#include <iostream>

using namespace std;

template <typename Object, typename Comparator = less<Object>>

class BinarySearchTree

{

public:

struct BinaryNode

{

Object element;

BinaryNode \* left;

BinaryNode \* right;

BinaryNode(const Object & theElement, BinaryNode \* lt, BinaryNode \* rt)

: element{ theElement }, left{ lt }, right{ rt } { }

BinaryNode(Object && theElement, BinaryNode \* lt, BinaryNode \* rt)

: element{ std::move(theElement) }, left{ lt }, right{ rt } { }

};

BinaryNode \* root;

Comparator isLessThan;

bool contains(const Object & Obj, BinaryNode \* Node) const

{

if (Node == nullptr)

return false;

else if (isLessThan(Obj, Node->element))

return contains(Obj, Node->left);

else if (isLessThan(Node->element, Obj))

return contains(Obj, Node->right);

else

return true;

}

template <int> BinaryNode\* findMin(BinaryNode\* Node) const {

if (Node == nullptr)

return nullptr;

if (Node->left == nullptr)

return Node;

return findMin(Node->left);

}

BinaryNode\* findMax(BinaryNode\* Node) const

{

if (Node != nullptr)

while (Node->right != nullptr)

Node = Node->right;

return Node;

}

void insert(const Object& Obj, BinaryNode\*& Node)

{

if (Node == nullptr)

Node = new BinaryNode{ Obj, nullptr, nullptr };

else if (Obj < Node->element)

insert(Obj, Node->left);

else if (Node->element < Obj)

insert(Obj, Node->right);

else

;

}

void insert(Object&& Obj, BinaryNode\*& Node)

{

if (Node == nullptr)

Node = new BinaryNode{ std::move(Obj), nullptr, nullptr };

else if (Obj < Node->element)

insert(std::move(Obj), Node->left);

else if (Node->element < Obj)

insert(std::move(Obj), Node->right);

else

;

}

void remove(const Object& Obj, BinaryNode\* &Node)

{

if (Node == nullptr)

return;

if (Obj < Node->element)

remove(Obj, Node->left);

else if (Node->element < Obj)

remove(Obj, Node->right);

else if (Node->left != nullptr && Node->right != nullptr)

{

Node->element = findMin(Node->right)->element;

remove(Node->element, Node->right);

}

else

{

BinaryNode\* oldNode = Node;

Node = (Node->left != nullptr) ? Node->left : Node->right;

delete oldNode;

}

}

void makeEmpty(BinaryNode\*& Node)

{

if (Node != nullptr)

{

makeEmpty(Node->left);

makeEmpty(Node->right);

delete Node;

}

Node = nullptr;

}

~BinarySearchTree() {

makeEmpty(root);

}

BinarySearchTree(const BinarySearchTree& rhs) : root{ nullptr }

{

root = clone(rhs.root);

}

BinaryNode\* clone(BinaryNode\* Node) const

{

if (Node == nullptr)

return nullptr;

else

return new BinaryNode{ Node->element, clone(Node->left), clone(Node->right) };

}

void printTree(BinaryNode\* Node, ostream& out) const

{

if (Node != nullptr)

{

printTree(Node->left, out);

out << Node->element << endl;

printTree(Node->right, out);

}

}

};

//-------------------------------------------------------------------------------------------------------------------------------//

template <typename Object, typename Comparator = less<Object>>

class AVLTree

{

public:

struct AvlNode

{

Object element;

AvlNode \* left;

AvlNode \* right;

int height;

AvlNode(const Object & ele, AvlNode \* lt, AvlNode \* rt, int h = 0)

: element{ ele }, left{ lt }, right{ rt }, height{ h } { }

AvlNode(Object && ele, AvlNode \* lt, AvlNode \* rt, int h = 0)

: element{ std::move(ele) }, left{ lt }, right{ rt }, height{ h } { }

};

AvlNode\* root;

Comparator isLessThan;

bool contains(const Object& Obj, AvlNode\* Node) const

{

if (Node == nullptr)

return false;

else if (isLessThan(Obj, Node->element))

return contains(Obj, Node->left);

else if (isLessThan(Node->element, Obj))

return contains(Obj, Node->right);

else

return true;

}

AvlNode\* findMin(AvlNode\* Node) const {

if (Node == nullptr)

return nullptr;

if (Node->left == nullptr)

return Node;

return findMin(Node->left);

}

AvlNode\* findMax(AvlNode\* Node) const

{

if (Node != nullptr)

while (Node->right != nullptr)

Node = Node->right;

return Node;

}

void insert(const Object& Obj, AvlNode\*& Node)

{

if (Node == nullptr)

Node = new AvlNode{ Obj, nullptr, nullptr };

else if (Obj < Node->element)

insert(Obj, Node->left);

else if (Node->element < Obj)

insert(Obj, Node->right);

else

;

}

void insert(Object&& Obj, AvlNode\*& Node)

{

if (Node == nullptr)

Node = new AvlNode{ std::move(Obj), nullptr, nullptr };

else if (Obj < Node->element)

insert(std::move(Obj), Node->left);

else if (Node->element < Obj)

insert(std::move(Obj), Node->right);

else

;

}

void remove(const Object& Obj, AvlNode\*& Node)

{

if (Node == nullptr)

return;

if (Obj < Node->element)

remove(Obj, Node->left);

else if (Node->element < Obj)

remove(Obj, Node->right);

else if (Node->left != nullptr && Node->right != nullptr)

{

Node->element = findMin(Node->right)->element;

remove(Node->element, Node->right);

}

else

{

AvlNode\* oldNode = Node;

Node = (Node->left != nullptr) ? Node->left : Node->right;

delete oldNode;

}

}

void makeEmpty(AvlNode\*& Node)

{

if (Node != nullptr)

{

makeEmpty(Node->left);

makeEmpty(Node->right);

delete Node;

}

Node = nullptr;

}

~AVLTree() {

makeEmpty();

}

AVLTree(const AVLTree& rhs) : root{ nullptr }

{

root = clone(rhs.root);

}

AvlNode\* clone(AvlNode\* Node) const

{

if (Node == nullptr)

return nullptr;

else

return new AvlNode{ Node->element, clone(Node->left), clone(Node->right) };

}

int height(AvlNode\* Node) const

{

return Node == nullptr ? -1 : Node->height;

}

void printTree(ostream& out = cout) const

{

printTree(root, out);

}

void printTree(AvlNode\* Node, ostream& out) const

{

if (Node != nullptr)

{

printTree(Node->left, out);

out << Node->element << endl;

printTree(Node->right, out);

}

}

};

enum Color { RED, BLACK };

template <typename Object>

class RedBlackTree

{

public:

struct RedBlackNode

{

Object element;

RedBlackNode\* left;

RedBlackNode\* right;

int color;

RedBlackNode(const Object& theElement = Object{ },

RedBlackNode\* lt = nullptr, RedBlackNode\* rt = nullptr,

int c = BLACK)

: element{ theElement }, left{ lt }, right{ rt }, color{ c } { }

RedBlackNode(Object&& theElement, RedBlackNode\* lt = nullptr,

RedBlackNode\* rt = nullptr, int c = BLACK)

: element{ std::move(theElement) }, left{ lt }, right{ rt }, color{ c } { }

};

RedBlackNode\* header;

RedBlackNode\* nullNode;

RedBlackNode\* current;

RedBlackNode\* parent;

RedBlackNode\* grand;

RedBlackNode\* great;

explicit RedBlackTree(const Object& negInf)

{

nullNode = new RedBlackNode;

nullNode->left = nullNode->right = nullNode;

header = new RedBlackNode{ negInf };

header->left = header->right = nullNode;

}

RedBlackTree(const RedBlackTree& rhs)

{

nullNode = new RedBlackNode;

nullNode->left = nullNode->right = nullNode;

header = new RedBlackNode{ rhs.header->element };

header->left = nullNode;

header->right = clone(rhs.header->right);

}

RedBlackNode\* clone(RedBlackNode\* t) const

{

if (t == t->left)

return nullNode;

else

return new RedBlackNode{ t->element, clone(t->left),

clone(t->right), t->color };

}

RedBlackNode\* rotate(const Object& item, RedBlackNode\* theParent)

{

if (item < theParent->element)

{

item < theParent->left->element ?

rotateWithLeftChild(theParent->left) :

rotateWithRightChild(theParent->left);

return theParent->left;

}

else

{

item < theParent->right->element ?

rotateWithLeftChild(theParent->right) :

rotateWithRightChild(theParent->right);

return theParent->right;

}

}

void handleReorient(const Object& item)

{

// Do the color flip

current->color = RED;

current->left->color = BLACK;

current->right->color = BLACK;

if (parent->color == RED){

grand->color = RED;

if (item < grand->element != item < parent->element)

parent = rotate(item, grand); // Start dbl rotate

current = rotate(item, great);

current->color = BLACK;

}

header->right->color = BLACK; // Make root black}

}

void insert(const Object& x)

{

current = parent = grand = header;

nullNode->element = x;

while (current->element != x)

{

great = grand; grand = parent; parent = current;

current = x < current->element ? current->left : current->right;

// Check if two red children; fix if so

if (current->left->color == RED && current->right->color == RED)

handleReorient(x);

}

// Insertion fails if already present

if (current != nullNode)

return;

current = new RedBlackNode{ x, nullNode, nullNode };

// Attach to parent

if (x < parent->element)

parent->left = current;

else

parent->right = current;

handleReorient(x);

}

};