





#pragma once

#include<iostream>

#include<queue>

using namespace std;

template <typename T>

class Node

{

public:

T key;

Node<T>\* left;

Node<T>\* right;

Node(T value)

{

key = value;

left = nullptr;

right = nullptr;

}

};

template <typename T>

class BST

{

private:

Node<T>\* root;

public:

BST()

{

root = nullptr;

}

void insert(T insertItem)

{

Node<T>\* ptr = new Node<T>(insertItem);

if (root == nullptr)

{

root = ptr;

return;

}

Node<T>\* current = root;

Node<T>\* parent = nullptr;

while (current != nullptr)

{

parent = current;

if (insertItem < current->key)

{

current = current->left;

}

else

{

current = current->right;

}

}

if (insertItem < parent->key)

{

parent->left = ptr;

}

else

{

parent->right = ptr;

}

}

bool search(T searchItem)

{

return searchHelper(root, searchItem);

}

bool searchHelper(Node<T>\* var1, T var2)

{

if (var1 == NULL)

{

return 0;

}

if (var2 == var1->key)

{

return 1;

}

else if (var2 < var1->key)

{

return searchHelper(var1->left, var2);

}

else

{

return searchHelper(var1->right, var2);

}

}

void inorderTraversalRecursive(Node<T>\* node)

{

if (node == NULL)

return;

inorderTraversalRecursive(node->left);

cout << node->key << " ";

inorderTraversalRecursive(node->right);

}

void inorderTraversal()

{

inorderTraversalRecursive(root);

}

void preorderTraversalRecursive(Node<T>\* node)

{

if (node == NULL)

return;

cout << node->key << " ";

preorderTraversalRecursive(node->left);

preorderTraversalRecursive(node->right);

}

void preOrderTraversal()

{

preorderTraversalRecursive(root);

std::cout << std::endl;

}

void postorderTraversalRecursive(Node<T>\* node)

{

if (node == NULL)

return;

postorderTraversalRecursive(node->left);

postorderTraversalRecursive(node->right);

cout << node->key << " ";

}

void postOrderTraversal()

{

postorderTraversalRecursive(root);

std::cout << std::endl;

}

Node<T>\* deleteNode(Node<T>\* currentNode, T key)

{

if (currentNode == NULL)

return currentNode;

if (key < currentNode->key)

{

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

}

else if (key > currentNode->key)

{

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

}

else

{

if (currentNode->left == nullptr)

{

Node<T>\* temp = currentNode->right;

delete currentNode;

return temp;

}

else if (currentNode->right == nullptr)

{

Node<T>\* temp = currentNode->left;

delete currentNode;

return temp;

}

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

currentNode->key = temp->key;

currentNode->right = deleteNode(currentNode->right, temp->key);

}

return currentNode;

}

void remove(T key)

{

root = deleteNode(root, key);

}

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

{

Node<T>\* current = node;

while (current->left != nullptr)

{

current = current->left;

}

return current;

}

void levelWise() {

if (root == nullptr)

{

return;

}

queue<Node<T>\*> ptr;

ptr.push(root);

while (!ptr.empty())

{

Node<T>\* current = ptr.front();

ptr.pop();

cout << current->key << " ";

if (current->left != nullptr)

{

ptr.push(current->left);

}

if (current->right != nullptr)

{

ptr.push(current->right);

}

}

cout << endl;

}

void display()

{

printTreeHelper(root, "", true);

}

void printTreeHelper(Node<T>\* node, string prefix, bool isLeft)

{

if (node != nullptr)

{

cout << prefix;

cout << (isLeft ? "|-- " : "|-- ");

cout << node->key << endl;

// Calculate new prefix for left and right child

string childPrefix = prefix + (isLeft ? "| " : " ");

printTreeHelper(node->left, childPrefix, true);

printTreeHelper(node->right, childPrefix, false);

}

}

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

{

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

return node;

if (key < node->key)

return findNode(node->left, key);

return findNode(node->right, key);

}

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

{

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

return nullptr;

if ((node->left != nullptr && node->left->key == key) ||

(node->right != nullptr && node->right->key == key))

return node;

if (key < node->key)

return findParent(node->left, key);

return findParent(node->right, key);

}

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

{

if (node->right != nullptr)

return minValueNode(node->right);

Node<T>\* successor = nullptr;

Node<T>\* ancestor = root;

while (ancestor != node)

{

if (node->key < ancestor->key)

{

successor = ancestor;

ancestor = ancestor->left;

}

else

ancestor = ancestor->right;

}

return successor;

}

int getHeight(Node<T>\* node)

{

if (node == nullptr)

return -1; // height of an empty tree

return max(getHeight(node->left), getHeight(node->right)) + 1;

}

int getHeight()

{

return getHeight(root);

}

int getNodeLevel(Node<T>\* node, T key, int level)

{

if (node == nullptr)

return -1;

if (node->key == key)

return level;

int leftLevel = getNodeLevel(node->left, key, level + 1);

if (leftLevel != -1)

return leftLevel;

return getNodeLevel(node->right, key, level + 1);

}

int getNodeLevel(T key)

{

return getNodeLevel(root, key, 0);

}

int getNodeDegree(Node<T>\* node)

{

int degree = 0;

if (node->left != nullptr)

degree++;

if (node->right != nullptr)

degree++;

return degree;

}

int getNodeDegree(T key)

{

Node<T>\* node = findNode(root, key);

if (node == nullptr)

return -1;

return getNodeDegree(node);

}

int getNodeCount(Node<T>\* node)

{

if (node == nullptr)

return 0;

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

}

int getNodeCount()

{

return getNodeCount(root);

}

bool isLeaf(Node<T>\* node)

{

return node != nullptr && node->left == nullptr && node->right == nullptr;

}

bool isLeaf(T key)

{

Node<T>\* node = findNode(root, key);

return isLeaf(node);

}

bool isNonLeaf(T key)

{

return !isLeaf(key);

}

bool isParent(T parentKey, T childKey)

{

Node<T>\* parentNode = findNode(root, parentKey);

if (parentNode == nullptr)

return false;

return (parentNode->left != nullptr && parentNode->left->key == childKey) ||

(parentNode->right != nullptr && parentNode->right->key == childKey);

}

bool isChild(T childKey, T parentKey)

{

return isParent(parentKey, childKey);

}

bool isDescendent(T ancestorKey, T descendentKey)

{

Node<T>\* ancestorNode = findNode(root, ancestorKey);

return searchHelper(ancestorNode, descendentKey);

}

bool isSuccessor(T nodeKey, T successorKey)

{

Node<T>\* node = findNode(root, nodeKey);

if (node == nullptr)

return false;

Node<T>\* successor = findSuccessor(node);

return successor != nullptr && successor->key == successorKey;

}

bool isSibling(T key1, T key2)

{

return findParent(root, key1) == findParent(root, key2);

}

int leafNodeCount(Node<T>\* node)

{

if (node == nullptr)

return 0;

if (isLeaf(node))

return 1;

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

}

int leafNodeCount()

{

return leafNodeCount(root);

}

int internalNodeCount(Node<T>\* node)

{

if (node == nullptr || isLeaf(node))

return 0;

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

}

int internalNodeCount()

{

return internalNodeCount(root);

}

};

int main()

{

BST<int> bst;

bst.insert(50);

bst.insert(30);

bst.insert(70);

bst.insert(20);

bst.insert(40);

bst.insert(60);

bst.insert(80);

bst.insert(35);

bst.insert(65);

cout << "Binary Search Tree:" << endl;

bst.display();

cout << endl;

cout << "Search for 40: " << (bst.search(40) ? "Found" : "Not Found") << endl;

cout << "Search for 100: " << (bst.search(100) ? "Found" : "Not Found") << endl;

cout << "Inorder Traversal: ";

bst.inorderTraversal();

cout << endl;

cout << "Preorder Traversal: ";

bst.preOrderTraversal();

cout << endl;

cout << "Postorder Traversal: ";

bst.postOrderTraversal();

cout << endl;

cout << "Tree Height: " << bst.getHeight() << endl;

cout << "Level of Node 40: " << bst.getNodeLevel(40) << endl;

cout << "Degree of Node 70: " << bst.getNodeDegree(70) << endl;

cout << "Total Number of Nodes: " << bst.getNodeCount() << endl;

cout << "Is Node 40 a Leaf? " << (bst.isLeaf(40) ? "Yes" : "No") << endl;

cout << "Is Node 20 a Leaf? " << (bst.isLeaf(20) ? "Yes" : "No") << endl;

cout << "Is Node 30 a Non-Leaf? " << (bst.isNonLeaf(30) ? "Yes" : "No") << endl;

cout << "Is Node 50 a Parent of 30? " << (bst.isParent(50, 30) ? "Yes" : "No") << endl;

cout << "Is Node 70 a Parent of 60? " << (bst.isParent(70, 60) ? "Yes" : "No") << endl;

cout << "Is Node 30 a Child of 50? " << (bst.isChild(30, 50) ? "Yes" : "No") << endl;

cout << "Is Node 60 a Descendant of 50? " << (bst.isDescendent(50, 60) ? "Yes" : "No") << endl;

cout << "Is Node 20 a Descendant of 70? " << (bst.isDescendent(70, 20) ? "Yes" : "No") << endl;

cout << "Is Node 60 a Successor of 50? " << (bst.isSuccessor(50, 60) ? "Yes" : "No") << endl;

cout << "Is Node 35 a Successor of 30? " << (bst.isSuccessor(30, 35) ? "Yes" : "No") << endl;

cout << "Are Nodes 30 and 70 Siblings? " << (bst.isSibling(30, 70) ? "Yes" : "No") << endl;

cout << "Are Nodes 20 and 40 Siblings? " << (bst.isSibling(20, 40) ? "Yes" : "No") << endl;

cout << "Leaf Node Count: " << bst.leafNodeCount() << endl;

cout << "Internal Node Count: " << bst.internalNodeCount() << endl;

cout << "Deleting node 20 (Leaf):" << endl;

bst.remove(20);

bst.display();

cout << "Deleting node 30 (Node with one child):" << endl;

bst.remove(30);

bst.display();

cout << "Deleting node 50 (Node with two children):" << endl;

bst.remove(50);

bst.display();

return 0;

}