal(tree);

}

void postorderTraversal()

{

postorderTraversal(tree);

}

void DeleteKey(T value)

{

DeleteKey(tree, value);

}

void SearchKey(T value)

{

if (SearchKey(tree, value) == true)

{

cout << "Node with value[" << value << "], has been found" << endl;

}

else

{

cout << "Node with value[" << value << "], not found" << endl;

}

}

int treeHeight()

{

int height = treeHeight(tree);

return height;

}

int treeNodeCount()

{

int nodes = treeNodeCount(tree);

return nodes;

}

int treeLeavesCount()

{

int leaves = treeLeavesCount(tree);

return leaves;

}

void printNodeLevel()

{

printNodeLevel(tree);

}

};

int main()

{

BST<int> bst; // 65 55 22 44 61 19 90 10 78 52

bst.Insert(65);

bst.Insert(55);

bst.Insert(22);

bst.Insert(44);

bst.Insert(61);

bst.Insert(19);

bst.Insert(90);

bst.Insert(10);

bst.Insert(78);

bst.Insert(52);

bst.Insert(80);

bst.DeleteKey(10);

bst.DeleteKey(65);

bst.DeleteKey(100); // This value doesn't exist

bst.SearchKey(55);

bst.SearchKey(90);

bst.SearchKey(100); // This value doesn't exist

cout << "findMax(): " << bst.findMax() << endl;

cout << "findMin(): " << bst.findMin() << endl;

cout << "Pre Order: ";

bst.preorderTraversal();

cout << endl;

cout << "In Order: ";

bst.inorderTraversal();

cout << endl;

cout << "Post Order: ";

bst.postorderTraversal();

cout << endl;

cout << "treeHeight(): " << bst.treeHeight() << endl;

cout << "treeNodeCount(): " << bst.treeNodeCount() << endl;

cout << "treeLeavesCount(): " << bst.treeLeavesCount() << endl;

cout << "printNodeLevel(): " << endl;

bst.printNodeLevel();

cout << endl;

bst.isPairPresent(80);

return 0;

}

**A screenshot of a computer

Description automatically generated**