**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»**

**Тема: «Ассоциативный массив»**

Студентка гр. 8302 \_\_\_\_\_\_\_\_\_\_\_\_\_ Колмакова С.П.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_ Тутуева А.В.

Санкт-Петербург

2020

Цель работы

Реализовать шаблонный ассоциативный массив (map) на основе красно-черного дерева.

Описание реализуемого класса

Класс Map состоит из вложенного класса Node с полями: T key (ключ, по которому хранится значение) и T1 value (значение, которое хранится по определённому ключу), bool color (цвет ячейки, необходимый для дальнейшей проверки сбалансированности), Node\* parent (указатель на родителя ячейки), Node\* left и Node\* right (указатели на левого и правого потомков), а также собственные поля Node\* Top (вершина дерева), Node\* TNULL (обозначение пустого листа и родителя корня).

Реализованный мной класс Map основан на такой структуре данных как красно-чёрное дерево. Выбранная структура данных является самобалансирующимся бинарным деревом, оно имеет цвета (красный и чёрный) и 5 правил. При несоблюдении хотя бы одного из них при вставке или удалении, будет требоваться перебалансировка. Заметим, что key и value являются шаблонными, что позволяет нам хранить ключ и значение любого типа данных.

Класс содержит следущие методы:

• Конструктор – реализованы конструкторы по умолчанию для вложенного класса Node и для самого класса Map.

• Деструктор – реализован деструктор, вызывающий метод *clear()*.

• *insert(T key,T1 value)* – функция добавления элемента в дерево по ключу. Добавление происходит точно так же, как и в бинарном дереве. Дальше следует проверка на соблюдение всех свойств, при несоблюдении хотя бы одного происходит перебалансировка.

• *remove(T key)* – функция удаления элемента по ключу.

• *find(T key)* – функция получения значения по ключу.

• *clear()* – функция, по одному удаляющая элементы при постфиксном обходе дерева.

• *get\_keys()* – функция, возвращающая список ключей.

• *get\_values()* – функция, возвращающая список значений.

• *print()*– функция вывода дерева.

Оценка временной сложности алгоритмов

• *remove(T key)* – O(log n)

• *insert(T key,T1 value)* – O(log n)

• *find(T key)* – O(log n)

• *clear()* – O(n)

• *get\_keys()* – O(n)

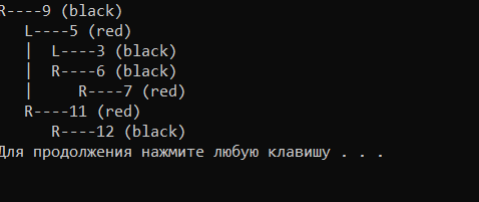
• *get\_values()* – O(n)

• *print()* – O(n)

Описание реализованных unit-тестов

В тестах, реализованных для класса Map мы протестировали добавление элемента в дерево с помощью функции *insert()* и удаление с помощью функции remove(), и проверили их c помощью функции find(), которая возвращает нам значение по ключу или бросает исключение. Так же мои unit-тесты затрагивают такие методы как get\_keys() и get\_values(), которые возвращают списки ключей и значений, и метод очистки дерева clear().

Пример работы программы

**Пример**  


Листинг

**Map.h**

#define COLOR\_RED 1

#define COLOR\_BLACK 0

#include"List.h"

using namespace std;

template<typename T, typename T1>

class Map {

public:

class Node

{

public:

Node(bool color = COLOR\_RED, T key = T(), Node\* parent = NULL, Node\* left = NULL, Node\* right = NULL,

T1 value = T1()) :color(color), key(key), parent(parent), left(left), right(right), value(value) {}

T key;

T1 value;

bool color;

Node\* parent;

Node\* left;

Node\* right;

};

~Map()

{

if (this->Root != NULL)

this->clear();

Root = NULL;

delete TNULL;

TNULL = NULL;

}

Map(Node\* Root = NULL, Node\* TNULL = new Node(0)) :Root(Root), TNULL(TNULL) {}

void printTree()

{

if (Root)

{

print\_helper(this->Root, "", true);

}

else throw std::out\_of\_range("Tree is empty!");

}

void insert(T key, T1 value)

{

if (this->Root != NULL)

{

Node\* node = NULL;

Node\* parent = NULL;

/\* Search leaf for new element \*/

for (node = this->Root; node != TNULL; )

{

parent = node;

if (key < node->key)

node = node->left;

else if (key > node->key)

node = node->right;

else if (key == node->key)

throw std::out\_of\_range("key is repeated");

}

node = new Node(COLOR\_RED, key, TNULL, TNULL, TNULL, value);

node->parent = parent;

if (parent != TNULL)

{

if (key<parent->key)

parent->left = node;

else

parent->right = node;

}

insert\_fix(node);

}

else

{

this->Root = new Node(COLOR\_BLACK, key, TNULL, TNULL, TNULL, value);

}

}

List<T>\* get\_keys()

{

List<T>\* list = new List<T>();

this->list\_key\_or\_value(1, list);

return list;

}

List<T1>\* get\_values()

{

List<T1>\* list = new List<T1>();

this->list\_key\_or\_value(2, list);

return list;

}

T1 find(T key)

{

Node\* node = Root;

while (node != TNULL && node->key != key)

{

if (node->key > key)

node = node->left;

else

if (node->key < key)

node = node->right;

}

if (node != TNULL)

return node->value;

else

throw std::out\_of\_range("Key is missing");

}

void remove(T key)

{

this->delete\_node(this->find\_key(key));

}

void clear()

{

this->clear\_tree(this->Root);

this->Root = NULL;

}

private:

Node\* Root;

Node\* TNULL;

//delete functions

void delete\_node(Node\* find\_node)

{

Node\* node\_with\_fix, \* cur\_for\_change;

cur\_for\_change = find\_node;

bool cur\_for\_change\_original\_color = cur\_for\_change->color;

if (find\_node->left == TNULL)

{

node\_with\_fix = find\_node->right;

transplant(find\_node, find\_node->right);

}

else if (find\_node->right == TNULL)

{

node\_with\_fix = find\_node->left;

transplant(find\_node, find\_node->left);

}

else

{

cur\_for\_change = minimum(find\_node->right);

cur\_for\_change\_original\_color = cur\_for\_change->color;

node\_with\_fix = cur\_for\_change->right;

if (cur\_for\_change->parent == find\_node)

{

node\_with\_fix->parent = cur\_for\_change;

}

else

{

transplant(cur\_for\_change, cur\_for\_change->right);

cur\_for\_change->right = find\_node->right;

cur\_for\_change->right->parent = cur\_for\_change;

}

transplant(find\_node, cur\_for\_change);

cur\_for\_change->left = find\_node->left;

cur\_for\_change->left->parent = cur\_for\_change;

cur\_for\_change->color = find\_node->color;

}

delete find\_node;

if (cur\_for\_change\_original\_color == COLOR\_RED)

{

this->delete\_fix(node\_with\_fix);

}

}

//swap links(parent and other) for rotate

void transplant(Node\* current, Node\* current1)

{

if (current->parent == TNULL)

{

Root = current1;

}

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

{

current->parent->left = current1;

}

else

{

current->parent->right = current1;

}

current1->parent = current->parent;

}

void clear\_tree(Node\* tree)

{

if (tree != TNULL)

{

clear\_tree(tree->left);

clear\_tree(tree->right);

delete tree;

}

}

//find functions

Node\* minimum(Node\* node)

{

while (node->left != TNULL)

{

node = node->left;

}

return node;

}

Node\* maximum(Node\* node)

{

while (node->right != TNULL)

{

node = node->right;

}

return node;

}

Node\* grandparent(Node\* current)

{

if ((current != TNULL) && (current->parent != TNULL))

return current->parent->parent;

else

return TNULL;

}

Node\* uncle(Node\* current)

{

Node\* current1 = grandparent(current);

if (current1 == TNULL)

return TNULL; // No grandparent means no uncle

if (current->parent == current1->left)

return current1->right;

else

return current1->left;

}

Node\* sibling(Node\* n)

{

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

return n->parent->right;

else

return n->parent->left;

}

Node\* find\_key(T key)

{

Node\* node = this->Root;

while (node != TNULL && node->key != key)

{

if (node->key > key)

node = node->left;

else

if (node->key < key)

node = node->right;

}

if (node != TNULL)

return node;

else

throw std::out\_of\_range("Key is missing");

}

//all print function

void print\_helper(Node\* root, string indent, bool last)

{

if (root != TNULL)

{

cout << indent;

if (last)

{

cout << "R----";

indent += " ";

}

else

{

cout << "L----";

indent += "| ";

}

string sColor = !root->color ? "black" : "red";

cout << root->key << " (" << sColor << ")" << endl;

print\_helper(root->left, indent, false);

print\_helper(root->right, indent, true);

}

}

void list\_key\_or\_value(int mode, List<T>\* list)

{

if (this->Root != TNULL)

this->key\_or\_value(Root, list, mode);

else

throw std::out\_of\_range("Tree empty!");

}

void key\_or\_value(Node\* tree, List<T>\* list, int mode)

{

if (tree != TNULL)

{

key\_or\_value(tree->left, list, mode);

if (mode == 1)

list->push\_back(tree->key);

else

list->push\_back(tree->value);

key\_or\_value(tree->right, list, mode);

}

}

//fix

void insert\_fix(Node\* node)

{

Node\* uncle;

/\* Current node is COLOR\_RED \*/

while (node != this->Root && node->parent->color == COLOR\_RED)//

{

/\* node in left tree of grandfather \*/

if (node->parent == this->grandparent(node)->left)//

{

/\* node in left tree of grandfather \*/

uncle = this->uncle(node);

if (uncle->color == COLOR\_RED)

{

/\* Case 1 - uncle is COLOR\_RED \*/

node->parent->color = COLOR\_BLACK;

uncle->color = COLOR\_BLACK;

this->grandparent(node)->color = COLOR\_RED;

node = this->grandparent(node);

}

else {

/\* Cases 2 & 3 - uncle is COLOR\_BLACK \*/

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

{

/\*Reduce case 2 to case 3 \*/

node = node->parent;

this->left\_rotate(node);

}

/\* Case 3 \*/

node->parent->color = COLOR\_BLACK;

this->grandparent(node)->color = COLOR\_RED;

this->right\_rotate(this->grandparent(node));

}

}

else {

/\* Node in right tree of grandfather \*/

uncle = this->uncle(node);

if (uncle->color == COLOR\_RED)

{

/\* Uncle is COLOR\_RED \*/

node->parent->color = COLOR\_BLACK;

uncle->color = COLOR\_BLACK;

this->grandparent(node)->color = COLOR\_RED;

node = this->grandparent(node);

}

else {

/\* Uncle is COLOR\_BLACK \*/

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

{

node = node->parent;

this->right\_rotate(node);

}

node->parent->color = COLOR\_BLACK;

this->grandparent(node)->color = COLOR\_RED;

this->left\_rotate(this->grandparent(node));

}

}

}

this->Root->color = COLOR\_BLACK;

}

void delete\_fix(Node\* node)

{

Node\* sibling;

while (node != this->Root && node->color == COLOR\_BLACK)//

{

sibling = this->sibling(node);

if (sibling != TNULL)

{

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

{

if (sibling->color == COLOR\_BLACK)

{

node->parent->color = COLOR\_BLACK;

sibling->color = COLOR\_RED;

this->left\_rotate(node->parent);

sibling = this->sibling(node);

}

if (sibling->left->color == COLOR\_RED && sibling->right->color == COLOR\_RED)

{

sibling->color = COLOR\_BLACK;

node = node->parent;

}

else

{

if (sibling->right->color == COLOR\_RED)

{

sibling->left->color = COLOR\_RED;

sibling->color = COLOR\_BLACK;

this->left\_rotate(sibling);

sibling = this->sibling(node);

}

sibling->color = node->parent->color;

node->parent->color = COLOR\_RED;

sibling->right->color = COLOR\_RED;

this->left\_rotate(node->parent);

node = this->Root;

}

}

else

{

if (sibling->color == COLOR\_BLACK);

{

sibling->color = COLOR\_RED;

node->parent->color = COLOR\_BLACK;

this->right\_rotate(node->parent);

sibling = this->sibling(node);

}

if (sibling->left->color == COLOR\_RED && sibling->right->color)

{

sibling->color = COLOR\_BLACK;

node = node->parent;

}

else

{

if (sibling->left->color == COLOR\_RED)

{

sibling->right->color = COLOR\_RED;

sibling->color = COLOR\_BLACK;

this->left\_rotate(sibling);

sibling = this->sibling(node);

}

sibling->color = node->parent->color;

node->parent->color = COLOR\_RED;

sibling->left->color = COLOR\_RED;

this->right\_rotate(node->parent);

node = Root;

}

}

}

}

this->Root->color = COLOR\_BLACK;

}

//Rotates

void left\_rotate(Node\* node)

{

Node\* right = node->right;

/\* Create node->right link \*/

node->right = right->left;

if (right->left != TNULL)

right->left->parent = node;

/\* Create right->parent link \*/

if (right != TNULL)

right->parent = node->parent;

if (node->parent != TNULL)

{

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

node->parent->left = right;

else

node->parent->right = right;

}

else {

this->Root = right;

}

right->left = node;

if (node != TNULL)

node->parent = right;

}

void right\_rotate(Node\* node)

{

Node\* left = node->left;

/\* Create node->left link \*/

node->left = left->right;

if (left->right != TNULL)

left->right->parent = node;

/\* Create left->parent link \*/

if (left != TNULL)

left->parent = node->parent;

if (node->parent != TNULL)

{

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

node->parent->right = left;

else

node->parent->left = left;

}

else

{

this->Root = left;

}

left->right = node;

if (node != TNULL)

node->parent = left;

}

};

**RedBlackTreeTest.cpp**

#include "pch.h"

#include "CppUnitTest.h"

#include"../RedBlackTree/Map.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace RedBlackTreeTest

{

TEST\_CLASS(RedBlackTreeTest)

{

public:

TEST\_METHOD(GetKeys)

{

Map<int, int>\* tree = new Map<int, int>();

tree->insert(8, -1);

tree->insert(5, -2);

tree->insert(7, -3);

tree->insert(11, -4);

List<int>\* list = tree->get\_keys();

int sum = 0;

for (int i = 0; i < list->get\_size(); i++)

sum += list->at(i);

Assert::AreEqual(31, sum);

}

TEST\_METHOD(GetValues)

{

Map<int, int>\* tree = new Map<int, int>();

tree->insert(8, -1);

tree->insert(5, -2);

tree->insert(7, -3);

tree->insert(11, -4);

List<int>\* list = tree->get\_values();

int sum = 0;

for (int i = 0; i < list->get\_size(); i++)

sum += list->at(i);

Assert::AreEqual(-10, sum);

}

TEST\_METHOD(InsertAndFind)

{

Map<int, int>\* tree = new Map<int, int>();

tree->insert(8, -1);

tree->insert(5, -2);

tree->insert(7, -3);

tree->insert(11, -4);

Assert::AreEqual(tree->find(8), -1);

}

TEST\_METHOD(FindExeption)

{

try {

Map<int, int>\* tree = new Map<int, int>();

tree->insert(8, -1);

tree->insert(5, -2);

tree->insert(7, -3);

tree->insert(11, -4);

tree->find(29);

}

catch (std::out\_of\_range exc) {

Assert::AreEqual("Key is missing", exc.what());

}

}

TEST\_METHOD(Remove)

{

try {

Map<int, int>\* tree = new Map<int, int>();

tree->insert(8, -1);

tree->insert(5, -2);

tree->insert(7, -3);

tree->insert(11, -4);

tree->remove(8);

tree->find(8);

}

catch (std::out\_of\_range exc) {

Assert::AreEqual("Key is missing", exc.what());

}

}

TEST\_METHOD(ClearExeption)

{

try {

Map<int, int>\* tree = new Map<int, int>();

tree->insert(8, -1);

tree->insert(5, -2);

tree->insert(7, -3);

tree->insert(11, -4);

tree->clear();

tree->printTree();

}

catch (std::out\_of\_range exc) {

Assert::AreEqual("Tree is empty!", exc.what());

}

}

};

}

Вывод

В данной лабораторной работе я познакомилась с красно-черными деревьями и ассоциативными массивами и реализовала ассоциативный массив на основе красно-черного дерева.