

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ  
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ им. ПЕТРА ВЕЛИКОГО

Институт Компьютерных наук и технологий

Высшая школа искусственного интеллекта

Направление 02.03.01 Математика и компьютерные науки

Отчет по лабораторной работе №6 по дисциплине «Теория графов»

Словарь на основе красно-черных деревьев. Словарь на основе  
Хеш-таблицы.

Группа: 3530201/00002

Студент: \_\_\_\_\_

Перекрестов Глеб Владимирович

Преподаватель: \_\_\_\_\_

Востров Алексей Владимирович

«\_\_\_\_\_» \_\_\_\_\_ 20\_\_г.

Санкт-Петербург – 2022

# Содержание

<b>Введение</b>	<b>3</b>
<b>1 Постановка задачи</b>	<b>4</b>
<b>2 Математическое описание</b>	<b>5</b>
2.1 Красно-черные деревья . . . . .	5
2.1.1 Определение красно-чёрного дерева . . . . .	5
2.1.2 Вставка . . . . .	5
2.1.3 Удаление . . . . .	6
2.1.4 Поиск . . . . .	7
2.2 Хеш-таблицы . . . . .	7
2.2.1 Определение Хеш-таблицы . . . . .	7
2.2.2 Хеш-функция . . . . .	7
2.2.3 Функции хеш-таблицы . . . . .	8
2.2.4 Разрешение коллизий . . . . .	8
<b>3 Особенности реализации</b>	<b>9</b>
3.1 Красно-чёрное дерево . . . . .	9
3.2 Хеш-таблица . . . . .	16
<b>4 Результаты работы программы</b>	<b>21</b>
<b>Заключение</b>	<b>29</b>
<b>Источники</b>	<b>30</b>

## Введение

В отчете содержится описание лабораторной работы по дисциплине «Теория графов». Лабораторная работа включает реализацию словаря на основе красно-черного дерева, а также реализацию словаря на основе Хеш-таблицы.

Работа была выполнена в среде Visual Studio 2019 на языке программирования C++.

# 1 Постановка задачи

В данной лабораторной работе требуется:

- Реализовать словарь на основе красно-черных деревьев и хеш-таблицы без использования готовых структур данных;
- Реализовать методы добавления, удаления и поиска элементов, заполнения словарей из текстового файла, методы очистки словарей.

## 2 Математическое описание

### 2.1 Красно-черные деревья

#### 2.1.1 Определение красно-чёрного дерева

Красно-черное дерево - это самобалансирующееся дерево поиска. Гарантирует логарифмический рост высоты дерева в зависимости от количества узлов. Бинарное дерево, баланс которого достигается за счет поддержания раскраски вершин в два цвета.

Корень красно-черного дерева окрашен в черный цвет. Узлы окрашены либо в черный, либо в красный цвета. Листьями объявляются NIL-узлы. Листья окрашены в чёрный цвет. Если узел красный, то оба его потомка черные.

#### 2.1.2 Вставка

1. Каждый элемент вставляется вместо листа, поэтому для выбора места вставки идём от корня до тех пор, пока указатель на следующего потомка не станет NIL.
2. Вставляем вместо него новый элемент красного цвета с NIL-потомками.
3. Проверяем балансировку.

В результате проверки балансировки могут быть произведены следующие действия:

1. Если текущий узел - корень дерева, то добавляемый узел перекрашивается в черный цвет.
2. Если «родитель» текущего узла черный, тогда свойства не нарушаются. Узел не перекрашивается.
3. Если «родитель» и «дядя» красные, в таком случае перекрасим «родителя» и «дядю» в чёрный цвет, а «дедушку» — в красный. При этом число чёрных вершин на любом пути от корня к листьям остаётся прежним. Теперь у текущего красного узла «родитель» черного цвета. Нарушение свойств красно-черного дерева возможно лишь в одном месте: вершина «дедушка» может иметь красного «родителя». Чтобы этого не произошло рекурсивно выполняется процедура первого случая. В остальных случаях начинает применяться поворот дерева вправо или влево.
4. Если «родитель» красный, «дядя» черный, а текущий узел является правым потомком, то добавленный узел является левым потомком красной вершины. В этом случае производится правое вращение и перекрашиваются две

вершины. Процесс перекраски окончится, так как вершина родитель будет чёрной.

5. Если «родитель» красный, «дядя» черный, а текущий узел является левым потомком, то добавленный узел является правым потомком красной вершины. В этом случае производится левое вращение, которое сводит это случай к случаю 2, когда добавляемый узел является потомком своего родителя. После вращения глубина, измеренная в чёрных узлах от корня к листьям, остаётся прежней.

Сложность алгоритма -  $O(\log N)$ .

### 2.1.3 Удаление

В процессе удаления вершины могут возникнуть 3 случая в зависимости от количества её потомков:

- Если у вершины нет потомков, то изменяем указатель на неё у родителя на NIL.
- Если у неё только один потомок, то делаем у родителя ссылку на него вместо этой вершины.
- Если имеются оба потомка, находим вершину со следующим значением ключа. У такой вершины нет левого потомка. Удаляем уже эту вершину, описанным в предыдущем пункте способом, скопировав её ключ в изначальную вершину.

Производится проверка балансировки, в результате которой могут быть произведены следующие действия:

1. При удалении красной вершины свойства дерева не нарушаются и балансировка не производится.
2. При удалении черной вершины, если «брат» этого потомка красный, то делаем вращение вокруг ребра между «отцом» и «братом», тогда «брат» становится родителем «отца». Красим его в чёрный цвет, а «отца» - в красный цвет.
3. При удалении черной вершины, если «брат» текущей вершины был чёрным, то требуется рассмотреть следующие случаи:
  - Если оба ребёнка у «брата» чёрные, то красим «брата» в красный цвет и рассматриваем далее «отца» вершины.

- Если у «брата» правый «ребёнок» черный, а левый красный, то перекрашиваем брата и его левого «сына» и делаем вращение.
- Если у брата правый «ребёнок» красный, то перекрашиваем «брата» в цвет отца, его «ребёнка» и «отца» - в чёрный, делаем вращение и выходим из алгоритма.

Продолжаем тот же алгоритм, пока текущая вершина чёрная и мы не дошли до корня дерева.

Сложность алгоритма -  $O(\log N)$ .

#### 2.1.4 Поиск

Алгоритм поиска начинается с корня. На каждой итерации происходит проверка, соответствует ли ключ рассматриваемого узла искомому. Возможны следующие случаи:

- Если да, то возвращается узел в качестве ответа.
- Если нет, то происходит сравнение текущего значения ключа и искомого. В зависимости от того больше или меньше переходим в правое или левое поддерево. Выполнение алгоритма продолжается до тех пор, пока не будет найден искомый узел или не дойдет до листа NIL.
- Если ответ не найден, возвращаем NULL.

Сложность алгоритма -  $O(\log N)$ .

## 2.2 Хеш-таблицы

### 2.2.1 Определение Хеш-таблицы

Хеш-таблица - это структура данных, которая хранит пары ключ-значение, где в качестве ключа выступает любой объект, для которого можно вычислить хеш-код. Возможны следующие операции над хеш-таблицами: добавление новой пары ключ-значение, поиск значения по ключу, удаление пары ключ-значение по ключу.

### 2.2.2 Хеш-функция

Хеш-функция  $f$  находит остаток от деления суммы кодов всех элементов строки на размер корзины. Результат хеш-функции является номером ячейки объекта.

$$f : T \rightarrow \{0, 1, \dots, m - 1\}$$

$$f(x) = (\sum_{i=0}^k n_i) \bmod m$$

$m$  - количество элементов,  $k$  - длина слова,  $n_i$  - код  $i$ -го символа в слове.

### 2.2.3 Функции хеш-таблицы

- Добавление элемента:

Вычисляется хеш-код ключа и ему присваивается индекс. Возвращаемое хеш-функцией, которая вычисляет хеш-код, значение не должно превосходить размер массива.

- Поиск элемента:

Находится  $id$  искомого элемента. Если элемент с найденным  $id$  - не единственный в цепочке, происходит поиск этого элемента в цепочке по найденному  $id$ .

- Удаление элемента:

Находится  $id$  искомого элемента. Если элемент с найденным  $id$  - единственный в цепочке, он удаляется из корзины, в ином случае он удаляется из цепочки по найденному  $id$ .

Сложность всех операций -  $O(1)$ .

### 2.2.4 Разрешение коллизий

У элементов может совпадать идентификатор. Разрешение коллизий в этой реализации происходит посредством метода цепочек.

Каждая ячейка корзины является указателем на цепочку пар ключ-значение, соответствующих одному и тому же хеш-значению ключа. В результате коллизии число элементов в цепочке увеличивается.



## 3 Особенности реализации

### 3.1 Красно-чёрное дерево

Дерево реализовано и хранится в памяти с помощью структуры `tnode`, которая содержит ключ, индекс, цвет, указатель на правого и левого потомка, указатель на родителя узла, и класса `RedBlackTree`, представляющего красно-черное дерево, который содержит одно поле - корень дерева.

---

```
struct tnode {
    string key;
    int ind = 0;
    treeColor color;
    tnode* left;
    tnode* right;
    tnode* parent;
    tnode(string k, treeColor c, tnode* p, tnode* l, tnode* r) :
        key(k), color(c), parent(p), left(l), right(r) { };
};

class RedBlackTree {
private:
    tnode* root;
public:
    RedBlackTree();
    ~RedBlackTree();
    void insert(string key);
    void loadFile(string file);
    void delElement(string key);
    tnode* find(string key);
    void PrintTree();
    void Clear();
private:
    void RotateLeft(tnode*& root, tnode* x);
    void RotateRight(tnode*& root, tnode* y);
    void DeleteTree(tnode*& node);
    void DeleteForDest(tnode*& node);
    tnode* find(tnode* node, string key) const;
    void PrintTree(tnode* node) const;
};
```

---

#### Основные методы:

**`void insert(string key)`** - метод добавления элемента в словарь. Принимает строку, возвращает словарь с добавленным словом.

Если дерево пустое, то добавленный элемент перекрашивается в чёрный цвет и становится корнем дерева.

Если дерево не пустое, то находится потенциальный родитель для добавленного элемента. Исходя из его значения, принятое значение помещается слева или справа.

С помощью перекраски узлов и функций поворота влево и вправо сохраняются свойства красно-чёрных деревьев.

Если элемент уже есть в словаре, то индекс этого элемента инкрементируется.

---

```
void RedBlackTree::insert(string key) {
    string tmpkey = key;
    for (int i = 0; i < tmpkey.size(); i++) {
        if (tmpkey[i] >= -64 && tmpkey[i] <= -33)
            tmpkey[i] = tmpkey[i] + 32;
        if (tmpkey[i] == -88)
            tmpkey[i] = 'ë';
    }
    tnode* k = this->find(tmpkey);
    if (k) {
        k->ind++;
        return;
    }
    tnode* z = new tnode(tmpkey, Red, NULL, NULL, NULL);
    tnode* x = root;
    tnode* y = NULL;
    while (x != NULL) {
        y = x;
        if (z->key > x->key)
            x = x->right;
        else
            x = x->left;
    }
    z->parent = y;
    if (y != NULL) {
        if (z->key > y->key)
            y->right = z;
        else
            y->left = z;
    }
    else root = z;
    z->color = Red;
    tnode* parent;
    parent = z->parent;
    while (z != root && parent->color == Red) {
        tnode* gparent = parent->parent;
        if (gparent->left == parent) {
            tnode* uncle = gparent->right;
            if (uncle != NULL && uncle->color == Red) {
                parent->color = Black;
                uncle->color = Black;
                gparent->color = Red;
                z = gparent;
                parent = z->parent;
            }
        }
        else {
            if (parent->right == z) {
                RotateLeft(root, parent);
                swap(z, parent);
            }
        }
    }
}
```

```

    }
    RotateRight(root, gparent);
    gparent->color = Red;
    parent->color = Black;
    break;
}
}
else {
    tnode* uncle = gparent->left;
    if (uncle != NULL && uncle->color == Red) {
        gparent->color = Red;
        parent->color = Black;
        uncle->color = Black;

        z = gparent;
        parent = z->parent;
    }
    else {
        if (parent->left == z) {
            RotateRight(root, parent);
            swap(parent, z);
        }
        RotateLeft(root, gparent);
        parent->color = Black;
        gparent->color = Red;
        break;
    }
}
}
root->color = Black;
};

```

---

**void loadFile(string file)** - метод заполнения словаря из файла. Принимает имя файла, возвращает словарь с загруженными из файла словами.

---

```

void RedBlackTree::loadFile(string file) {
    vector<char> text;
    char symbol;
    ifstream myfile(file);
    if (myfile.is_open()) {
        while (myfile.get(symbol)) {
            if (symbol != '\n') {
                text.push_back(symbol);
            }
            else {
                text.push_back(' ');
            }
        }
        myfile.close();
    }
    string res;
    for (int i = 0; i < text.size(); i++) {
        if ((text[i] <=-1 && text[i] >= -64) || text[i] ==-72 || text[i]==-88) {

```

```

        if (text[i] >= -64 && text[i] <= -33)
            text[i] = text[i] + 32;
        if (text[i] == -88)
            text[i] = 'ë';
        res += text[i];
    }
    else {
        if (res.size()) {
            this->insert(res);
            res.clear();
        }
    }
    if (i == text.size() - 1)
        this->insert(res);
}
}

```

---

**tnode\* find(string key)** - метод поиска элемента по ключу в словаре. Принимает в качестве параметра строку, возвращает указатель на найденный узел. В ней вызывается следующая перегрузка метода:

**tnode\* find(tnode\* node, string key)**, которая принимает в качестве параметра указатель на просматриваемый узел и строку. Является рекурсивным.

---

```

tnode* RedBlackTree::find(string key) {
    return find(root, key);
}

tnode* RedBlackTree::find(tnode* node, string key) {
    if (node == NULL || node->key == key)
        return node;
    else
        if (key > node->key)
            return find(node->right, key);
        else
            return find(node->left, key);
}

```

---

**void delElement(string key)** - метод удаления элемента из словаря. Принимает строку, возвращает словарь без удаляемого элемента.

С помощью перекраски узлов и функций поворота влево и вправо сохраняются свойства красно-чёрных деревьев.

---

```

tnode* el = find(root, key);
if (el == root && el->left == NULL && el->right == NULL) {
    root = NULL;
    return;
}
if (el != NULL) {
    tnode* child, * parent;
    treeColor color;
    if (el->left != NULL && el->right != NULL) {

```

```

tnode* replace = el;
replace = el->right;
while (replace->left != NULL) {
    replace = replace->left;
}
if (el->parent != NULL) {
    if (el->parent->left == el)
        el->parent->left = replace;
    else
        el->parent->right = replace;
}
else root = replace;
child = replace->right;
parent = replace->parent;
color = replace->color;
if (parent == el) parent = replace;
else {
    if (child != NULL) child->parent = parent;
    parent->left = child;
    replace->right = el->right;
    el->right->parent = replace;
}
replace->parent = el->parent;
replace->color = el->color;
replace->left = el->left;
el->left->parent = replace;
if (color == Black) {
    tnode* on;
    while ((!child) || child->color == Black && child != RedBlackTree::root) {
        if (parent->left == child) {
            on = parent->right;
            if (on->color == Red) {
                on->color = Black;
                parent->color = Red;
                RotateLeft(root, parent);
                on = parent->right;
            }
            else {
                if (!(on->right) || on->right->color == Black) {
                    on->left->color = Black;
                    on->color = Red;
                    RotateRight(root, on);
                    on = parent->right;
                }
                on->color = parent->color;
                parent->color = Black;
                on->right->color = Black;
                RotateLeft(root, parent);
                child = root;
                break;
            }
        }
    }
}
else {

```

```

        on = parent->left;
        if (on->color == Red) {
            on->color = Black;
            parent->color = Red;
            RotateRight(root, parent);
            on = parent->left;
        }
        if ((!on->left || on->left->color == Black) && (!on->right ||
            on->right->color == Black)) {
            on->color = Red;
            child = parent;
            parent = child->parent;
        }
        else {
            if (!(on->left) || on->left->color == Black) {
                on->right->color = Black;
                on->color = Red;
                RotateLeft(root, on);
                on = parent->left;
            }
            on->color = parent->color;
            parent->color = Black;
            on->left->color = Black;
            RotateRight(root, parent);
            child = root;
            break;
        }
    }
}

if (child) child->color = Black;
}
delete el;
return;
}
if (el->left != NULL) child = el->left;
else child = el->right;
parent = el->parent;
color = el->color;
if (child) child->parent = parent;
if (parent) {
    if (el == parent->left) parent->left = child;
    else parent->right = child;
}
else RedBlackTree::root = child;
if (color == Black) {
    tnode* on;
    while ((!child) || child->color == Black && child != RedBlackTree::root) {
        if (parent->left == child) {
            on = parent->right;
            if (on->color == Red){
                on->color = Black;
                parent->color = Red;
                RotateLeft(root, parent);
            }
        }
    }
}

```

```

        on = parent->right;
    }
    else {
        if (!(on->right) || on->right->color == Black){
            on->left->color = Black;
            on->color = Red;
            RotateRight(root, on);
            on = parent->right;
        }
        on->color = parent->color;
        parent->color = Black;
        on->right->color = Black;
        RotateLeft(root, parent);
        child = root;
        break;
    }
}
else {
    on = parent->left;
    if (on->color == Red){
        on->color = Black;
        parent->color = Red;
        RotateRight(root, parent);
        on = parent->left;
    }
    if ((!on->left || on->left->color == Black) && (!on->right ||
        on->right->color == Black)){
        on->color = Red;
        child = parent;
        parent = child->parent;
    }
    else{
        if (!(on->left) || on->left->color == Black){
            on->right->color = Black;
            on->color = Red;
            RotateLeft(root, on);
            on = parent->left;
        }
        on->color = parent->color;
        parent->color = Black;
        on->left->color = Black;
        RotateRight(root, parent);
        child = root;
        break;
    }
}
}
if (child) child->color = Black;
}
delete el;
}

```

---

**void Clear()** - метод очистки словаря. Принимает словарь дерева, возвра-

щает его пустым. В методе вызывается метод **void DeleteTree(tnode\* node)**, которому в качестве параметра передается корень дерева и который ничего не возвращает. Это рекурсивный метод, удаляющий принимаемый узел.

---

```
void RedBlackTree::Clear() {
    DeleteTree(root);
}

void RedBlackTree::DeleteTree(tnode*& node) {
    if (root == NULL) cout << "Словарь пуст." << endl;
    if (node == NULL) return;
    DeleteForDest(node->left);
    DeleteForDest(node->right);
    delete node;
    node = nullptr;
    cout << "Очистка успешна." << endl;
}
```

---

## 3.2 Хеш-таблица

Хеш-таблица реализована и хранится в памяти с помощью структуры `el`, которая содержит ключ, индекс, цвет, указатель на следующий элемент с тем же `id`, и класса `HashTable`, представляющего хеш-таблицу, который содержит два поля: `data` - корзина, `size` - размер корзины.

---

```
struct el {
    string data;
    int ind = 1;
    el* ref;
    el(string d = "", el* r = NULL) {
        data = d;
        ref = r;
    }
};

class HashTable {
    int size;
public:
    el* data;
    ~HashTable();
    HashTable();
    int HF(string e);
    void insert(string k);
    void loadFile(string file);
    el* find(string k);
    void delElement(string k);
    void Clear();
    void PrintTable();
};
```

---



**int HF(string e)** - это метод, реализующий хеш-функцию для определения id элемента. Принимает строку, возвращает целочисленное значение в диапазоне от 0 до 99. Функция складывает коды ASCII каждого символа строки и делит на размер корзины. Результат - остаток от деления.

---

```
int HashTable::HF(string e) {
    int sum = 0;
    for (int i = 0; i < e.size(); i++) sum += e[i];
    if (sum < 0) sum *= -1;
    return (sum % size);
}
```

---

**void insert(string k)** - метод добавления элемента в словарь. Принимает строку, возвращает словарь с добавленным словом.

Если строка совпадает со строкой существующего элемента, индекс этого элемента инкрементируется.

Если элементов с найденным id не существует в словаре, в корзину добавляется элемент с ключом, переданным в качестве параметра.

Если в словаре существуют элементы с подобным id, в конец цепочки добавляется элемент с ключом, переданным в качестве параметра.

---

```
void HashTable::insert(string k) {
    if (this->find(k) == NULL) {
        int id = HF(k);
        el d;
        d.data = k;
        d.ref = NULL;
        if (data[id].data == "") data[id] = d;
        else {
            el* cur = &data[id];
            while (cur->ref) cur = cur->ref;
            cur->ref = new el;
            cur->ref->data = k;
            cur->ref->ref = NULL;
        }
    }
    else {
        if (this->find(k)->data == k) {
            this->find(k)->ind++;
        }
        else {
            this->find(k)->ref->ind++;
        }
    }
}
```

---

**void loadFile(string file)** - метод заполнения словаря из файла. Принимает имя файла, возвращает словарь с загруженными из файла словами.

---

```
void HashTable::loadFile(string file) {
```

```

vector<char> text;
char symbol;
ifstream myfile(file);
if (myfile.is_open()) {
    while (myfile.get(symbol)) {
        if (symbol != '\n') {
            text.push_back(symbol);
        }
        else {
            text.push_back(' ');
        }
    }
    myfile.close();
}
string res;
for (int i = 0; i < text.size(); i++) {
    if ((text[i] <= -1 && text[i] >= -32) || (text[i] <= -33 && text[i] >= -64) ||
        text[i] == -72 || text[i] == -88
        || text[i] <= 175 && text[i] >= 128 || text[i] <= 240 && text[i] >= 224) res +=
        text[i];
    else {
        if (res.size()) {
            this->insert(res);
            res.clear();
        }
    }
    if ((i == text.size() - 1) && res.size()) this->insert(res);
}
}

```

---

**el\* find(string k)** - метод поиска элемента в словаре. Принимает строку, возвращает указатель на элемент, если он один в цепи, иначе возвращает указатель на предшествующий в цепи элемент.

---

```

el* HashTable::find(string k) {
    int id = HF(k);
    if (data[id].data == k) {
        return &data[id];
    }
    else {
        el* cur = &data[id];
        while (cur->ref != nullptr) {
            if (cur->ref->data == k) {
                return cur;
            }
            cur = cur->ref;
        }
    }
    return NULL;
}

```

---

**void delElement(string k)** - метод удаления элемента из словаря. Принимает

строку, возвращает словарь без удаляемого элемента.

Если элементов с таким ключом несколько, их количество уменьшается на 1.

---

```
void HashTable::delElement(string k) {
    el* rmv = find(k);
    if (rmv != NULL) {
        if (rmv->ind > 1 && rmv->data == k) {
            rmv->ind--;
            return;
        }
        if (rmv->ref != NULL) {
            if (rmv->data != k) {
                if (rmv->ref->ind > 1) {
                    rmv->ref->ind--;
                    return;
                }
                if (rmv->ref->data == k) {
                    if (!rmv->ref->ref) {
                        delete rmv->ref;
                        rmv->ref = NULL;
                    }
                    else {
                        el* del = rmv->ref;
                        rmv->ref->ind = rmv->ref->ref->ind;
                        rmv->ref->data = rmv->ref->ref->data;
                        rmv->ref = rmv->ref->ref;
                        delete del;
                    }
                }
            }
        }
        else {
            el* del = rmv->ref;
            rmv->data = rmv->ref->data;
            rmv->ind = rmv->ref->ind;
            rmv->ref = rmv->ref->ref;
            delete del;
        }
    }
    else {
        rmv->data = "";
        rmv->ref = NULL;
    }
}
```

---

**void Clear()** - метод очистки словаря посредством множественного выполнения метода delElement(). Принимает словарь таблицы, возвращает его пустым.

---

```
void HashTable::Clear() {
    bool flag = true;
    for (int i = 0; i < size; i++) {
        if (data[i].data != "") {
            if (data[i].ind > 1) data[i].ind = 1;
        }
    }
}
```

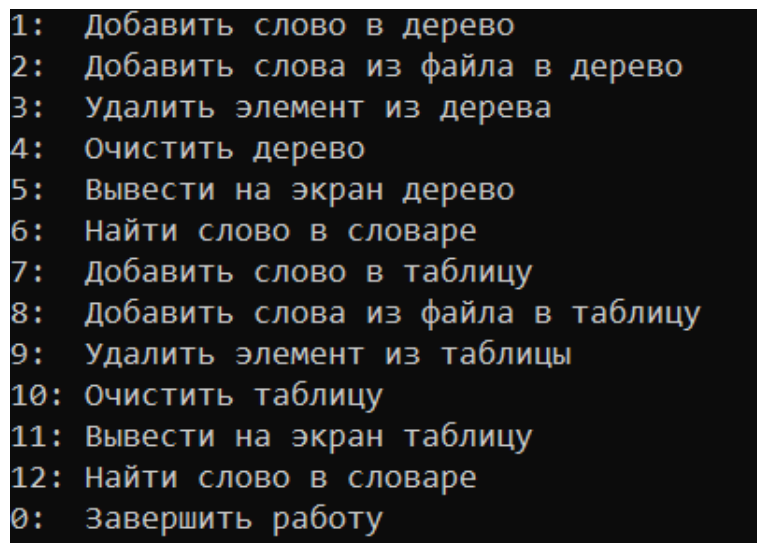
---

```
        while (data[i].ref != NULL) delElement(data[i].ref->data);
        delElement(data[i].data);
        flag = false;
    }
}
if (flag) cout << "Словарь пуст." << endl;
else cout << "Очистка успешна." << endl;
}
```

---

## 4 Результаты работы программы

На рис. 1 представлен интерфейс программы, который выводится в консоль в начале работы программы и после каждой выполненной операции.



```
1:  Добавить слово в дерево
2:  Добавить слова из файла в дерево
3:  Удалить элемент из дерева
4:  Очистить дерево
5:  Вывести на экран дерево
6:  Найти слово в словаре
7:  Добавить слово в таблицу
8:  Добавить слова из файла в таблицу
9:  Удалить элемент из таблицы
10: Очистить таблицу
11: Вывести на экран таблицу
12: Найти слово в словаре
0:  Завершить работу
```

Рис. 1: Меню операций

На рис. 2-11 представлены примеры работы программы, а именно:

- Добавление слова в дерево, как показано на рис. 2.
- Добавление слов в словарь дерева из файла, как показано на рис. 3.
- Попытка удалить отсутствующее в словаре дерева слово, как показано на рис. 4.
- Успешное удаление слова из словаря дерева, как показано на рис. 5.
- Очистка дерева, как показано на рис. 6.
- Поиск слова в словаре дерева, как показано на рис. 7.
- Добавление слова в хеш-таблицу, как показано на рис. 8.
- Добавление слов в словарь из файла, как показано на рис. 9.
- Попытка удалить отсутствующее в словаре таблицы слово, как показано на рис. 10.
- Успешное удаление слова из словаря таблицы, как показано на рис. 11.
- Очистка словаря на основе хеш-таблицы, как показано на рис. 12.
- Поиск слова в словаре таблицы, как показано на рис. 13.

```

Выберете пункт меню:
1

Введите слово:
Вася

1:  Добавить слово в дерево
2:  Добавить слова из файла в дерево
3:  Удалить элемент из дерева
4:  Очистить дерево
5:  Вывести на экран дерево
6:  Найти слово в словаре
7:  Добавить слово в таблицу
8:  Добавить слова из файла в таблицу
9:  Удалить элемент из таблицы
10: Очистить таблицу
11: Вывести на экран таблицу
12: Найти слово в словаре
0:  Завершить работу

Выберете пункт меню:
5

Корень 'вася' - Черный
Левый потомок 'вася' - NIL
Правый потомок 'вася' - NIL

```

Рис. 2: Добавление слова в дерево

```

Выберете пункт меню:
2

1:  Добавить слово в дерево
2:  Добавить слова из файла в дерево
3:  Удалить элемент из дерева
4:  Очистить дерево
5:  Вывести на экран дерево
6:  Найти слово в словаре
7:  Добавить слово в таблицу
8:  Добавить слова из файла в таблицу
9:  Удалить элемент из таблицы
10: Очистить таблицу
11: Вывести на экран таблицу
12: Найти слово в словаре
0:  Завершить работу

Выберете пункт меню:
5

Корень 'зовут' - Черный
Левый потомок 'зовут' : вася - Черный
Левый потомок 'вася' : василий - Красный
Левый потомок 'василий' - NIL
Правый потомок 'василий' - NIL
Правый потомок 'вася' - NIL
Правый потомок 'зовут' : меня - Черный
Левый потомок 'меня' - NIL
Правый потомок 'меня' - NIL

```

Рис. 3: Добавление слов в словарь дерева из файла

```
Выберете пункт меню:
3

Введите слово:
удалить

Такого элемента нет.
1: Добавить слово в дерево
2: Добавить слова из файла в дерево
3: Удалить элемент из дерева
4: Очистить дерево
5: Вывести на экран дерево
6: Найти слово в словаре
7: Добавить слово в таблицу
8: Добавить слова из файла в таблицу
9: Удалить элемент из таблицы
10: Очистить таблицу
11: Вывести на экран таблицу
12: Найти слово в словаре
0: Завершить работу

Выберете пункт меню:
5

Корень 'зовут' - Черный
Левый потомок 'зовут' : вася - Черный
Левый потомок 'vasя' : василий - Красный
Левый потомок 'vasилий' - NIL
Правый потомок 'vasилий' - NIL
Правый потомок 'vasя' - NIL
Правый потомок 'зовут' : меня - Черный
Левый потомок 'меня' - NIL
Правый потомок 'меня' - NIL
```

Рис. 4: Попытка удалить отсутствующее в словаре дерева слово

```
Выберете пункт меню:
9

Введите слово:
Меня

Элемент удален.
1: Добавить слово в дерево
2: Добавить слова из файла в дерево
3: Удалить элемент из дерева
4: Очистить дерево
5: Вывести на экран дерево
6: Найти слово в словаре
7: Добавить слово в таблицу
8: Добавить слова из файла в таблицу
9: Удалить элемент из таблицы
10: Очистить таблицу
11: Вывести на экран таблицу
12: Найти слово в словаре
0: Завершить работу

Выберете пункт меню:
11

( 0): зовут 2
( 1): Василий 1
```

Рис. 5: Успешное удаление слова из словаря дерева

```
Выберете пункт меню:
4

Очистка успешна.
1: Добавить слово в дерево
2: Добавить слова из файла в дерево
3: Удалить элемент из дерева
4: Очистить дерево
5: Вывести на экран дерево
6: Найти слово в словаре
7: Добавить слово в таблицу
8: Добавить слова из файла в таблицу
9: Удалить элемент из таблицы
10: Очистить таблицу
11: Вывести на экран таблицу
12: Найти слово в словаре
0: Завершить работу

Выберете пункт меню:
5

Дерево пусто.
```

Рис. 6: Очистка дерева



```

Выберете пункт меню:
6

Введите слово:
зовут

Слово найдено.
1: Добавить слово в дерево
2: Добавить слова из файла в дерево
3: Удалить элемент из дерева
4: Очистить дерево
5: Вывести на экран дерево
6: Найти слово в словаре
7: Добавить слово в таблицу
8: Добавить слова из файла в таблицу
9: Удалить элемент из таблицы
10: Очистить таблицу
11: Вывести на экран таблицу
12: Найти слово в словаре
0: Завершить работу

Выберете пункт меню:
5

Корень 'зовут' - Черный
Левый потомок 'зовут' : василий - Красный
Левый потомок 'василий' - NIL
Правый потомок 'василий' - NIL
Правый потомок 'зовут' : меня - Красный
Левый потомок 'меня' - NIL
Правый потомок 'меня' - NIL

```

Рис. 7: Поиск слова в дереве

```

Выберете пункт меню:
7

Введите слово:
СлооВо

1: Добавить слово в дерево
2: Добавить слова из файла в дерево
3: Удалить элемент из дерева
4: Очистить дерево
5: Вывести на экран дерево
6: Найти слово в словаре
7: Добавить слово в таблицу
8: Добавить слова из файла в таблицу
9: Удалить элемент из таблицы
10: Очистить таблицу
11: Вывести на экран таблицу
12: Найти слово в словаре
0: Завершить работу

Выберете пункт меню:
11

(84): СлооВо 1

```

Рис. 8: Добавление слова в хеш-таблицу

```

Выберете пункт меню:
8

1:  Добавить слово в дерево
2:  Добавить слова из файла в дерево
3:  Удалить элемент из дерева
4:  Очистить дерево
5:  Вывести на экран дерево
6:  Найти слово в словаре
7:  Добавить слово в таблицу
8:  Добавить слова из файла в таблицу
9:  Удалить элемент из таблицы
10: Очистить таблицу
11: Вывести на экран таблицу
12: Найти слово в словаре
0:  Завершить работу

Выберете пункт меню:
11

( 0): зовут 2
( 1): Василий 1
(99): Меня 1

```

Рис. 9: Добавление слов в словарь из файла

```

Выберете пункт меню:
9

Введите слово:
Нергигант

Такого элемента нет.
1:  Добавить слово в дерево
2:  Добавить слова из файла в дерево
3:  Удалить элемент из дерева
4:  Очистить дерево
5:  Вывести на экран дерево
6:  Найти слово в словаре
7:  Добавить слово в таблицу
8:  Добавить слова из файла в таблицу
9:  Удалить элемент из таблицы
10: Очистить таблицу
11: Вывести на экран таблицу
12: Найти слово в словаре
0:  Завершить работу

Выберете пункт меню:
11

( 0): зовут 2
( 1): Василий 1
(99): Меня 1

```

Рис. 10: Попытка удалить отсутствующее в словаре таблицы слово

```

Выберете пункт меню:
9

Введите слово:
Василий

Элемент удален.
1: Добавить слово в дерево
2: Добавить слова из файла в дерево
3: Удалить элемент из дерева
4: Очистить дерево
5: Вывести на экран дерево
6: Найти слово в словаре
7: Добавить слово в таблицу
8: Добавить слова из файла в таблицу
9: Удалить элемент из таблицы
10: Очистить таблицу
11: Вывести на экран таблицу
12: Найти слово в словаре
0: Завершить работу

Выберете пункт меню:
11

( 0): зовут 2
(99): Меня 1

```

Рис. 11: Успешное удаление слова из словаря таблицы

```

Выберете пункт меню:
10

Очистка успешна.
1: Добавить слово в дерево
2: Добавить слова из файла в дерево
3: Удалить элемент из дерева
4: Очистить дерево
5: Вывести на экран дерево
6: Найти слово в словаре
7: Добавить слово в таблицу
8: Добавить слова из файла в таблицу
9: Удалить элемент из таблицы
10: Очистить таблицу
11: Вывести на экран таблицу
12: Найти слово в словаре
0: Завершить работу

Выберете пункт меню:
11

Словарь пуст.

```

Рис. 12: Очистка словаря на основе хеш-таблицы

```
Выберете пункт меню:
12

Введите слово:
Василий

Слово найдено.
1:  Добавить слово в дерево
2:  Добавить слова из файла в дерево
3:  Удалить элемент из дерева
4:  Очистить дерево
5:  Вывести на экран дерево
6:  Найти слово в словаре
7:  Добавить слово в таблицу
8:  Добавить слова из файла в таблицу
9:  Удалить элемент из таблицы
10: Очистить таблицу
11: Вывести на экран таблицу
12: Найти слово в словаре
0:  Завершить работу

Выберете пункт меню:
11

( 0): зовут 2
( 1): Василий 1
(99): Меня 1
```

Рис. 13: Очистка словаря на основе хеш-таблицы

## Заключение

В результате выполнения работы были реализованы словари на основе красно-черного дерева и хеш-таблицы, операции добавления, поиска и удаления элемента для каждого, функции загрузки слов из файла и очистки словарей.

### Достоинства:

- Хранение повторяющихся слов в словаре красно-черного дерева в численном виде.
- Быстрый поиск в словаре на основе красно-черного дерева.
- Высокая скорость выполнения операций со словарем на основе хеш-таблицы ( $O(1)$ ).
- Подсчет количества одинаковых слов в словаре на основе хеш-таблицы.

### Недостатки:

- Непрактичное текстовое отображение красно-черного дерева.
- Не выполняется требование о невозможности нахождения двух разных сообщений с одинаковым хеш-значением.

### Масштабируемость:

- Добавление возможности пересоздания хеш-таблицы с заданным пользователем объемом.

## Источники

1. Новиков Ф.А. Дискретная математика для программистов. 3-е издание. СПб.:Питер,2009. 384 с.
2. Востров А.В. Лекция «Теория графов. Информационные деревья» СПб., 2022. 54 с.
3. Красно-черное дерево [neerc.ifmo.ru/wiki/index.php?title=Красно-черное\\_дерево](https://neerc.ifmo.ru/wiki/index.php?title=Красно-черное_дерево)