#include <iostream>

using namespace std;

template<typename T>

class Node{

    public:

        T key;

        int height;

        Node<T>\* left;

        Node<T>\* right;

        Node(T data) : key(data), height(1), left(nullptr), right(nullptr){}

};

template<typename T>

class AVL{

    private:

        Node<T>\* root;

        int max(int a,int b){

            return (a>b)? a:b;

        }

        Node<T>\* findMin(Node<T>\* root){

            if(!root->left) return root;

            return findMin(root->left);

        }

        Node<T>\* findMax(Node<T>\* root){

            if(!root->right) return root;

            return findMax(root->right);

        }

        bool search(Node<T>\* root,T key){

            if(!root) return false;

            if(key==root->key) return true;

            else if(key<root->key) return search(root->left,key);

            else return search(root->right,key);

        }

        int height(Node<T>\* root){

            if(!root) return 0;

            return root->height;

        }

        int getBalanceFactor(Node<T>\* root){

            if(!root) return 0;

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

        }

        Node<T>\* rotateLeft(Node<T>\* root){

                Node<T>\* tmp=root->right;

                Node<T>\* leftChild=tmp->left;

                tmp->left=root;

                root->right=leftChild;

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

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

                return tmp;

        }

        Node<T>\* rotateRight(Node<T>\* root){

                Node<T>\* tmp=root->left;

                Node<T>\* rightChild=tmp->right;

                tmp->right=root;

                root->left=rightChild;

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

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

                return tmp;

        }

        Node<T>\* balance\_tree(Node<T>\* root){

            int balanceFactor=getBalanceFactor(root);

            if(balanceFactor>1 && getBalanceFactor(root->left)>=0) return rotateRight(root);

            if(balanceFactor>1 && getBalanceFactor(root->left)<0){

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

                return rotateRight(root);

            }

            if(balanceFactor<-1 && getBalanceFactor(root->right)<=0) return rotateLeft(root);

            if(balanceFactor<-1 && getBalanceFactor(root->right)>0){

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

                return rotateLeft(root);

            }

            return root;

        }

        Node<T>\* insert(Node<T>\* root,T key){

            if(!root) return new Node<T>(key);

            if(key<root->key) root->left=insert(root->left,key);

            else if(key>root->key) root->right=insert(root->right,key);

            else return root;

            if(!root) return root;

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

            return balance\_tree(root);

        }

        Node<T>\* remove(Node<T>\* root,T key){

            if(!root) return root;

            if(key<root->key) root->left=remove(root->left,key);

            else if(key>root->key) root->right=remove(root->right,key);

            else {

                if (!root->left && !root->right) {

                    delete root;

                    return nullptr;

                } else if (!root->left) {

                    Node<T>\* tmp=root->right;

                    delete root;

                    return tmp;

                } else if (!root->right) {

                    Node<T>\* tmp=root->left;

                    delete root;

                    return tmp;

                } else {

                    Node<T>\* tmp=findMin(root->right);

                    root->key=tmp->key;

                    root->right=remove(root->right, tmp->key);

                }

            }

            if(!root) return root;

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

            return balance\_tree(root);

        }

        void printCurrentLevel(Node<T>\* root, int level){

            if(!root) return;

            if(level==0) cout<<root->key<<" ";

            else if(level>0){

                printCurrentLevel(root->left,level-1);

                printCurrentLevel(root->right,level-1);

            }

        }

        void levelOrderTraversal(Node<T>\* root){

            if(!root) return;

            int h=height(root)+1;

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

                printCurrentLevel(root,i);

                cout<<endl;

            }

        }

        void printCurrentLevel\_height(Node<T>\* root, int level){

            if(!root) return;

            if(level==0) cout<<"   "<<root->key<<"->"<<root->height<<"   ";

            else if(level>0){

                printCurrentLevel\_height(root->left,level-1);

                printCurrentLevel\_height(root->right,level-1);

            }

        }

        void printHeight(Node<T>\* root){

            if(!root) return;

            int h=root->height;

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

                printCurrentLevel\_height(root,i);

                cout<<endl;

            }

        }

        void preOrderTraversal(Node<T>\* root){

            if(!root) return;

            cout<<root->key<<" ";

            preOrderTraversal(root->left);

            preOrderTraversal(root->right);

        }

        void inOrderTraversal(Node<T>\* root){

            if(!root) return;

            inOrderTraversal(root->left);

            cout<<root->key<<" ";

            inOrderTraversal(root->right);

        }

        void postOrderTraversal(Node<T>\* root){

            if(!root) return;

            postOrderTraversal(root->left);

            postOrderTraversal(root->right);

            cout<<root->key<<" ";

        }

    public:

        AVL(): root(nullptr){}

        void preOrderTraversal(){

            if(!root) return;

            preOrderTraversal(root);

        }

        void inOrderTraversal(){

            if(!root) return;

            inOrderTraversal(root);

        }

        void postOrderTraversal(){

            if(!root) return;

             postOrderTraversal(root);

        }

        void levelOrderTraversal(){

            if(!root) return;

            levelOrderTraversal(root);

        }

        void printHeight(){

            if(!root) return;

            printHeight(root);

        }

        void remove(T key){

            if(!root) return;

            root=remove(root, key);

        }

        void insert(T key){

            root=insert(root, key);

        }

        bool search(T key){

            return search(root, key);

        }

        int findMin(){

            return findMin(root)->key;

        }

        int findMax(){

            return findMax(root)->key;

        }

};

int main(){

    AVL<int> tree;

    tree.insert(1);

    tree.insert(2);

    tree.insert(3);

    tree.insert(4);

    tree.insert(5);

    tree.insert(6);

    tree.insert(7);

    cout<<"LEVEL-ORDER TRAVERSAL:"<<endl;

    tree.levelOrderTraversal();

    cout<<endl;

    cout<<"IN-ORDER TRAVERSAL: " ;

    tree.inOrderTraversal();

    cout<<endl;

    cout<<"PRE-ORDER TRAVERSAL: ";

    tree.preOrderTraversal();

    cout<<endl;

    cout<<"POST-ORDER TRAVERSAL: ";

    tree.postOrderTraversal();

    tree.remove(3);

    cout<<endl;

    cout<<"AFTER REMOVING 3:"<<endl;

    cout<<"LEVEL-ORDER TRAVERSAL:"<<endl;

    tree.levelOrderTraversal();

    cout<<endl;

    cout<<"IN-ORDER TRAVERSAL: ";

    tree.inOrderTraversal();

    cout<<endl;

    cout<<"PRE-ORDER TRAVERSAL: ";

    tree.preOrderTraversal();

    cout<<endl;

    cout<<"POST-ORDER TRAVERSAL: ";

    tree.postOrderTraversal();

    cout<<endl;

    int key=8;

    if(tree.search(key)){

        cout<<"FOUND: "<<key;

    } else{

        cout<<key<<" NOT FOUND"<<endl;

        tree.insert(key);

        cout<<"INSERTING......."<<endl;

    }

    cout<<endl;

    cout<<"SMALLEST VALUE: "<<tree.findMin()<<endl;

    cout<<"LARGEST  VALUE: "<<tree.findMax()<<endl;

    cout<<endl;

    cout<<"HEIGHT"<<endl;

    tree.printHeight();

    return 0;

}

