#include <iostream>

#include <algorithm>

#include <queue>

using namespace std;

class Node{

public:

Node\* left;

Node\* right;

int data;

Node(int val){

left=NULL;

right=NULL;

data=val;

}

};

class BST {

public:

Node\* root;

BST();

Node\* nn(int val); // Function to create a New Node

void insertNode(); // Function to insert value in BST

Node\* getroot(); // Function which returns root value

Node\* search(int key); // Function to search key in the tree

Node\* minNode(Node\* node); // Function which returns Address of node with minimum value

Node\* deleteNode(Node\* root, int key); // Function which deletes node

void inOrder(Node\* root); // Function which performs in-order traversal of tree

void preOrder(Node\* root); // Function which performs pre-order traversal of tree

void postOrder(Node\* root); // Function which performs post-order traversal of tree

int height(Node\* root); // Function which returns height of tree

void mirror(Node\* root); // Function which mirrors the tree

void levelOrder(Node\* root); // Function which performs level wise traversal

Node\* maxNode(Node\* root); // Function which returns Address of node with maximum value

void displayLeaf(Node\* root); // Function which displays all Leaf Nodes

void displayChildParent(Node\* root); // Function which displays all parent with their child

Node\* deleteNodeIterative(Node\* root, int key);

};

BST::BST(){

root = nullptr;

}

Node\* BST::nn(int val){

Node\* newNode=new Node(val); // Create a new Node

return newNode;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* INSERT NODE \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

void BST::insertNode() {

int val;

cout<<"Enter value: ";

cin>>val;

Node\* newNode=nn(val); // Create new Node

if(root==NULL){

root=newNode;

return;

}

Node\* search=root;

Node\* follow=NULL;

while(search!=NULL){

follow=search;

if(val > search->data){ // Value to be inserted greater than node

search=search->right;

}

else if(val < search->data){

search=search->left; // Value to be inserted greater than node

}

else{

cout<<"Duplicate Value!!."<<endl;

return; // Duplicate value

}

}

if(val > follow->data)

follow->right=newNode;

else

follow->left=newNode;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* RETURNS ROOT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

Node\* BST::getroot(){

return root;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* SEARCH NODE IN THE TREE \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

Node\* BST::search(int key) {

Node\* search = root;

while(search!= NULL) {

if(key==search->data){ // Key matches with node

return search;

}

else if(key>search->data){ // Key greater than node value

search=search->right;

}

else{

search=search->left; // Key lesser than node value

}

}

return NULL;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MINIMUM NODE \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

Node\* BST::minNode(Node\* node){

Node\* current=node;

while(current->left!=NULL){ // return minimum value by travelling to leftmost node

current=current->left;

}

return current;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MAXIMUM NODE \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

Node\* BST::maxNode(Node\* node){

Node\* current=node; // return minimum value by travelling to rightmost node

while(current->right!=NULL){

current=current->right;

}

return current;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* DELETE A NODE \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

Node\* BST::deleteNode(Node\* root, int key){

if(root==NULL){

return root;

}

Node\* temp = root;

if(root->data==key){

//0 Child

if(root->left==NULL && root->right==NULL){ // No child present, delete it

delete root;

return NULL;

}

//1 Child

//Left Child

if(root->left!=NULL && root->right==NULL){ // right child present and left child absent

delete root;

return temp;

}

//Right child

if(root->left==NULL && root->right!=NULL){ // right child absent and left child present

Node\* temp=root->right;

delete root;

return temp;

}

//2 Child

if(root->left!=NULL && root->right!=NULL){ // Both child present

int minval = minNode(root->right)->data;

root->data = minval;

root->right = deleteNode(root->right, minval);

return root;

}

}

else if(root->data > key){ // Key lesser than node value

root->left=deleteNode(root->left,key);

return root;

}

else{ // Key greater than node value

root->right=deleteNode(root->right,key);

return root;

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\* RECURSIVE IN-ORDER TRAVERSAL \*\*\*\*\*\*\*\*\*\*\*\*\*//

void BST::inOrder(Node\* root){

if(root==NULL) // Base case: if the current node is NULL, return

return;

inOrder(root->left); // Recursively traverse the left subtree

cout<<root->data<<" "; // Print the data of the current node

inOrder(root->right); // Recursively traverse the right subtree

}

//\*\*\*\*\*\*\*\*\*\*\*\*\* RECURSIVE PRE-ORDER TRAVERSAL \*\*\*\*\*\*\*\*\*\*\*\*\*//

void BST::preOrder(Node\* root){

if(root==NULL) // Base case: if the current node is NULL, return

return;

cout<<root->data<<" "; // Print the data of the current node

preOrder(root->left); // Recursively traverse the left subtree

preOrder(root->right); // Recursively traverse the right subtree

}

//\*\*\*\*\*\*\*\*\*\*\*\*\* RECURSIVE POST-ORDER TRAVERSAL \*\*\*\*\*\*\*\*\*\*\*\*\*//

void BST::postOrder(Node\* root){

if(root==NULL)

return;

postOrder(root->left); // Recursively traverse the left subtree

postOrder(root->right); // Recursively traverse the right subtree

cout<<root->data<<" "; // Print the data of the current node

}

//\*\*\*\*\*\*\*\*\*\*\*\*\* MIRROR THE TREE \*\*\*\*\*\*\*\*\*\*\*\*\*//

void BST::mirror(Node\* root){

Node\* temp=root;

if(root==NULL){

return;

}

swap(temp->right, temp->left);

mirror(temp->right);

mirror(temp->left);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\* HEIGHT OF THE TREE \*\*\*\*\*\*\*\*\*\*\*\*\*//

int BST::height(Node\* root){

if(root==NULL)

return 0;

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

}

//\*\*\*\*\*\*\*\*\*\*\*\*\* LEVEL WISE TRAVERSAL \*\*\*\*\*\*\*\*\*\*\*\*\*//

void BST::levelOrder(Node\* root){

if(!root)

return;

queue<Node\*>q; // Queue to store nodes for level-wise traversal

q.push(root); // Enqueue the root node

while(!q.empty()){

Node\* node=q.front(); // Front of the queue is the current node for processing

q.pop();

cout<<node->data<<" "; // Print the data of the current node

if(node->left!=NULL){ // Enqueue the left child if it exists

q.push(node->left);

}

if(node->right!=NULL){ // Enqueue the right child if it exists

q.push(node->right);

}

node=node->left;

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\* DISPLAY ALL LEAF NODES \*\*\*\*\*\*\*\*\*\*\*\*\*//

void BST::displayLeaf(Node\* root){

Node\* temp=root;

if(temp==NULL)

return; // Base case: If the current node is NULL, return

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

cout<<"Leaf Nodes: ";

cout<<temp->data<<" "; // Print the data of the leaf node

}

displayLeaf(temp->left); // Recursively call displayLeaf for the left subtree

displayLeaf(temp->right); // Recursively call displayLeaf for the right subtree

return;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\* DISPLAY PARENT WITH THEIR CHILD \*\*\*\*\*\*\*\*\*\*\*\*\*//

void BST::displayChildParent(Node\* root){

Node\* temp=root;

if(temp==NULL) // Base case: If the current node is NULL, return

return;

cout<<"Parent: "<<temp->data<<endl; // Print the data of the current node, representing the parent

if(temp->left)

cout<<"Left child: "<<temp->left->data<<endl; // Check if the left child exists and print its data

if(temp->right) // Check if the right child exists and print its data

cout<<"Right child: "<<temp->right->data<<endl;

displayChildParent(temp->left); // Recursively call displayChildParent for the left subtree

displayChildParent(temp->right); // Recursively call displayChildParent for the right subtree

}

//\*\*\*\*\*\*\*\*\*\*\*\*\* DISPLAY PARENT WITH THEIR CHILD \*\*\*\*\*\*\*\*\*\*\*\*\*//

Node\* BST::deleteNodeIterative(Node\* root, int key){

Node\* parent = nullptr;

Node\* current = root;

// Search for the node to be deleted

while(current != nullptr && current->data != key) {

parent = current;

if(key < current->data){

current = current->left;

}

else{

current = current->right;

}

}

// If the node is not found, return the original root

if(current == nullptr) {

return root;

}

// Node with only one child or no child

if(current->left == nullptr){

Node\* temp = current->right;

delete current;

if(parent == nullptr){

return temp; // If the node to be deleted is the root

}

else if (parent->left == current){

parent->left = temp;

}

else{

parent->right = temp;

}

}

else if(current->right == nullptr){

Node\* temp = current->left;

delete current;

if (parent == nullptr) {

return temp; // If the node to be deleted is the root

}

else if (parent->left == current){

parent->left = temp;

}

else{

parent->right = temp;

}

}

else{ // Node with two children

Node\* successor = minNode(current->right);

current->data = successor->data;

current->right = deleteNode(current->right, successor->data);

}

return root;

}

int main(){

BST b;

Node\* root=NULL;

Node\* result=NULL;

Node\* node=NULL;

int key,h;

while(true){

cout<<"\nEnter Operation"<<endl;

cout<<"1.Enter Data"<<endl;

cout<<"2.InOrder Traversal"<<endl;

cout<<"3.PreOrder Traversal"<<endl;

cout<<"4.PostOrder Traversal"<<endl;

cout<<"5.Search"<<endl;

cout<<"6.Mirror"<<endl;

cout<<"7.Delete"<<endl;

cout<<"8.Height"<<endl;

cout<<"9.Minimum Node"<<endl;

cout<<"10.Maximum Node"<<endl;

cout<<"11.Level Order Traversal"<<endl;

cout<<"12.Leaf Nodes"<<endl;

cout<<"13.Display Child and parent"<<endl;

cout<<"14.Exit"<<endl;

int op;

cin>>op;

switch(op){

case 1:

b.insertNode();

break;

case 2:

cout<<"\nInOrder: "<<endl;

root=b.getroot();

b.inOrder(root);

cout<<endl;

break;

case 3:

cout<<"\nPreOrder: "<<endl;

root=b.getroot();

b.preOrder(root);

cout<<endl;

break;

case 4:

cout<<"\nPostOrder: "<<endl;

root=b.getroot();

b.postOrder(root);

cout<<endl;

break;

case 5:

int key;

cout<<"Enter a value to search for: ";

cin>>key;

result=b.search(key);

if(result!=nullptr){

cout<<"Value "<<key<<" found in the tree." <<endl;

}

else{

cout<<"Value "<<key<<" not found in the tree."<<endl;

}

break;

case 6:

root=b.getroot();

b.mirror(root);

cout<<"Mirrored Tree: "<<endl;

b.inOrder(root);

cout<<endl;

case 7:

cout<<"Enter a value to delete: ";

cin>>key;

b.root=b.deleteNode(b.root, key);

break;

case 8:

h=b.height(root);

cout<<"Height is: "<<h<<endl;

break;

case 9:

node=b.minNode(root);

cout<<"Minimum Node: "<<node->data<<endl;

break;

case 10:

node=b.maxNode(root);

cout<<"Maximum Node: "<<node->data<<endl;

break;

case 11:

cout<<"\nLevel Order: "<<endl;

b.levelOrder(root);

break;

case 12:

b.displayLeaf(root);

break;

case 13:

b.displayChildParent(root);

break;

case 14:

return 0;

}

}

return 0;

}