#include <iostream> // (1/5)

#include <memory>

#include <vector>

#include <algorithm>

#include <iomanip>

#include <ctime>

using namespace std;

template <typename T>

class TreapNode

{

private:

    T key;

    int priority;

    shared\_ptr<TreapNode<T>> left;

    shared\_ptr<TreapNode<T>> right;

public:

    TreapNode() : key(T(0)), priority(rand() % 100), left(nullptr), right(nullptr) {}

    TreapNode(T key) : key(key), priority(rand() % 100), left(nullptr), right(nullptr) {}

    TreapNode(T key, int priority) : key(key), priority(priority%100), left(nullptr), right(nullptr) {}

    ~TreapNode()

    {

        cout << "Key " << key << " destroyed." << endl;

    }

    T getKey() const

    {

        return key;

    }

    int getPriority() const

    {

        return priority;

    }

    void right\_rotation(shared\_ptr<TreapNode<T>> &x)

    {

        auto y = x->left; // auto y = x-> left;

        auto g = y->right;

        auto b = x->right;

        x->left = g;

        y->right = x;

        x = y;

    }

    void left\_rotation(shared\_ptr<TreapNode<T>> &y) // (2/5)

    {

        auto x = y->right;

        auto g = x->left;

        auto b = x->right;

        x->left = y;

        y->right = g;

        y = x;

    }

    void insert(shared\_ptr<TreapNode<T>> &root, T key)

    {

        if (!root)

        {

            root = make\_shared<TreapNode<T>>(key);

            return;

        }

        insert(key <= root->key ? root->left : root->right, key);

        if (root->left && root->left->priority > root->priority)

        {

            right\_rotation(root);

        }

        if (root->right && root->right->priority > root->priority)

        {

            left\_rotation(root);

        }

    }

    void insert(shared\_ptr<TreapNode<T>> &root, T key, int priority)

    {

        if (root == nullptr)

        {

            root = make\_shared<TreapNode<T>>(key, priority);

            return;

        }

        insert(key <= root->key ? root->left : root->right, key);

        if (root->left && root->left->priority > root->priority)

        {

            right\_rotation(root);

        }

        if (root->right && root->right->priority > root->priority)

        {

            left\_rotation(root);

        }

    }

    bool remove(shared\_ptr<TreapNode<T>> &root, T key) // (3/5)

    {

        if (!root)

        {

            return false;

        }

        if (key < root->key)

        {

            return remove(root->left, key);

        }

        if (key > root->key)

        {

            return remove(root->right, key);

        }

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

        {

            root.reset();

        }

        else if (!root->left || !root->right)

        {

            shared\_ptr<TreapNode<T>> child = (root->left) ? root->left : root->right;

            root = child;

        }

        else

        {

            if (root->left->priority < root->right->priority)

            {

                left\_rotation(root);

                remove(root->left, key);

            }

            else

            {

                right\_rotation(root);

                remove(root->right, key);

            }

        }

        return true;

    }

    void split(shared\_ptr<TreapNode<T>> &t, T key, shared\_ptr<TreapNode<T>> &l, shared\_ptr<TreapNode<T>> &r)

    {

        int inf = INT\_MAX;

        insert(t, key, inf);

        l = t->left;

        r = t->right;

    }

    shared\_ptr<TreapNode<T>> search(shared\_ptr<TreapNode<T>> &root, T key) // (4/5)

    {

        if (!root || root->key == key)

        {

            return root;

        }

        if (root->key < key)

        {

            return search(root->right, key);

        }

        if (root->key > key)

        {

            return search(root->left, key);

        }

    }

    void heapify(shared\_ptr<TreapNode<T>> t)

    {

        if (!t)

        {

            return;

        }

        auto max = t;

        if (t->left && t->left->priority > max->priority)

        {

            max = t->left;

        }

        if (t->right && t->right->priority > max->priority)

        {

            max = t->right;

        }

        if (max != t)

        {

            swap(t->priority, max->priority);

            heapify(max);

        }

    }

    shared\_ptr<TreapNode<T>> build(vector<T> &v, int l, int r)

    {

        sort(v.begin() + l, v.begin() + r + 1);

        if (l > r)

            return nullptr;

        int m = (l + r) / 2;

        auto t = make\_shared<TreapNode<T>>(v[m]); //

        t->left = build(v, l, m - 1);

        t->right = build(v, m + 1, r);

        heapify(t);

        return t;

    }

    int height(shared\_ptr<TreapNode<T>> &node) // (5/5)

    {

        if (!node)

            return 0;

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

    }

};

int main()

{

    srand(time(nullptr));

    shared\_ptr<TreapNode<int>> t = nullptr;

    vector<pair<int, int>> keys = {

        {5, 90}, {3, 70}, {1, 50}, {4, 60},

        {2, 55}, {9, 85}, {7, 65}, {8, 75},

        {6, 60}, {0, 45}, {10, 80}, {11, 70}

    };

    for (const auto& [key, priority] : keys) {

    t->insert(t, key, priority);

}

    t->print\_treap\_2D(t);

    cout << endl;

    cout << "Inserting 20" << endl;

    t->insert(t, 20, 95);

    t->print\_treap\_2D(t);

    cout << endl;

    cout << "Inserting 15" << endl;

    t->insert(t, 15, 78);

    t->print\_treap\_2D(t);

    cout << endl;

    cout << "Removing 5" << endl;

    t->remove(t, 5);

    t->print\_treap\_2D(t);

    cout << endl;

    cout << "Searching for 10" << endl;

    auto found = t->search(t, 10);

    if (found)

    {

        cout << "Found: " << found->getKey() << "(" << found->getPriority() << ")" << endl;

    }

    else

    {

        cout << "Not found!" << endl;

    }

    return 0;

}