#include <iostream>

#include <complex>

#include <memory>

template <typename T>

class PersistentRBTree {

private:

enum Color { RED, BLACK };

struct Node {

T value;

Color color;

std::shared\_ptr<Node> left, right, parent;

Node(T value, Color color, std::shared\_ptr<Node> parent = nullptr)

: value(value), color(color), parent(parent), left(nullptr), right(nullptr) {}

};

using NodePtr = std::shared\_ptr<Node>;

NodePtr root;

NodePtr createNode(T value, Color color, NodePtr parent = nullptr) {

return std::make\_shared<Node>(value, color, parent);

}

void leftRotate(NodePtr x) {

NodePtr y = x->right;

x->right = y->left;

if (y->left != nullptr)

y->left->parent = x;

y->parent = x->parent;

if (x->parent == nullptr)

root = y;

else if (x == x->parent->left)

x->parent->left = y;

else

x->parent->right = y;

y->left = x;

x->parent = y;

}

void rightRotate(NodePtr x) {

NodePtr y = x->left;

x->left = y->right;

if (y->right != nullptr)

y->right->parent = x;

y->parent = x->parent;

if (x->parent == nullptr)

root = y;

else if (x == x->parent->right)

x->parent->right = y;

else

x->parent->left = y;

y->right = x;

x->parent = y;

}

void insertFixup(NodePtr z) {

while (z->parent != nullptr && z->parent->color == RED) {

if (z->parent == z->parent->parent->left) {

NodePtr y = z->parent->parent->right;

if (y != nullptr && y->color == RED) {

z->parent->color = BLACK;

y->color = BLACK;

z->parent->parent->color = RED;

z = z->parent->parent;

} else {

if (z == z->parent->right) {

z = z->parent;

leftRotate(z);

}

z->parent->color = BLACK;

z->parent->parent->color = RED;

rightRotate(z->parent->parent);

}

} else {

NodePtr y = z->parent->parent->left;

if (y != nullptr && y->color == RED) {

z->parent->color = BLACK;

y->color = BLACK;

z->parent->parent->color = RED;

z = z->parent->parent;

} else {

if (z == z->parent->left) {

z = z->parent;

rightRotate(z);

}

z->parent->color = BLACK;

z->parent->parent->color = RED;

leftRotate(z->parent->parent);

}

}

}

root->color = BLACK;

}

NodePtr insertNode(NodePtr root, NodePtr node) {

if (root == nullptr)

return node;

if (node->value.real() < root->value.real() ||

(node->value.real() == root->value.real() && node->value.imag() < root->value.imag())) {

root->left = insertNode(root->left, node);

root->left->parent = root;

} else {

root->right = insertNode(root->right, node);

root->right->parent = root;

}

return root;

}

void transplant(NodePtr u, NodePtr v) {

if (u->parent == nullptr)

root = v;

else if (u == u->parent->left)

u->parent->left = v;

else

u->parent->right = v;

if (v != nullptr)

v->parent = u->parent;

}

NodePtr minimum(NodePtr node) {

while (node->left != nullptr)

node = node->left;

return node;

}

void deleteFixup(NodePtr x) {

while (x != root && (x == nullptr || x->color == BLACK)) {

if (x == x->parent->left) {

NodePtr w = x->parent->right;

if (w->color == RED) {

w->color = BLACK;

x->parent->color = RED;

leftRotate(x->parent);

w = x->parent->right;

}

if ((w->left == nullptr || w->left->color == BLACK) &&

(w->right == nullptr || w->right->color == BLACK)) {

if (w != nullptr)

w->color = RED;

x = x->parent;

} else {

if (w->right == nullptr || w->right->color == BLACK) {

if (w->left != nullptr)

w->left->color = BLACK;

if (w != nullptr)

w->color = RED;

rightRotate(w);

w = x->parent->right;

}

if (w != nullptr)

w->color = x->parent->color;

x->parent->color = BLACK;

if (w->right != nullptr)

w->right->color = BLACK;

leftRotate(x->parent);

x = root;

}

} else {

NodePtr w = x->parent->left;

if (w->color == RED) {

w->color = BLACK;

x->parent->color = RED;

rightRotate(x->parent);

w = x->parent->left;

}

if ((w->left == nullptr || w->left->color == BLACK) &&

(w->right == nullptr || w->right->color == BLACK)) {

if (w != nullptr)

w->color = RED;

x = x->parent;

} else {

if (w->left == nullptr || w->left->color == BLACK) {

if (w->right != nullptr)

w->right->color = BLACK;

if (w != nullptr)

w->color = RED;

leftRotate(w);

w = x->parent->left;

}

if (w != nullptr)

w->color = x->parent->color;

x->parent->color = BLACK;

if (w->left != nullptr)

w->left->color = BLACK;

rightRotate(x->parent);

x = root;

}

}

}

if (x != nullptr)

x->color = BLACK;

}

NodePtr deleteNode(NodePtr root, T value) {

NodePtr z = root;

NodePtr x, y;

while (z != nullptr && !(z->value == value))

if (value.real() < z->value.real() ||

(value.real() == z->value.real() && value.imag() < z->value.imag()))

z = z->left;

else

z = z->right;

if (z == nullptr)

return root;

y = z;

Color y\_original\_color = y->color;

if (z->left == nullptr) {

x = z->right;

transplant(z, z->right);

} else if (z->right == nullptr) {

x = z->left;

transplant(z, z->left);

} else {

y = minimum(z->right);

y\_original\_color = y->color;

x = y->right;

if (y->parent == z) {

if (x != nullptr)

x->parent = y;

} else {

transplant(y, y->right);

y->right = z->right;

y->right->parent = y;

}

transplant(z, y);

y->left = z->left;

y->left->parent = y;

y->color = z->color;

}

if (y\_original\_color == BLACK)

deleteFixup(x);

return root;

}

public:

PersistentRBTree() : root(nullptr) {}

void insert(T value) {

NodePtr node = createNode(value, RED);

root = insertNode(root, node);

insertFixup(node);

}

void remove(T value) {

root = deleteNode(root, value);

}

bool search(T value) {

NodePtr current = root;

while (current != nullptr) {

if (current->value == value) {

return true;

} else if (value.real() < current->value.real() ||

(value.real() == current->value.real() && value.imag() < current->value.imag())) {

current = current->left;

} else {

current = current->right;

}

}

return false;

}

void inOrderTraversal(NodePtr node) const {

if (node != nullptr) {

inOrderTraversal(node->left);

std::cout << "(" << node->value.real() << ", " << node->value.imag() <<") ";

inOrderTraversal(node->right);

}

}

void printTree() const {

inOrderTraversal(root);

std::cout << std::endl;

}

};

int main() {

PersistentRBTree<std::complex<int>> tree;

tree.insert({1, 2});

tree.insert({3, 4});

tree.insert({5, 6});

tree.insert({7, 8});

std::cout << "Tree after insertions:" << std::endl;

tree.printTree();

std::cout << "Search for (3, 4): " << (tree.search({3, 4}) ? "Found" : "Not Found") << std::endl;

std::cout << "Search for (9, 10): " << (tree.search({9, 10}) ? "Found" : "Not Found") << std::endl;

tree.remove({3, 4});

std::cout << "Tree after removing (3, 4):" << std::endl;

tree.printTree();

return 0;

}