TASK 1:

MAIN:

#include <iostream>

#include <stack>

#include <math.h>

#include "AVL\_Tree.h"

using namespace std;

int main(){

BT<int> obj;

int choice;

int n;

do {

cout << "1- Insert" << endl;

cout << "2- Inorder" << endl;

cout << "3- Preorder" << endl;

cout << "4- Postorder" << endl;

cout << "5- Total Nodes" << endl;

cout << "6- Leaf Nodes" << endl;

cout << "7- Search Node" << endl;

cout << "8- Maximum" << endl;

cout << "9- Minimum" << endl;

cout << "10- Height" << endl;

cout << "11- Delete" << endl;

cin >> choice;

switch (choice) {

case 1:

cout << "How many value to insert : ";

cin >> choice;

for (int i = 0; i < choice; i++) {

cout << "Enter value to insert : ";

cin >> n;

obj.insertion(n);

}

break;

case 2:

cout << "Inorder Traversal : ";

obj.inOrder\_traversal(obj.getRoot());

cout << endl;

break;

case 3:

cout<< endl << "Preorder : ";

obj.preOrder\_traversal(obj.getRoot());

cout << endl;

break;

case 4:

cout << endl << "Postorder : ";

obj.postOrder\_traversal(obj.getRoot());

cout << endl;

break;

case 5:

cout << endl << "Total Nodes: " << obj.total\_number\_of\_nodes() << endl;

break;

case 6:

cout << "Leaf Nodes: " << obj.treeLeavesCount(obj.getRoot()) << endl;

break;

case 7:

cout << "Enter value to search : ";

cin >> n;

cout << "Search Node: ";

obj.search(n, obj.getRoot());

cout << endl;

break;

case 8:

cout << "Maximum is : " << obj.findMax(obj.getRoot()) << endl;

break;

case 9:

cout << "Minimum is : " << obj.findMin(obj.getRoot()) << endl;

break;

case 10:

cout << "Height is : " << obj.height(obj.getRoot()) << endl;

break;

case 11:

cout << "Enter value to delete : ";

cin >> n;

obj.DeleteKey(n);

break;

}

} while (choice != 12);

}

HEADER:

#pragma once

#include <iostream>

#include <stack>

#include <math.h>

using namespace std;

template <class T>

class node{

public:

T data;

node<T>\* left;

node<T>\* right;

node(){

left = right = NULL;

}

};

template <class tree>

class BT{

private:

node<tree>\* root;

int count;

void insert(int value, node<tree>\*& Node){

if (Node == NULL){

Node = new node<tree>;

Node->data = value;

count++;

}

else if (value < Node->data){

insert(value, Node->left);

}

else{

insert(value, Node->right);

}

if (balance(Node) == 2 && balance(Node->left) == 1){

Node = llRotation(Node);

}

else if (balance(Node) == 2 && balance(Node->left) == -1){

Node = lrRotation(Node);

}

else if (balance(Node) == -2 && balance(Node->right) == 1){

Node = rlRotation(Node);

}

else if (balance(Node) == -2 && balance(Node->right) == -1){

Node = rrRotation(Node);

}

}

node<tree>\* rrRotation(node<tree>\*& Node){

node<tree>\* temp;

temp = Node->right;

Node->right = temp->left;

temp->left = Node;

return temp;

}

node<tree>\* llRotation(node<tree>\*& Node){

node<tree>\* temp;

temp = Node->left;

Node->left = temp->right;

temp->right = Node;

return temp;

}

node<tree>\* rlRotation(node<tree>\*& Node){

node<tree>\* temp;

node<tree>\* temp2;

temp = Node->right;

temp2 = Node->right->left;

Node->right = temp2->left;

temp->left = temp2->right;

temp2->left = Node;

temp2->right = temp;

return temp2;

}

node<tree>\* lrRotation(node<tree>\*& Node){

node<tree>\* temp;

node<tree>\* temp2;

temp = Node->left;

temp2 = Node->left->right;

Node->left = temp2->right;

temp->right = temp2->left;

temp2->right = Node;

temp2->left = temp;

return temp2;

}

public:

BT(){

root = NULL;

count = 0;

}

bool isEmpty(){

return(root == NULL);

}

void insertion(int value){

insert(value, root);

}

void DeleteKey(int element){

Delete(element, root);

}

void Delete(int element, node<tree>\*& Node) {

node<tree>\* tempNodePtr;

if (Node)

if (isEmpty()) {

cout << element << " not found.\n";

}

else if (element < Node->data) {

Delete(element, Node->left);

}

else if (element > Node->data) {

Delete(element, Node->right);

}

else if (element == Node->data){

if (Node->right == NULL) {

tempNodePtr = Node;

cout << tempNodePtr->data << endl;

Node = Node->left;

delete tempNodePtr;

}

else if (Node->left == NULL) {

tempNodePtr = Node;

cout << tempNodePtr->data << endl;

Node = Node->right;

delete tempNodePtr;

}

else {

tempNodePtr = Node->right;

while (tempNodePtr->left) {

tempNodePtr = tempNodePtr->left;

}

tempNodePtr->left = Node->left;

tempNodePtr = Node;

cout << tempNodePtr->data << endl;

Node = Node->right;

delete tempNodePtr;

}

}

if (balance(Node) == 2 && balance(Node->left) == 1){

Node = llRotation(Node);

}

else if (balance(Node) == 2 && balance(Node->left) == -1){

Node = lrRotation(Node);

}

else if (balance(Node) == -2 && balance(Node->right) == 1){

Node = rlRotation(Node);

}

else if (balance(Node) == -2 && balance(Node->right) == -1){

Node = rrRotation(Node);

}

}

int findMax(node<tree>\* Node) {

if (Node == NULL) {

return 0;

}

else if (Node->right == NULL){

return Node->data;

}

findMax(Node->right);

}

int findMin(node<tree>\* Node) {

if (Node == NULL) {

return 0;

}

else if (Node->left == NULL){

return Node->data;

}

findMin(Node->left);

}

bool is\_bst(node<tree>\* Node){

if (Node == NULL){

return true;

}

if (Node->left != NULL && Node->data < Node->left->data){

return false;

}

if (Node->right != NULL && Node->data > Node->right->data){

return false;

}

if (!is\_bst(Node->left) || !is\_bst(Node->right)){

return false;

}

return true;

}

int balance(node<tree>\* Node){

if (Node == NULL){

return 0;

}

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

}

node<tree>\* getRoot(){

return root;

}

void inOrder\_traversal(node<tree>\* p){

if (p != NULL){

inOrder\_traversal(p->left);

cout << p->data << " ";

inOrder\_traversal(p->right);

}

}

void preOrder\_traversal(node<tree>\* p){

if (p == NULL){

cout << "empty" << endl;

return;

}

stack<node<tree>\*> temp;

temp.push(p);

/\*stack<int> pre;\*/

while (!temp.empty()){

node<tree>\* curr = temp.top();

cout << curr->data << " ";

temp.pop();

/\*pre.push(curr->data);\*/

if (curr->right != NULL){

temp.push(curr->right);

}

if (curr->left != NULL){

temp.push(curr->left);

}

}

}

void postOrder\_traversal(node<tree>\* p){

if (p == NULL){

cout << "empty" << endl;

return;

}

stack<node<tree>\*> temp;

temp.push(root);

stack<int> post;

while (!temp.empty()){

node<tree>\* curr = temp.top();

temp.pop();

post.push(curr->data);

if (curr->left != NULL){

temp.push(curr->left);

}

if (curr->right != NULL){

temp.push(curr->right);

}

}

while (!post.empty()){

cout << post.top() << " ";

post.pop();

}

}

int height(node<tree>\* temp){

if (temp == NULL){

return -1;

}

int left = height(temp->left);

int right = height(temp->right);

if (left >= right){

return left + 1;

}

else{

return right + 1;

}

}

int total\_number\_of\_nodes(){

return count;

}

int treeLeavesCount(node<tree>\* Node){

if (Node == NULL) {

return 0;

}

if (Node->left == NULL && Node->right == NULL) {

return 1;

}

else {

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

}

}

node<tree>\* search(int key, node<tree>\* temp){

node<tree>\* curr = temp;

if (curr == NULL){

return curr;

}

else if (key < curr->data){

search(key, curr->left);

}

else if (key > curr->data){

search(key, curr->right);

}

else{

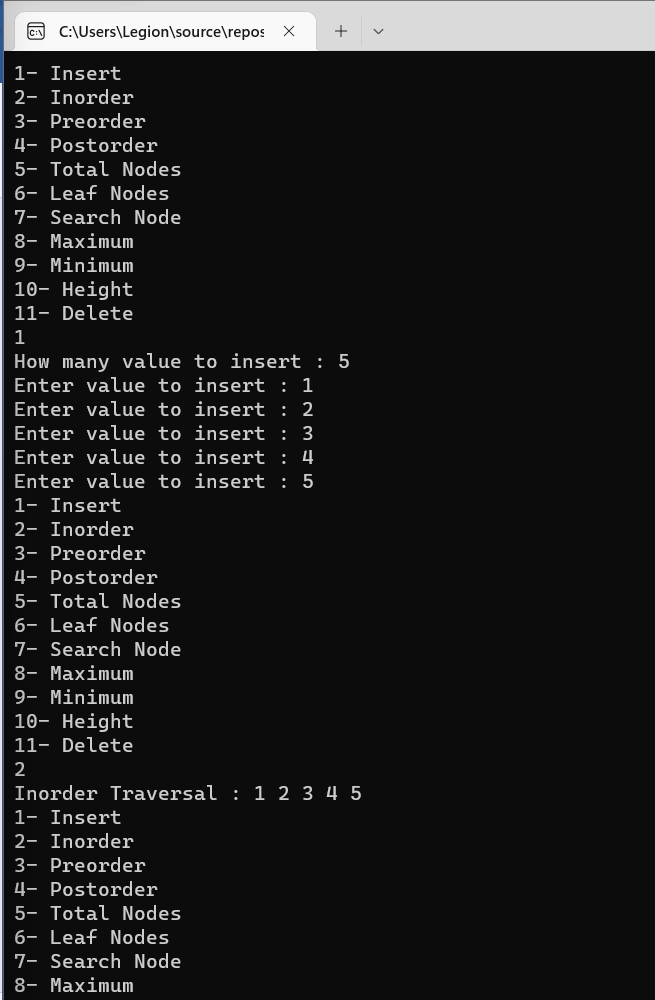
cout << curr->data << endl;

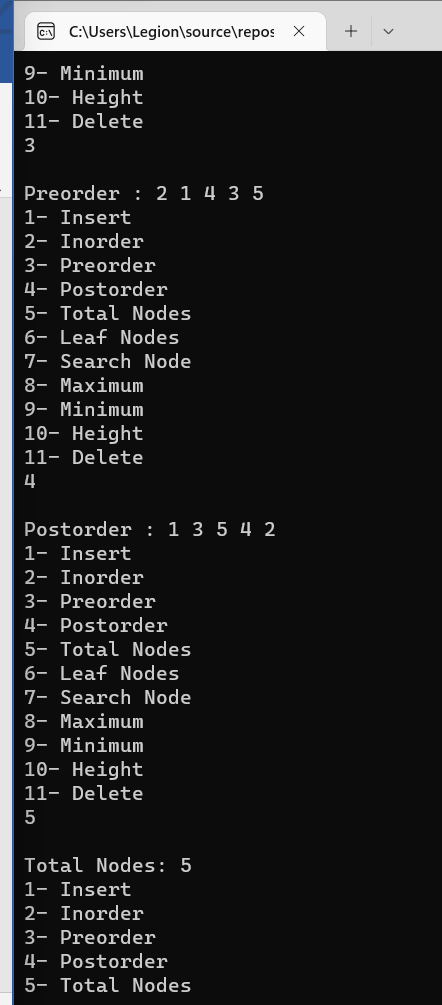
return curr;

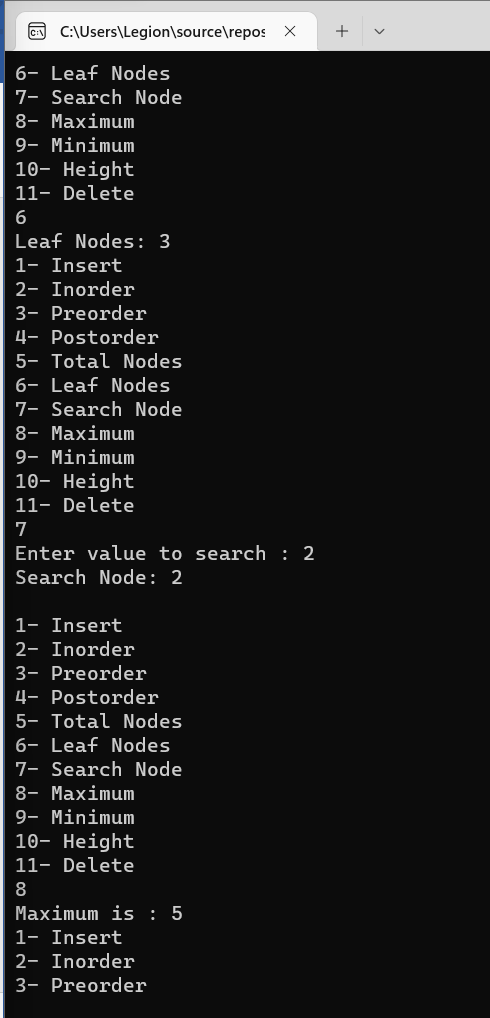
}

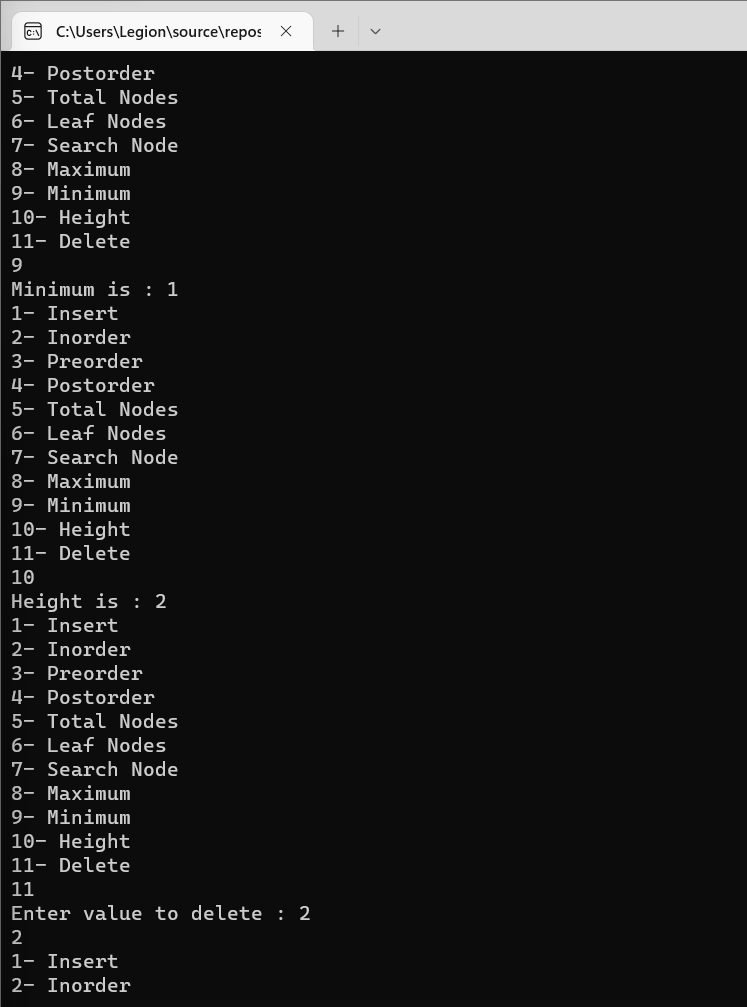
}

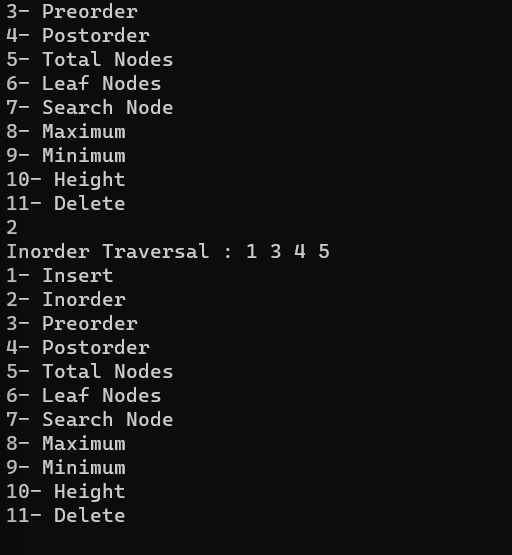
};











TASK 2:

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* left;

node\* right;

node() {

data = 0;

left = NULL;

right = NULL;

}

};

class AVLTree {

public:

node\* root;

AVLTree() {

root = NULL;

}

bool isTreeEmpty() {

if (root == NULL) {

return true;

}

else {

return false;

}

}

int height(node\* r) {

if (r == NULL)

return -1;

else {

int lheight = height(r->left);

int rheight = height(r->right);

if (lheight > rheight)

return (lheight + 1);

else return (rheight + 1);

}

}

int getBalanceFactor(node\* n) {

if (n == NULL)

return -1;

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

}

node\* rightRotate(node\* y) {

node\* x = y->left;

node\* T2 = x->right;

x->right = y;

y->left = T2;

return x;

}

node\* leftRotate(node\* x) {

node\* y = x->right;

node\* T2 = y->left;

y->left = x;

x->right = T2;

return y;

}

node\* insert(node\* r, node\* new\_node) {

if (r == NULL) {

r = new\_node;

cout << "data inserted successfully" << endl;

return r;

}

if (new\_node->data < r->data) {

r->left = insert(r->left, new\_node);

}

else if (new\_node->data > r->data) {

r->right = insert(r->right, new\_node);

}

else {

cout << "No duplicate datas allowed!" << endl;

return r;

}

int bf = getBalanceFactor(r);

if (bf > 1 && new\_node->data < r->left->data)

return rightRotate(r);

if (bf < -1 && new\_node->data > r->right->data)

return leftRotate(r);

if (bf > 1 && new\_node->data > r->left->data) {

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

return rightRotate(r);

}

if (bf < -1 && new\_node->data < r->right->data) {

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

return leftRotate(r);

}

return r;

}

node\* mindatanode(node\* root) {

node\* current = root;

while (current->left != NULL) {

current = current->left;

}

return current;

}

node\* deletenode(node\* r, int v) {

if (r == NULL) {

return NULL;

}

else if (v < r->data) {

r->left = deletenode(r->left, v);

}

else if (v > r->data) {

r->right = deletenode(r->right, v);

}

else {

if (r->left == NULL) {

node\* temp = r->right;

delete r;

return temp;

}

else if (r->right == NULL) {

node\* temp = r->left;

delete r;

return temp;

}

else {

node\* temp = mindatanode(r->right);

r->data = temp->data;

r->right = deletenode(r->right, temp->data);

}

}

int bf = getBalanceFactor(r);

if (bf == 2 && getBalanceFactor(r->left) >= 0)

return rightRotate(r);

else if (bf == 2 && getBalanceFactor(r->left) == -1) {

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

return rightRotate(r);

}

else if (bf == -2 && getBalanceFactor(r->right) <= -0)

return leftRotate(r);

else if (bf == -2 && getBalanceFactor(r->right) == 1) {

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

return leftRotate(r);

}

return r;

}

void printPreorder(node\* r)

{

if (r == NULL)

return;

cout << r->data << " ";

printPreorder(r->left);

printPreorder(r->right);

}

void printInorder(node\* r){

if (r == NULL)

return;

printInorder(r->left);

cout << r->data << " ";

printInorder(r->right);

}

void printPostorder(node\* r)

{

if (r == NULL)

return;

printPostorder(r->left);

printPostorder(r->right);

cout << r->data << " ";

}

node\* search(int v) {

if (root == NULL) {

return root;

}

else {

node\* temp = root;

while (temp != NULL) {

if (v == temp->data) {

return temp;

}

else if (v < temp->data) {

temp = temp->left;

}

else {

temp = temp->right;

}

}

return NULL;

}

}

};

node\* arrayToBS(int arr[], int start = 0, int end = 0)

{

if (start > end)

return NULL;

int mid = (start + end) / 2;

node\* newnode = new node;

newnode->data = arr[mid];

node\* root = newnode;

root->left = arrayToBS(arr, start, mid - 1);

root->right = arrayToBS(arr, mid + 1, end);

return root;

}

int main() {

AVLTree obj;

int arr[] = { 1,2,3,4,5,6,7 };

obj.root = arrayToBS(arr, 0, 6);

cout << "\nPreorder is: ";

obj.printPreorder(obj.root);

cout << "\nPostorder is: ";

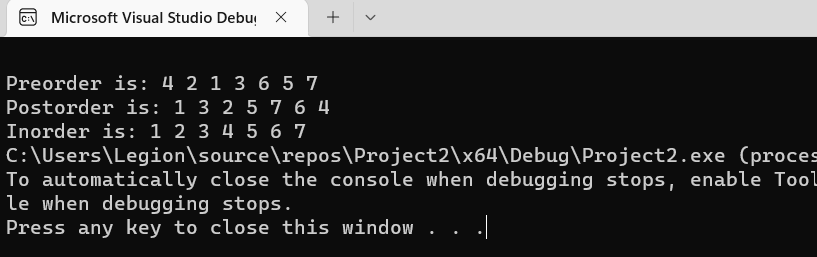
obj.printPostorder(obj.root);

cout << "\nInorder is: ";

obj.printInorder(obj.root);

return 0;

}



TASK 3:

#include <iostream>

#include <stack>

#include <math.h>

using namespace std;

template <class T>

class node{

public:

T data;

node<T>\* left;

node<T>\* right;

node()

{

left = right = NULL;

}

};

template <class tree>

class BT

{

private:

node<tree>\* root;

int count;

void insert(int value, node<tree>\*& Node)

{

if (Node == NULL)

{

Node = new node<tree>;

Node->data = value;

count++;

}

else if (value < Node->data)

{

insert(value, Node->left);

}

else

{

insert(value, Node->right);

}

if (balance(Node) == 2 && balance(Node->left) == 1)

{

Node = llRotation(Node);

}

else if (balance(Node) == 2 && balance(Node->left) == -1)

{

Node = lrRotation(Node);

}

else if (balance(Node) == -2 && balance(Node->right) == 1)

{

Node = rlRotation(Node);

}

else if (balance(Node) == -2 && balance(Node->right) == -1)

{

Node = rrRotation(Node);

}

}

node<tree>\* rrRotation(node<tree>\*& Node)

{

node<tree>\* temp;

temp = Node->right;

Node->right = temp->left;

temp->left = Node;

return temp;

}

node<tree>\* llRotation(node<tree>\*& Node)

{

node<tree>\* temp;

temp = Node->left;

Node->left = temp->right;

temp->right = Node;

return temp;

}

node<tree>\* rlRotation(node<tree>\*& Node)

{

node<tree>\* temp;

node<tree>\* temp2;

temp = Node->right;

temp2 = Node->right->left;

Node->right = temp2->left;

temp->left = temp2->right;

temp2->left = Node;

temp2->right = temp;

return temp2;

}

node<tree>\* lrRotation(node<tree>\*& Node)

{

node<tree>\* temp;

node<tree>\* temp2;

temp = Node->left;

temp2 = Node->left->right;

Node->left = temp2->right;

temp->right = temp2->left;

temp2->right = Node;

temp2->left = temp;

return temp2;

}

public:

BT()

{

root = NULL;

count = 0;

}

bool isEmpty()

{

return(root == NULL);

}

void insertion(int value)

{

insert(value, root);

}

int balance(node<tree>\* Node)

{

if (Node == NULL)

{

return 0;

}

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

}

node<tree>\* getRoot()

{

return root;

}

void inOrder\_traversal(node<tree>\* p)

{

if (p != NULL)

{

inOrder\_traversal(p->left);

cout << p->data << " ";

inOrder\_traversal(p->right);

}

}

void preOrder\_traversal(node<tree>\* p)

{

if (p == NULL)

{

cout << "empty" << endl;

return;

}

stack<node<tree>\*> temp;

temp.push(p);

/\*stack<int> pre;\*/

while (!temp.empty())

{

node<tree>\* curr = temp.top();

cout << curr->data << " ";

temp.pop();

/\*pre.push(curr->data);\*/

if (curr->right != NULL)

{

temp.push(curr->right);

}

if (curr->left != NULL)

{

temp.push(curr->left);

}

}

}

void postOrder\_traversal(node<tree>\* p)

{

if (p == NULL)

{

cout << "empty" << endl;

return;

}

stack<node<tree>\*> temp;

temp.push(root);

stack<int> post;

while (!temp.empty())

{

node<tree>\* curr = temp.top();

temp.pop();

post.push(curr->data);

if (curr->left != NULL)

{

temp.push(curr->left);

}

if (curr->right != NULL)

{

temp.push(curr->right);

}

}

while (!post.empty())

{

cout << post.top() << " ";

post.pop();

}

}

int height(node<tree>\*temp)

{

if (temp == NULL)

{

return -1;

}

int left = height(temp->left);

int right = height(temp->right);

if (left >= right)

{

return left + 1;

}

else

{

return right + 1;

}

}

int search(int key, node<tree>\* temp)

{

node<tree>\* curr = temp;

if (curr == NULL)

{

return 0;

}

else if (key < curr->data)

{

search(key, curr->left);

}

else if (key > curr->data)

{

search(key, curr->right);

}

else

{

return curr->data;

}

}

bool FindSumPair(node<tree>\* Node, int sum)

{

if (Node == NULL)

{

return false;

}

FindSumPair(Node->left, sum);

FindSumPair(Node->right, sum);

if (search(sum - Node->data, Node) == sum - Node->data)

{

cout << "Pair: " << Node->data << search(sum - Node->data, Node) << endl;

return true;

}

}

};

int main()

{

BT<int> obj;

obj.insertion(1);

obj.insertion(2);

obj.insertion(3);

obj.insertion(5);

obj.insertion(4);

cout << "InOrder:";

obj.inOrder\_traversal(obj.getRoot());

cout << endl << "Finding pairs of 10 \n";

obj.FindSumPair(obj.getRoot(), 10);

system("pause");

return 0;

}

