**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра САПР**

**ОТЧЕТ**

**по лабораторной работе № 1**

**по дисциплине «Алгоритмы и структуры данных»**

**Вариант №1**

|  |  |  |
| --- | --- | --- |
| Студент гр. 8301 | **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** | Пискунович К.В. |
| Преподаватель | **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** | Тутуева А.В. |

Санкт-Петербург

2020

**Цель работы.**

Реализовать контейнер map на основе красно-черного дерева и функции к нему.

**Описание реализуемого класса и методов**

*Map – основной класс, в котором будут реализовываться функции.*

*Tree – класс, который будет реализовывать красно-черное дерево.*

*Node – класс элемента дерева.*

* insert(TKey, TValue) - добавление элемента с ключом и значением.
* remove(TKey) - удаление элемента дерева по ключу.
* find(TKey) - поиск элемента.
* clear() - очищение ассоциативного массива.
* get\_keys() - возвращает список ключей
* get\_values() - возвращает список значений
* print() - вывод дерева в консоль

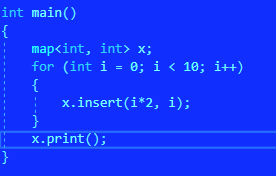
**Оценка временной сложности каждого метода.**

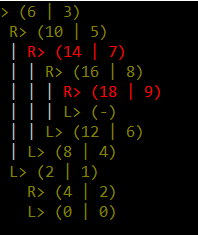
* insert(TKey, TValue) – O(logN)
* remove(TKey) – O(logN)
* find(TKey) – O(logN)
* clear() – O(N)
* get\_keys() – O(N)
* get\_values() – O(N)
* print() – O(N)

**Описание реализованных Unit-тестов.**

* insert\_and\_find\_test – проверяет нахождение и вставку элемента в контейнер. Проверяются две функции, так как результат теста зависит от работы обеих функций.
* remove\_test – проверяет удаление элемента из контейнера.
* clear\_test – проверяет функцию полного очищения контейнера.
* get\_keys\_test – проверяет функцию возвращения списка List ключей дерева.
* get\_values\_test – проверяет функцию возвращения списка List значений дерева.

**Пример работы программы**





**Листинг.**

Map.h:

#pragma once

#include <Windows.h>

#include <exception>

using namespace std;

#define MAP\_FUNC\_RETURNS\_NODE\_POINTER map<TKey, TValue>::Tree::Node\* map<TKey, TValue>

#define TREE\_FUNC\_RETURNS\_NODE\_POINTER map<TKey, TValue>::Tree::Node\* map<TKey, TValue>::Tree

/\* Red-Black tree description \*/

typedef enum { BLACK, RED } nodeColor;

#pragma region LIST

template <typename T>

class List

{

private:

class Node

{

public:

Node\* next = nullptr;

T data;

};

Node\* end = nullptr;

Node\* current = nullptr;

Node\* start = nullptr;

public:

void newElement(T element)

{

if (!end)

{

end = start = current = new Node;

end->data = element;

}

else

{

end->next = new Node;

end = end->next;

end->data = element;

}

}

T next()

{

if (current)

{

T value = current->data;

current = current->next;

return value;

}

}

bool isCurrent() {

return current ? true : false;

}

};

#pragma endregion

#pragma region MAP\_CLASS

template <typename TKey, typename TValue>

class map {

private:

class Tree;

Tree\* tree\_of\_elements;

public:

map()

{

tree\_of\_elements = new Tree;

}

typename Tree::Node\* insert(TKey, TValue); // добавление элемента с ключом и значением

void remove(TKey); // удаление элемента дерева по ключу

typename Tree::Node\* find(TKey); // поиск элемента

void clear(); // очищение ассоциативного массива

List<TKey> get\_keys(); // возвращает список ключей

List<TValue> get\_values(); // возвращает список значений

void print(); // вывод дерева в консоль

};

#pragma endregion

#pragma region RED-BLACK\_TREE\_CLASS

template <typename TKey, typename TValue>

class map<TKey, TValue>::Tree {

private:

friend class map<TKey, TValue>;

class Node {

public:

Node\* right = nullptr;

Node\* left = nullptr;

Node\* parent = nullptr;

pair <TKey, TValue> data;

nodeColor color = BLACK;

};

void insertFixup(Node\*);

void deleteFixup(Node\*);

void rotate\_left(Node\*);

void clear(Node \*);

void get\_values(typename Node\*, List<TValue>&);

void get\_keys(typename Node \*, List<TKey> &);

void print(Node\*, string);

void rotate\_right(Node\*);

public:

typename Node\* insert(TKey, TValue);

void deleteNode(Node \*);

Node\* find(TKey);

Node\* root = nullptr;

};

#pragma endregion

#pragma region FUNCTIONS\_OF\_MAP\_REALIZATION

template <typename TKey, typename TValue>

void map<TKey, TValue>::remove(TKey key)

{

auto node = find(key);

if (!node) throw exception("There is no element in the tree");

tree\_of\_elements->deleteNode(node);

}

template <typename TKey, typename TValue>

List<TValue> map<TKey, TValue>::get\_values() {

List<TValue> list;

tree\_of\_elements->get\_values(tree\_of\_elements->root, list);

return list;

}

template <typename TKey, typename TValue>

List<TKey> map<TKey, TValue>::get\_keys() {

List<TKey> list;

tree\_of\_elements->get\_keys(tree\_of\_elements->root, list);

return list;

}

template <typename TKey, typename TValue>

typename MAP\_FUNC\_RETURNS\_NODE\_POINTER::insert(TKey key, TValue value)

{

return tree\_of\_elements->insert(key,value);

}

template <typename TKey, typename TValue>

void map<TKey, TValue>::print()

{

tree\_of\_elements->print(tree\_of\_elements->root, "");

}

template <typename TKey, typename TValue>

typename MAP\_FUNC\_RETURNS\_NODE\_POINTER::find(TKey key)

{

return tree\_of\_elements->find(key);

}

template <typename TKey, typename TValue>

void map<TKey, TValue>::clear()

{

tree\_of\_elements->clear(tree\_of\_elements->root);

}

#pragma endregion

#pragma region FUNCTIONS\_OF\_TREE\_REALIZATION

template <typename TKey, typename TValue>

void map<TKey, TValue>::Tree::deleteFixup(Node\* node) {

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* maintain Red-Black tree balance \*

\* after deleting node x \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

while (node != root && node->color == BLACK) {

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

Node\* w = node->parent->right;

if (w->color == RED) {

w->color = BLACK;

node->parent->color = RED;

rotate\_left(node->parent);

w = node->parent->right;

}

if (w->left->color == BLACK && w->right->color == BLACK) {

w->color = RED;

node = node->parent;

}

else {

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

w->left->color = BLACK;

w->color = RED;

rotate\_right(w);

w = node->parent->right;

}

w->color = node->parent->color;

node->parent->color = BLACK;

w->right->color = BLACK;

rotate\_left(node->parent);

node = root;

}

}

else {

Node\* w = node->parent->left;

if (w->color == RED) {

w->color = BLACK;

node->parent->color = RED;

rotate\_right(node->parent);

w = node->parent->left;

}

if (w->right->color == BLACK && w->left->color == BLACK) {

w->color = RED;

node = node->parent;

}

else {

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

w->right->color = BLACK;

w->color = RED;

rotate\_left(w);

w = node->parent->left;

}

w->color = node->parent->color;

node->parent->color = BLACK;

w->left->color = BLACK;

rotate\_right(node->parent);

node = root;

}

}

}

if(node) node->color = BLACK;

}

template <typename TKey, typename TValue>

void map<TKey, TValue>::Tree::deleteNode(Node\* node) {

Node\* x, \* y;

if (!node || node == nullptr) return;

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

/\* y has a NIL node as a child \*/

y = node;

}

else {

/\* find tree successor with a NIL node as a child \*/

y = node->right;

while (y->left != nullptr) y = y->left;

}

/\* x is y's only child \*/

if (y->left != nullptr)

x = y->left;

else

x = y->right;

/\* remove y from the parent chain \*/

if(x) x->parent = y->parent;

if (y->parent)

if (y == y->parent->left)

y->parent->left = x;

else

y->parent->right = x;

else

root = x;

if (y != node) node->data = y->data;

if (y->color == BLACK)

deleteFixup(x);

delete y;

}

template <typename TKey, typename TValue>

void map<TKey, TValue>::Tree::get\_keys(typename Tree::Node\* node, List<TKey>& list)

{

if (!root) return;

if (node->left) get\_keys(node->left, list);

if (node->right) get\_keys(node->right, list);

list.newElement(node->data.first);

}

template <typename TKey, typename TValue>

void map<TKey, TValue>::Tree::get\_values(typename Tree::Node\* node, List<TValue>& list)

{

if (!root) return;

if (node->left) get\_values(node->left, list);

if (node->right) get\_values(node->right, list);

list.newElement(node->data.second);

}

template <typename TKey, typename TValue>

void map<TKey, TValue>::Tree::clear(typename Tree::Node\* node)

{

if (node->left) clear(node->left);

if (node->right) clear(node->right);

if (node == root) root = nullptr;

delete node;

}

template <typename TKey, typename TValue>

void map<TKey, TValue>::Tree::print(typename Tree::Node\* root, string str)

{

if (!root) return;

HANDLE hConsole = GetStdHandle(STD\_OUTPUT\_HANDLE);

if (root == this->root)

{

SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 6));

cout << "> (" << root->data.first << " | " << root->data.second << ")" << endl;

SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 7));

str += " ";

}

if (root->right) {

string \_str = str;

cout << \_str;

if (root->right->color == BLACK)

SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 6));

else SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 12));

cout << "R> (" << root->right->data.first << " | " << root->right->data.second << ")" << endl;

SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 7));

\_str += "| ";

print(root->right, \_str);

}

else if (root->left) {

cout << str;

SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 6));

cout << "R> (-)" << endl;

SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 7));

}

if (root->left) {

string \_str = str;

cout << \_str;

if (root->left->color == BLACK)

SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 6));

else SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 12));

cout << "L> (" << root->left->data.first << " | " << root->left->data.second << ")" << endl;

SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 7));

\_str += " ";

print(root->left, \_str);

}

else if (root->right) {

cout << str;

SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 6));

cout << "L> (-)" << endl;

SetConsoleTextAttribute(hConsole, (WORD)((0 << 4) | 7));

}

}

template <typename TKey, typename TValue>

void map<TKey, TValue>::Tree::rotate\_left(Node\* node)

{

Node\* y = node->right;

node->right = y->left;

if (y->left != nullptr) y->left->parent = node;

if (y != nullptr) y->parent = node->parent;

if (node->parent) {

if (node == node->parent->left)

node->parent->left = y;

else

node->parent->right = y;

}

else {

root = y;

}

y->left = node;

if (node != nullptr) node->parent = y;

}

template <typename TKey, typename TValue>

void map<TKey, TValue>::Tree::rotate\_right(Node\* node) {

Node\* y = node->left;

node->left = y->right;

if (y->right != nullptr) y->right->parent = node;

if (y != nullptr) y->parent = node->parent;

if (node->parent) {

if (node == node->parent->right)

node->parent->right = y;

else

node->parent->left = y;

}

else {

root = y;

}

y->right = node;

if (node != nullptr) node->parent = y;

}

template <typename TKey, typename TValue>

typename TREE\_FUNC\_RETURNS\_NODE\_POINTER::find(TKey key)

{

Node\* current = root;

while (current != nullptr)

if (key == current->data.first)

return current;

else

{

current = key < current->data.first ? current->left : current->right;

}

return 0;

}

template <typename TKey, typename TValue>

void map<TKey, TValue>::

Tree::insertFixup(Node\* element)

{

while (element != root && element->parent->color == RED) {

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

Node\* y = element->parent->parent->right;

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

/\* uncle is RED \*/

element->parent->color = BLACK;

y->color = BLACK;

element->parent->parent->color = RED;

element = element->parent->parent;

}

else {

/\* uncle is BLACK \*/

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

/\* make x a left child \*/

element = element->parent;

rotate\_left(element);

}

/\* recolor and rotate \*/

element->parent->color = BLACK;

element->parent->parent->color = RED;

rotate\_right(element->parent->parent);

}

}

else {

/\* mirror image of above code \*/

Node\* y = element->parent->parent->left;

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

/\* uncle is RED \*/

element->parent->color = BLACK;

y->color = BLACK;

element->parent->parent->color = RED;

element = element->parent->parent;

}

else {

/\* uncle is BLACK \*/

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

element = element->parent;

rotate\_right(element);

}

element->parent->color = BLACK;

element->parent->parent->color = RED;

rotate\_left(element->parent->parent);

}

}

}

root->color = BLACK;

}

template <typename TKey, typename TValue>

typename TREE\_FUNC\_RETURNS\_NODE\_POINTER::insert(TKey key, TValue value)

{

Node \*current, \*newNode, \*parent;

current = root;

parent = 0;

while (current != nullptr) {

if (key == current->data.first) return current;

parent = current;

current = key < current->data.first ? current->left : current->right;

}

/\* setup new node \*/

newNode = new Node;

newNode->data = make\_pair(key, value);

newNode->parent = parent;

newNode->color = RED;

/\* insert node in tree \*/

if (parent) {

if (key < parent->data.first)

parent->left = newNode;

else

parent->right = newNode;

}

else {

root = newNode;

}

insertFixup(newNode);

return newNode;

}

#pragma endregion

Lab1Tests.cpp (Unit-тесты):

#include "CppUnitTest.h"

#include "../lab1/map.h"

#include <stdexcept>

using namespace std;

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace Lab1Tests

{

TEST\_CLASS(Lab1Tests)

{

public:

TEST\_METHOD(insert\_and\_find\_test)

{

map<int, int> card;

bool before = card.find(5);

card.insert(5, 1);

bool after = card.find(5);

Assert::AreEqual(!before, after);

}

TEST\_METHOD(remove\_test)

{

map<int, int> card;

card.insert(5, 1);

bool before = card.find(5);

card.remove(5);

bool after = card.find(5);

Assert::AreEqual(before, !after);

}

TEST\_METHOD(clear\_test)

{

map<int, int> card;

card.insert(5, 1);

card.insert(6, 2);

card.clear();

Assert::AreEqual(!card.find(5), !card.find(6));

}

TEST\_METHOD(get\_keys\_test)

{

map<int, int> card;

card.insert(5, 1);

card.insert(6, 2);

card.insert(7, 3);

List<int> list = card.get\_keys();

int sum\_of\_keys = 0;

while (list.isCurrent())

sum\_of\_keys += list.next();

Assert::IsTrue(sum\_of\_keys == 18);

}

TEST\_METHOD(get\_values\_test)

{

map<int, int> card;

card.insert(5, 1);

card.insert(6, 2);

card.insert(7, 3);

List<int> list = card.get\_values();

int sum\_of\_values = 0;

while (list.isCurrent())

sum\_of\_values += list.next();

Assert::IsTrue(sum\_of\_values == 6);

}

};

}