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»

Студент гр. 8301 _____ Попурей Н.

Преподаватель _____ Тутуева А.В.

Санкт-Петербург

2020

Оглавление

1. Постановка задачи

Реализовать класс красно-чёрного дерева со следующими методами:

1. insert(T key,T1 value) – функция добавления элемента в дерево
2. remove(T key) – функция удаления элемента по ключу.
3. find(T key) – функция получения значения по ключу.
4. clear() – функция, по одному удаляющая элементы при постфиксном обходе дерева.
5. get_keys() – функция, возвращающая список ключей.
6. get_values() – функция, возвращающая список значений.
7. print()– функция вывода дерева.

2. Оценка временной сложности

1. remove(T key) – O(log n)
2. insert(T key,T1 value) – O(log n)
3. find(T key) – O(log n)
4. clear() – O(n)
5. get_keys() – O(n)
6. get_values() – O(n)
7. print() – O(n)

3. Описание реализованных юнит-тестов

В тестах, реализованных для класса Map мы протестировал добавление элемента в дерево с помощью функции insert() и удаление с помощью функции remove(), и проверил их с 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().

4. Пример работы

```
R----9 (black)
   L----5 (red)
   |  L----3 (black)
   |  R----6 (black)
   |     R----7 (red)
   R----11 (red)
      R----12 (black)
Для продолжения нажмите любую клавишу . . .
```

5. Листинг

HeaderTree.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(par
ent), 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;
    }
};
```

HeaderList.h

```
#include <iostream>
using namespace std;
template<typename T>
class List
{
private:
    class Node {
    public:
        Node(T data = T(), Node* Next = NULL)
        {
            this->data = data;
            this->Next = Next;
        }
        Node* Next;
        T data;
    };

public:
    void push_back(T obj)  // add to the end of the list
    {
        if (head != NULL)
        {
            this->tail->Next = new Node(obj);
            tail = tail->Next;
        }
        else {
            this->head = new Node(obj);
            this->tail = this->head;
        }
        Size++;
    }

    void insert(T obj, size_t k) // adding an item by index (insert before an ite
m that was previously available by this index)
    {
        if (k >= 0 && this->Size > k)
        {
            if (this->head != NULL)
```

```
            {
                if (k == 0)
                    this->push_front(obj);
                else
                    if (k == this->Size - 1)
                        this->push_back(obj);
                    else
                    {
                        Node* current = new Node;//to add an element
                        Node* current1 = head;//to search for the total element
                        for (int i = 0; i < k - 1; i++)
                        {
                            current1 = current1->Next;
                        }
                        current->data = obj;
                        current->Next = current1-
>Next;//points to the next element
                        current1->Next = current;
                        Size++;
                    }
            }
        }
        else {
            throw std::out_of_range("out_of_range");
        }
    }

    T at(size_t k) {// getting an item by index
        if (this->head != NULL && k >= 0 && k <= this->Size - 1)
        {
            if (k == 0)
                return this->head->data;
            else
                if (k == this->Size - 1)
                    return this->tail->data;
                else
                {
                    Node* current = head;
                    for (int i = 0; i < k; i++)
                    {
                        current = current->Next;
                    }
                    return current->data;
                }
        }
        else {
            throw std::out_of_range("out_of_range");
        }
    }

    void remove(int k) { // deleting an item by index
```

```
        if (head != NULL && k >= 0 && k <= Size - 1)
        {
            if (k == 0) this->pop_front();
            else
                if (k == this->Size - 1) this->pop_back();
                else
                    if (k != 0)
                    {
                        Node* current = head;
                        for (int i = 0; i < k - 1; i++) //go to the pre element
                        {
                            current = current->Next;
                        }

                        Node* current1 = current->Next;
                        current->Next = current->Next->Next;
                        delete current1;
                        Size--;
                    }
        }
        else
        {
            throw std::out_of_range("out_of_range");
        }
    }

    size_t get_size() { // getting the list size
        return Size;
    }

    void clear() // deleting all list items
    {
        if (head != NULL)
        {
            Node* current = head;
            while (head != NULL)
            {
                current = current->Next;
                delete head;
                head = current;
            }
            Size = 0;
        }
    }

public:
    List(Node* head = NULL, Node* tail = NULL, int Size = 0) :head(head), tail(ta
il), Size(Size) {}
    ~List()
    {
        if (head != NULL)
```

```
        {
            this->clear();
        }
    };

private:
    Node* head;
    Node* tail;
    int Size;
};
```

UnitTest1.cpp

```
#include "stdafx.h"
#include "CppUnitTest.h"
#include "../Lab1/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());
            }
```

```
        }
    };
}
```

6. Вывод:

Приобретены навыки работы с красно-чёрными деревьями и операциями с ними, а также юнит-тестами.