

МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

**«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
ПЕТРА ВЕЛИКОГО»**

Институт прикладной математики и механики

Высшая школа искусственного интеллекта

Секция: «Телематика» (при ЦНИИ РТК)

Направление 02.03.01 Математика и компьютерные науки

Теория графов

Отчёт

по лабораторной работе №6

«Реализация словаря на основе красно-черного дерева.

Реализация хэш-таблицы.»

Студент: _____ группа 3630201/90002 Коренова А.Д.

Преподаватель: _____ Востров А.В.

« ____ » _____ 20__

Санкт-Петербург — 2021

Содержание

Введение	3
1 Математическое описание	4
1.1 Красно-черное дерево	4
1.1.1 Определение красно-черного дерева	4
1.1.2 Вставка элемента	4
1.1.3 Удаление элемента	5
1.1.4 Поиск в красно-черном дереве	7
1.2 Хэш-таблица	8
1.2.1 Понятие хэш-таблицы	8
1.2.2 Выбор хэш-функции	8
2 Особенности реализации	10
2.1 Структура данных	10
2.2 Красно-черное дерево	10
2.2.1 Добавление элемента в дерево	10
2.2.2 Балансировка красно-черного дерева при добавлении в него элемента	11
2.2.3 Удаление элемента	12
2.2.4 Балансировка красно-черного дерева при удалении элемента	14
2.2.5 Поиск элемента	16
2.2.6 Полная очистка словаря	16
2.2.7 Добавление в словарь текста из файла	17
2.3 Хэш-таблица	18
2.3.1 Построение хэш-таблицы	18
2.3.2 Удаление элемента	18
2.3.3 Поиск элемента	19
3 Результаты работы программы	21
3.1 Красно-черное дерево	21
3.2 Хэш-таблица	25
Заключение	30
Список литературы	30

Введение

Красно-черное дерево — двоичное дерево поиска, в котором баланс осуществляется на основе «цвета» узла дерева, который может принимать только два значения: красный и черный. При этом все листья дерева являются фиктивными и не содержат данных, но относятся к дереву и являются черными.

Целью данной лабораторной работы является:

- построить словарь на основе красно-черного дерева;
- реализовать функции добавления, удаления, поиска слова;
- реализовать функцию полной очистки словаря и дополнения словаря из текстового файла.

1 Математическое описание

1.1 Красно-черное дерево

1.1.1 Определение красно-черного дерева

Красно-черное дерево — бинарное дерево, баланс которого достигается за счет поддержания раскраски вершин в два цвета, подчиняющейся следующим *правилам*:

1. корень окрашен в **черный** цвет;
2. каждый узел покрашен либо в **черный**, либо в **красный** цвет;
3. листьями объявляются **NIL-узлы** (то есть «виртуальные» узлы, наследники узлов, которые обычно называют листьями; на них указывают NULL-указатели). Листья покрашены в черный цвет;
4. если узел красный, то **оба его потомка** черные;
5. на всех ветвях дерева, ведущих от его корня к листьям, число черных узлов **одинаково**[1].

В реализации для этого вида дерева необходимо в каждой вершине хранить дополнительно 1 бит информации для цвета[2].

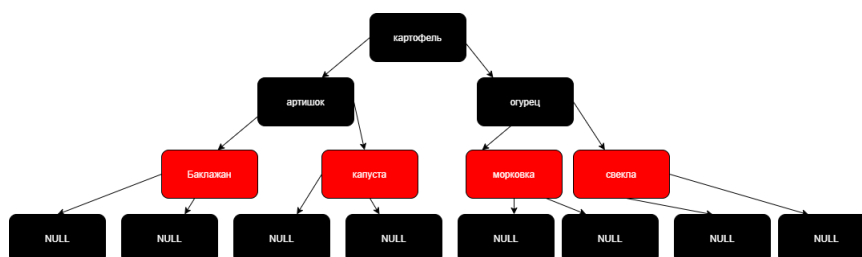


Рис. 1: Пример красно-черного дерева

Существуют эффективные процедуры добавления и удаления узлов, которые не нарушают свойств красно-черных деревьев.

1.1.2 Вставка элемента

Каждый элемент *вставляется вместо листа*, поэтому для выбора места вставки идем от корня до тех пор, пока указатель на следующего сына не станет NIL. Вставляем вместо него новый элемент с NIL-потомками и красным цветом. Теперь проверяем балансировку. Если отец нового элемента черный, то никакое из свойств дерева не нарушено. Если же он красный, то нарушается свойство, что *у красного узла родительский узел — черный*, для исправления этой ситуации достаточно рассмотреть два случая[4].

- **«Дядя» этого узла тоже красный.** Тогда, перекрашиваем «отца» и «дядю» в черный цвет, а «деда» — в красный. В таком случае черная высота в этом поддереве одинакова для всех листьев и у всех красных вершин «отцы» черные. Проверяем,

не нарушена ли балансировка. Если в результате этих перекрашиваний мы дойдем до корня, то в нем в любом случае ставим черный цвет, чтобы дерево удовлетворяло свойству 1.

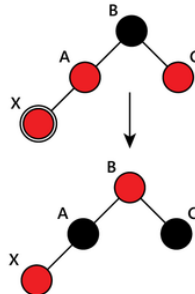


Рис. 2: Вставка элемента. Случай красного «дяди»

- **«Дядя» черный.** Если выполнить только перекрашивание, то может нарушиться постоянство черной высоты дерева по всем ветвям. Поэтому выполняем поворот. Если добавляемый узел был правым потомком, то необходимо сначала выполнить левое вращение, которое сделает его левым потомком. Таким образом, постоянство черной высоты сохраняется.

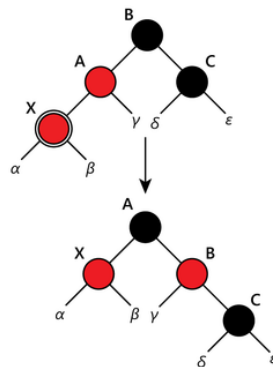


Рис. 3: Вставка элемента. Случай черного «дяди»

1.1.3 Удаление элемента

При **удалении** вершины могут возникнуть три случая в зависимости от количества ее детей:

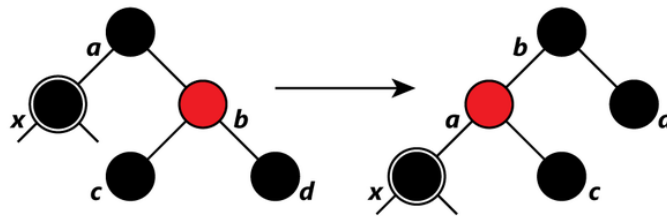
- если у вершины нет детей, то изменяем указатель на нее у родителя на NIL;
- если у нее только один ребенок, то делаем у родителя ссылку на него вместо этой вершины;
- если же имеются оба ребенка, то находим вершину со следующим значением ключа. У такой вершины нет левого ребенка (так как такая вершина находится в правом поддереве исходной вершины и она самая левая в нем, иначе мы бы взяли ее левого

ребенка. Иными словами сначала мы переходим в правое поддерево, а после спускаемся вниз в левое до тех пор, пока у вершины есть левый ребенок). Удаляем уже эту вершину описанным во втором пункте способом, скопировав её ключ в изначальную вершину[4].

1.1.3.1 Проверка балансировки дерева

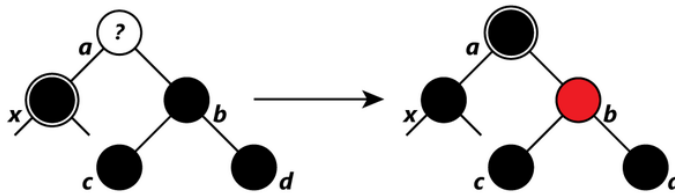
Далее необходимо проверить **балансировку** дерева. Так как при удалении красной вершины свойства дерева не нарушаются, то *восстановление балансировки* потребуется только при удалении черной. Рассмотрим ребенка удаленной вершины.

- Если брат этого ребенка **красный**, то делаем вращение вокруг ребра между отцом и братом, тогда брат становится родителем отца. Красим его в **черный**, а отца — в **красный** цвет, сохраняя таким образом черную высоту дерева. Хотя все пути по-прежнему содержат одинаковое количество чёрных узлов, сейчас x имеет чёрного брата и красного отца. Таким образом, мы можем перейти к следующему шагу.

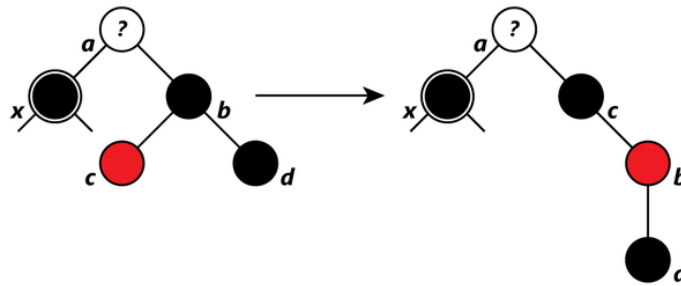


- Если брат текущей вершины был **черным**, то получаем три случая.

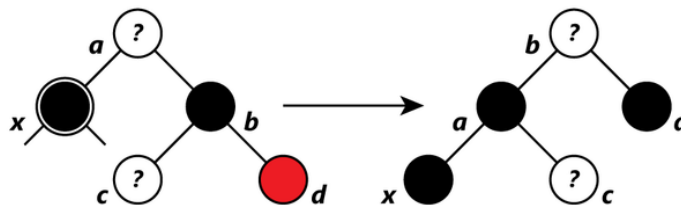
1. **Оба ребенка у брата черные.** Красим брата в красный цвет и рассматриваем далее отца вершины. Делаем его черным, это не повлияет на количество черных узлов на путях, проходящих через b , но добавит один к числу черных узлов на путях, проходящих через x , восстанавливая тем самым влияние удаленного черного узла. Таким образом, после удаления вершины черная глубина от отца этой вершины до всех листьев в этом поддереве будет одинаковой.



2. **Если у брата правый ребёнок чёрный, а левый красный,** то перекрашиваем брата и его левого сына и делаем вращение. Все пути по-прежнему содержат одинаковое количество чёрных узлов, но теперь у x есть чёрный брат с красным правым потомком, и мы переходим к следующему случаю. Ни x , ни его отец не влияют на эту трансформацию.



3. Если у брата правый ребёнок красный, то перекрашиваем брата в цвет отца, его ребёнка и отца — в чёрный, делаем вращение. Поддерево по-прежнему имеет тот же цвет корня. Но у x теперь появился дополнительный чёрный предок: либо a стал чёрным, или он и был чёрным и b был добавлен в качестве чёрного дедушки. Таким образом, проходящие через x пути проходят через один дополнительный чёрный узел. Выходим из алгоритма.



1.1.4 Поиск в красно-черном дереве

Поиск в красно-черном дереве происходит как и в бинарном дереве[4].

Алгоритм стартует от вершины, на каждой итерации происходит проверка, соответствует ли ключ нашего узла ключу, который мы ищем.

- 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 соответственно в правое или левое поддерево. Алгоритм повторяет вышеперечисленное действие до тех пор, пока не дойдет до листа NIL.
- Если ответ **не найден**, то возвращаем NULL.

Сложность алгоритма: $O(\log N)$.

1.2 Хэш-таблица

1.2.1 Понятие хэш-таблицы

Хэш-таблица — это специальная структура данных для хранения пар ключей и их значений. По сути это ассоциативный массив, в котором ключ представлен в виде *хэш-функции*.

Как правило, мощность множества ключей существенно больше размера пространства адресов и количества одновременно хранимых записей. Поэтому при использовании хэширования возможны **коллизии** — ситуации, когда хэш-функция сопоставляет один и тот же адрес двум актуальным записям с различными ключами[3].

Одним из способов борьбы с коллизиями является **метод цепочек** или **метод открытого хэширования**. При использовании данного метода элементы с одинаковым хэшем попадают в одну ячейку в виде связного списка.

Второй распространенный метод — **открытая индексация**. Это значит, что пары ключ-значение хранятся непосредственно в хэш-таблице. А алгоритм вставки проверяет ячейки в некотором порядке, пока не будет найдена пустая ячейка.

Пожалуй, главное свойство хэш-таблиц — все три операции: вставка, поиск и удаление — в среднем выполняются за время $O(1)$, среднее время поиска по ней также равно $O(1)$ и $O(n)$ в худшем случае.

1.2.2 Выбор хэш-функции

В качестве хэш-функции была выбрана функция, использующая первую букву слова, добавляемого в словарь.

Таким образом, слова организуются в словаре в алфавитном порядке, что позволяет обеспечить эффективный поиск.

Например, пусть на вход хэш-функции подается слово `string str = <<apбyz>>`, тогда хэш-функция будет выглядеть следующим образом $h(str) = 'a'$.


```
C:\Users\acer\Desktop\Лабы\Graph Theory\Graph Theory-2021082
Выберите файл:
[1] - products.txt;
[2] - murakami.txt;
[3] - gaiman.txt;

>> 1

Текст файла:
Картофель, молоко, лимон, огурец, лапша, курица, мо
, жимолость, нектарины.

***СЛОВАРЬ***
К: Картофель
в: вино виноград
ж: жимолость
к: курица картофель
л: лимон лапша лайм
м: молоко моцарелла морковь мандарины
н: нектарины
о: огурец
р: руккола

*****
```

Рис. 4: Пример хэш-таблицы

2 Особенности реализации

2.1 Структура данных

В данном разделе будут перечислены все структуры данных, в которых хранятся: узел дерева, само красно-черное дерево, а также хэш-таблица.

- `enum COLOR` — перечисление, в котором хранятся цвета узлов красно-черного дерева.
- `class Node` — класс, который содержит информацию об одном узле: значение элемента, цвет узла и указатели на родителя и потомков. Методы класса `Node` являются вспомогательными, используются для сохранения свойств красно-черных деревьев.
- `class RBTree` — класс, который реализует структуру красно-черного дерева. В качестве поля содержит элемент типа `Node`, что позволяет оперировать узлами дерева, не теряя свойств дерева.
- `map<char, vector<string>> HashTable` — ассоциативный контейнер, в котором в качестве ключа находится первая буква слова, а в качестве значения вектор слов, которые начинаются на ту букву, что хранится в ключе.
- `vector<char> number` — вспомогательный вектор, в котором индексу элементов соответствует буква. Нужен для лучшей ориентации по хэш-таблице.

2.2 Красно-черное дерево

2.2.1 Добавление элемента в дерево

Функция: `void RBTree::insert(string str).`

Вход функции: на вход функции подается слово, которое необходимо добавить в дерево.

Выход функции: дерево с добавленным словом.

Пример операции: см. рисунок 6.

Описание работы функции: при добавлении элемента первым делом проверяем не пустое ли дерево. Если дерево не содержит элементов, то перекрашиваем добавленный элемент в черный и делаем его корнем дерева. Если дерево уже содержит элементы, тогда проверяем, чтобы не было повторяющихся слов.

Находим потенциального родителя для добавленного элемента, далее по значению добавляем его слева или справа (если значение меньше, чем значение родителя, тогда слева, если больше — справа).

С помощью дополнительных функций поворота и перекраски узлов сохраняем свойства красно-черных деревьев.

```

void RBTREE::insert(string str)
{
    Node* newNode = new Node(str);

    if (root == NULL)
    {
        newNode->color = BLACK;
        root = newNode;
    }
    else
    {
        Node* tmp = search(str);
        if (tmp->value == str)
        {
            cout << "Слово уже присутствует в словаре!" << endl;
            cout << "Расположение: " << endl;
            return;
        }
        newNode->parent = tmp;

        if (str < tmp->value)
            tmp->left = newNode;
        else
            tmp->right = newNode;

        fixRed(newNode);
    }
}

```

2.2.2 Балансировка красно-черного дерева при добавлении в него элемента

Функция: `void RBTREE::fixRed(Node* x).`

Вход функции: на вход функции подается указатель на узел, который был добавлен в дерево.

Выход функции: сбалансированное дерево с добавленным узлом.

Пример операции: см. рисунок 6.

Описание работы функции: для вставки узла используется функция `void RBTREE::insert(string str)`, которая вставляет узел в дерево, а затем окрашивает его в красный цвет. Для того, чтобы вставка сохраняла свойства красно-черных деревьев, вызывается данная вспомогательная функция, которая перекрашивает узлы и выполняет повороты.

```

void RBTREE::fixRed(Node* x)
{
    if (x == root)
    {
        x->color = BLACK;
        return;
    }
}

```

```

Node* parent = x->parent;
Node* grandparent = parent->parent;
Node* uncle = x->uncle();

if (parent->color != BLACK)
{
    if (uncle != NULL && uncle->color == RED)
    {
        parent->color = BLACK;
        uncle->color = BLACK;
        grandparent->color = RED;
        fixRed(grandparent);
    }
    else
    {
        if (parent->onLeft())
        {
            if (x->onLeft())
                chColor(parent, grandparent);
            else
            {
                leftR(parent);
                chColor(x, grandparent);
            }
            rightR(grandparent);
        }
        else
        {
            if (x->onLeft())
            {
                rightR(parent);
                chColor(x, grandparent);
            }
            else
            {
                chColor(parent, grandparent);
                leftR(grandparent);
            }
        }
    }
}
}

```

2.2.3 Удаление элемента

Функция: `void RBTREE::deleteNode(Node* x).`

Вход функции: на вход функции подается указатель на узел, который необходимо удалить..

Выход функции: красно-черное дерево с удаленным узлом.

Пример операции: см. рисунок 7 и 8.

Описание работы функции: функция удаления является рекурсивной. Сначала по

значению находим узел, который нужно удалить. При удалении узла необходимо учитывать его цвет, а точнее, если узел окрашен в черный цвет, то вызывается метод, который следит, чтобы черная высота дерева оставалась неизменной для каждой ветви.

Также узел проверяется на количество потомков:

- если у узла нет потомков, то просто удаляем этот узел;
- если узел имеет одного потомка, то меняем местами значения узлов и удаляем тот, который остался без потомков;
- если узел имеет двух потомков, тогда выбираем подходящий по значению узел и меняем их местами, второй потомок станет потомком первого. Следим за сохранением свойств с помощью поворотов и метода перекраски узлов.

```
void RBTREE::deleteNode(Node* x)
{
    Node* y = BSTreeplace(x);
    Node* parent = x->parent;

    bool xyBlack = ((y == NULL || y->color == BLACK) && (x->color == BLACK));

    if (y == NULL)
    {
        if (x == root)
            root = NULL;
        else
        {
            if (xyBlack)
                fixDBlack(x);
            else
            {
                if (x->subling() != NULL)
                    x->subling()->color = RED;
            }

            if (x->onLeft())
                parent->left = NULL;
            else
                parent->right = NULL;
        }
        delete x;
        return;
    }

    if (x->left == NULL || x->right == NULL)
    {
        if (x == root)
        {
            x->value = y->value;
            x->left = x->right = NULL;
            delete y;
        }
        else
        {

```

```

        if (x->onLeft())
        {
            parent->left = y;
            parent->place = 0;
        }
        else
        {
            parent->right = y;
            parent->place = 1;
        }
        delete x;

        y->parent = parent;
        if (xyBlack)
            fixDBlack(y);
        else
            y->color = BLACK;
    }
    return;
}
chValue(y, x);
deleteNode(y);
}

```

2.2.4 Балансировка красно-черного дерева при удалении элемента

Функция: `void RBTre::fixDBlack(Node* x).`

Вход функции: на вход функции подается указатель на узел.

Выход функции: сбалансированное дерево с удаленным узлом.

Пример операции: см. рисунок 7 и 8.

Описание работы функции: после удаления узла вызывается данная вспомогательная процедура, которая изменяет цвета и выполняет повороты.

```

void RBTre::fixDBlack(Node* x)
{
    if (x == root)
        return;

    Node* subling = x->subling();
    Node* parent = x->parent;

    if (subling == NULL)
    {
        fixDBlack(parent);
    }
    else
    {
        if (subling->color == RED)
        {
            parent->color = RED;

```

```

        subling->color = BLACK;
        if (subling->onLeft())
            rightR(parent);
        else
            leftR(parent);
        fixDBlack(x);
    }
else
{
    if (subling->redChild())
    {
        if (subling->left != NULL and subling->left->color == RED) {
            if (subling->onLeft())
            {
                subling->left->color = subling->color;
                subling->color = parent->color;
                rightR(parent);
            }
            else
            {
                subling->left->color = parent->color;
                rightR(subling);
                leftR(parent);
            }
        }
        else
        {
            if (subling->onLeft())
            {
                subling->right->color = parent->color;
                leftR(subling);
                rightR(parent);
            }
            else
            {
                subling->right->color = subling->color;
                subling->color = parent->color;
                leftR(parent);
            }
        }
        parent->color = BLACK;
    }
else
{
    subling->color = RED;
    if (parent->color == BLACK)
        fixDBlack(parent);
    else
        parent->color = BLACK;
}
}
}
}

```

2.2.5 Поиск элемента

Функция: `Node* RBTREE::search(string str)`.

Вход функции: на вход функции подается значение элемента, который необходимо найти.

Выход функции: найденный узел.

Пример операции: см. рисунок 9 и 10.

Описание работы функции: алгоритм поиска по красно-черному дереву простой и быстрый.

Для начала указываем на корень дерева. Далее последовательно сравниваем значение элемента, который мы ищем, со значением элемента, на который указывает указатель.

Исходя из результатов сравнения, перемещаемся вправо или влево по дереву (при этом указатель тоже нужно перемещать), пока не дойдем до нужного элемента.

```
Node* RBTREE::search(string str)
{
    Node* tmp = root;
    while (tmp != NULL)
    {
        if (str < tmp->value)
        {
            if (tmp->left == NULL)
                break;
            else
                tmp = tmp->left;
        }
        else if (str == tmp->value)
            break;
        else
        {
            if (tmp->right == NULL)
                break;
            else
                tmp = tmp->right;
        }
    }

    return tmp;
}
```

2.2.6 Полная очистка словаря

Функция: `void RBTREE::clear()`.

Вход функции: словарь в виде красно-черного дерева.

Выход функции: сообщение о том, что словарь удален.

Пример операции: см. рисунок 11.

Описание работы функции: функция полной очистки словаря реализована с помощью функции удаления элемента. Последовательно удаляются все элементы, пока дерево не станет пустым.

```
void RBTREE::clear()
{
    while (root != NULL)
    {
        deleteE(root->value);
    }
}
```

2.2.7 Добавление в словарь текста из файла

Функция: `void RBTREE::outFile(string str).`

Вход функции: название текстового файла.

Выход функции: дополненный словарь.

Пример операции: см. рисунок 12.

Описание работы функции: из файла считывается текст по одному слову, каждое из которых поочередно добавляется в словарь.

```
void RBTREE::outFile(string str)
{
    string s1, s2;
    ifstream file(str);
    string f_element("", .; :!? )(\ / ");

    while (getline(file, s1, ' '))
    {
        s2 = s1;
        if (s2.find_first_of("", .; :!? )(\ / ") != string::npos)
            s1.erase(s1.end() - 1);

        insertF(s1);
    }

    file.close();
}
```

2.3 Хэш-таблица

2.3.1 Построение хэш-таблицы

Функция: `void insertF(string str)`.

Вход функции: название текстового файла.

Выход функции: построенная хэш-таблица.

Пример операции: см. рисунок 15.

Описание работы функции: построение хэш-таблицы основывается на данной функции добавления элементов в таблицу.

Сначала функция проверяет нет ли в таблице текущего ключа (хэш-функции). Если ключ есть, то элемент добавляется в массив, который соответствует данному ключу, а если нет, то создается новый ключ и новый соответствующий массив.

```
void insertF(string str)
{
    HashVector.push_back(vector<string>());
    if (HashTable.find(str[0]) == HashTable.end())
    {
        HashVector[count1].push_back(str);
        HashTable[str[0]] = HashVector[count1];
        number.push_back(str[0]);
        count1 += 1;
    }
    else
    {
        for (int i = 0; i < number.size(); i++)
        {
            if (number[i] == str[0])
            {
                HashVector[i].push_back(str);
                HashTable[str[0]] = HashVector[i];
            }
        }
    }
}
```

2.3.2 Удаление элемента

Функция: `void deleteOrNot(string str)`.

Вход функции: имя элемента, который нужно удалить.

Выход функции: хэш-таблица с удаленным элементом.

Пример операции: см. рисунок 17, 18 и 19.

Описание работы функции: первым делом проверяется есть ли вообще ключ в таблице, если ключа нет, то нет и элементов по этому ключу, соответственно, выдается сообщение о том, что слово не найдено.

Далее, если вышеназванная проверка пройдена, проверяется есть ли такой элемент по текущему ключу. Если элемента нет, то выдается сообщение о том, что слово не найдено.

Если же ключ есть и элемент найден, то происходит удаление элемента, причем, если элемент по ключу один в массиве, то происходит и удаление ключа.

```
void deleteOrNot(string str)
{
    HashVector.push_back(vector<string>());
    if (HashTable.find(str[0]) == HashTable.end())
    {
        cout << "\nСлово '" << str << "' отсутствует в словаре!" << endl;
    }
    else
    {
        for (int i = 0; i < number.size(); i++)
        {
            if (number[i] == str[0])
            {
                auto it = find(HashVector[i].begin(), HashVector[i].end(), str);
                if (it == HashVector[i].end())
                {
                    cout << "Слово '" << str << "' отсутствует в словаре!" << endl;
                }
                else
                {
                    HashVector[i].erase(it);
                    HashTable[str[0]] = HashVector[i];
                    cout << "\nСлово удалено!" << endl;
                }
                if (HashVector[i].size() == 0)
                    HashTable.erase(str[0]);
            }
        }
    }
}
```

2.3.3 Поиск элемента

Функция: void searchE(string str).

Вход функции: имя элемента, который нужно найти.

Выход функции: найденный элемент.

Пример операции: см. рисунок 20 и 21.

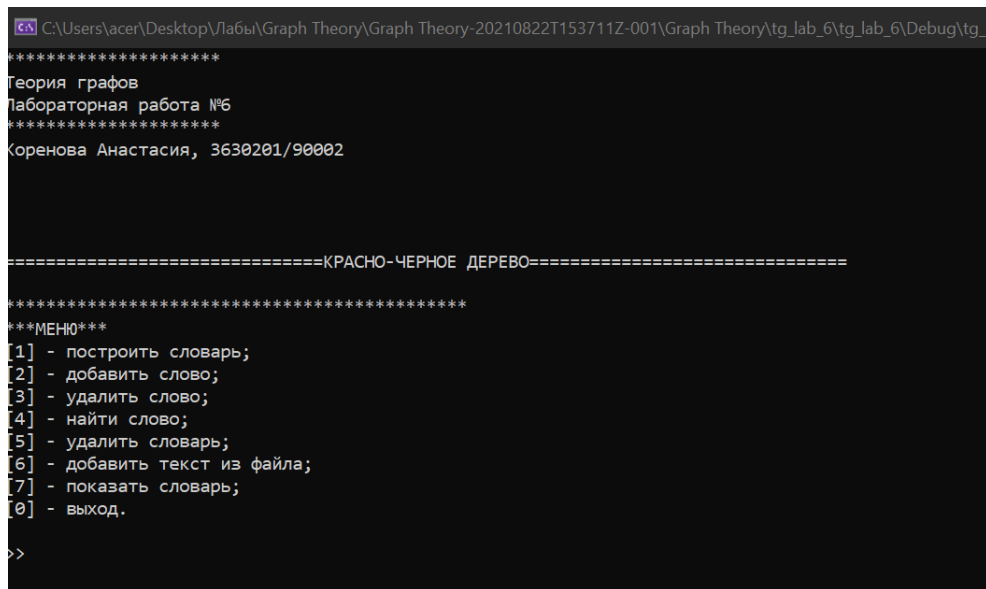
Описание работы функции: функция поиска элемента работает аналогично функции удаления, с той разницей, что найденный элемент не удаляется, а выводится на консоль.

```
void searchE(string str)
{
    HashVector.push_back(vector<string>());
    if (HashTable.find(str[0]) == HashTable.end())
    {
        cout << "\nСлово '" << str << "' отсутствует в словаре!" << endl;
    }
    else
    {
        for (int i = 0; i < number.size(); i++)
        {
            if (number[i] == str[0])
            {
                auto it = find(HashVector[i].begin(), HashVector[i].end(), str);
                if (it == HashVector[i].end())
                {
                    cout << "\nСлово '" << str << "' отсутствует в словаре!" << endl;
                }
                else
                {
                    cout << "\nСлово найдено!" << endl << endl;
                    auto itt = HashTable.find(str[0]);
                    cout << (*itt).first << ": ";
                    for (int i = 0; i < itt->second.size(); i++)
                    {
                        cout << itt->second[i] << " ";
                    }
                    cout << endl;
                }
            }
        }
    }
}
```

3 Результаты работы программы

3.1 Красно-черное дерево

На рисунках 5-13 показаны результаты работы основных операций со словарем, построенным на основе красно-черного дерева.



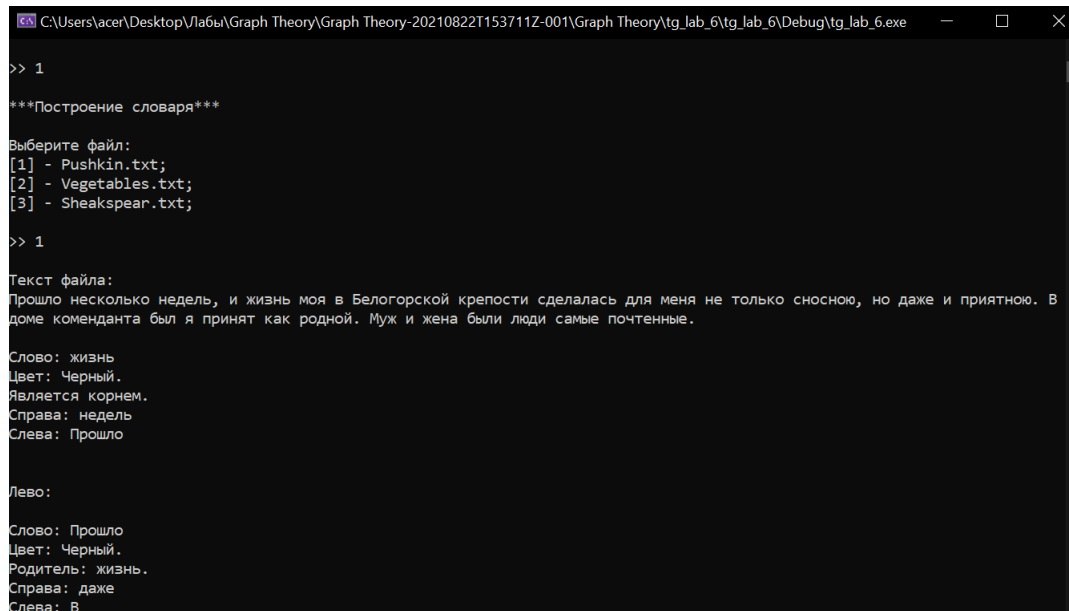
```
C:\Users\acer\Desktop\Лабы\Graph Theory\Graph Theory-20210822T153711Z-001\Graph Theory\tg_lab_6\tg_lab_6\Debug\tg_lab_6.exe
*****
Теория графов
Лабораторная работа №6
*****
Коренова Анастасия, 3630201/90002

=====КРАСНО-ЧЕРНОЕ ДЕРЕВО=====

*****
***МЕНЮ***
[1] - построить словарь;
[2] - добавить слово;
[3] - удалить слово;
[4] - найти слово;
[5] - удалить словарь;
[6] - добавить текст из файла;
[7] - показать словарь;
[0] - выход.

>>
```

Рис. 5: Меню для работы с красно-черным деревом



```
C:\Users\acer\Desktop\Лабы\Graph Theory\Graph Theory-20210822T153711Z-001\Graph Theory\tg_lab_6\tg_lab_6\Debug\tg_lab_6.exe
>> 1
***Построение словаря***
Выберите файл:
[1] - Pushkin.txt;
[2] - Vegetables.txt;
[3] - Sheakspear.txt;
>> 1
Текст файла:
Прошло несколько недель, и жизнь моя в Белогорской крепости сделалась для меня не только сносною, но даже и приятною. В
доме коменданта был я принят как родной. Муж и жена были люди самые почтенные.

Слово: жизнь
Цвет: Черный.
Является корнем.
Справа: недель
Слева: Прошло

Левое:

Слово: Прошло
Цвет: Черный.
Родитель: жизнь.
Справа: даже
Слева: В
```

Рис. 6: Построение красно-черного дерева, на основе выбранного файла

```
C:\Users\acer\Desktop\Лабы\Graph Theory\Graph Theory-20210822T1537

*****
***МЕНЮ***
[1] - построить словарь;
[2] - добавить слово;
[3] - удалить слово;
[4] - найти слово;
[5] - удалить словарь;
[6] - добавить текст из файла;
[7] - показать словарь;
[0] - выход.

>> 3

***Удаление слова***

Введите слово
>> родной

Слово удалено!

Выберите действие:
[1] - вывести весь словарь
[2] - продолжить.

>>
```

Рис. 7: Успешное удаление узла

```
C:\Users\acer\Desktop\Лабы\Graph Theory\Graph Theory-20210822T1537

>> 2

*****
***МЕНЮ***
[1] - построить словарь;
[2] - добавить слово;
[3] - удалить слово;
[4] - найти слово;
[5] - удалить словарь;
[6] - добавить текст из файла;
[7] - показать словарь;
[0] - выход.

>> 3

***Удаление слова***

Введите слово
>> баклажан
Слово баклажан не было найдено в словаре!

Выберите действие:
[1] - вывести весь словарь
[2] - продолжить.

>>
```

Рис. 8: Удаление узла, которое отсутствует в красно-черном дереве

```
C:\Users\acer\Desktop\Лабы\Graph Theory\Graph Theory-20210822T153711Z-001\Graph Theo
[0] - выход.

>> 4

***Поиск слова***

Введите слово:
>> сносню

Слово 'сносною' есть в словаре!

Цвет: Красный.
Родитель: только.
Справа: NIL Leaves
Цвет: Черный.
Значение: NULL.
Слева: NIL Leaves
Цвет: Черный.
Значение: NULL.

*****
```

Рис. 9: Успешный поиск заданного пользователем узла

```
C:\Users\acer\Desktop\Лабы\Graph Theory\Graph Theory-20210822T153711Z-001V
Значение: NULL.

*****
***МЕНЮ***
[1] - построить словарь;
[2] - добавить слово;
[3] - удалить слово;
[4] - найти слово;
[5] - удалить словарь;
[6] - добавить текст из файла;
[7] - показать словарь;
[0] - выход.

>> 4

***Поиск слова***

Введите слово:
>> баклажан

Слова 'баклажан' нет в словаре!

*****
```

Рис. 10: Поиск узла, которое отсутствует в красно-черном дереве

```
C:\Users\acer\Desktop\Лабы\Graph Theory\Graph Theory-20210822T153711Z-001\Graph T

*****
***МЕНЮ***
[1] - построить словарь;
[2] - добавить слово;
[3] - удалить слово;
[4] - найти слово;
[5] - удалить словарь;
[6] - добавить текст из файла;
[7] - показать словарь;
[0] - выход.

>> 5

***Удаление словаря***

Словарь удален!

*****
***МЕНЮ***
```

Рис. 11: Полное удаление словаря

```
C:\Users\acer\Desktop\Лабы\Graph Theory\Graph Theory-20210822T153711Z-001\Graph Theory\tg_lab_6

*****
***МЕНЮ***
[1] - построить словарь;
[2] - добавить слово;
[3] - удалить слово;
[4] - найти слово;
[5] - удалить словарь;
[6] - добавить текст из файла;
[7] - показать словарь;
[0] - выход.

>> 6

***Добавление текста из файла***

Выберите файл:
[1] - Pushkin.txt;
[2] - Vegetables.txt;
[3] - Sheakspear.txt;

>> 2

Слово: картофель
Цвет: Черный.
Является корнем.
Справа: огурец
Слева: артишок
```

Рис. 12: Дополнение словаря из файла


```
Выбрать C:\Users\acer\Desktop\Лабы\Graph Theory\Graph Theory-20210822T153711Z-001\Graph Theory\td

*****
***МЕНЮ***
[1] - построить словарь;
[2] - добавить слово;
[3] - удалить слово;
[4] - найти слово;
[5] - удалить словарь;
[6] - добавить текст из файла;
[7] - показать словарь;
[0] - выход.

>> 7

***Вывод словаря***

Слово: картофель
Цвет: Черный.
Является корнем.
Справа: огурец
Слева: артишок

Лево:

Слово: артишок
Цвет: Черный.
Родитель: картофель.
Справа: капуста
Слева: Баклажан
```

Рис. 13: Демонстрация словаря

3.2 Хэш-таблица

На рисунках 14-21 показаны результаты работы основных операций со словарем, построенным на основе хэш-таблицы.

```
C:\Users\acer\Desktop\Лабы\Graph Theory\Graph Theory-20210822T153711Z-001\Graph Theory\tg_lab_6\tg_lab_6.D

=====ХЭШ- ТАБЛИЦА=====

*****
***МЕНЮ***
[1] - построить словарь;
[2] - добавить слово;
[3] - удалить слово;
[4] - найти слово;
[5] - показать словарь;
[0] - выход.

>>
```

Рис. 14: Меню для работы с хэш-таблицей

```
C:\Users\acer\Desktop\Лабы\Graph Theory\Graph Theory-20210822T153711Z-001\Graph Theory\tg_lab_6\tg_lab_6\Debug\tg_lab_6.exe
Выберите файл:
[1] - products.txt;
[2] - murakami.txt;
[3] - gaiman.txt;

>> 1

Текст файла:
Картофель, молоко, лимон, огурец, лапша, курица, моцарелла, руккола, картофель, вино, морковь, виноград, лайм, мандарины,
, жимолость, нектарины.

***СЛОВАРЬ***
К: Картофель
в: вино виноград
ж: жимолость
к: курица картофель
л: лимон лапша лайм
м: молоко моцарелла морковь мандарины
н: нектарины
о: огурец
р: руккола

*****
```

Рис. 15: Построение хэш-таблицы

```
C:\Users\acer\Desktop\Лабы\Graph Theory\Graph Theory-20210822T153711Z-001\Graph Theory\tg_lab_6\tg_lab_6\Debug\tg_lab_6.exe
>> 2

***Добавление слова***

Введите слово
>> кефир

Слово: кефир

Выберите действие:
[1] - вывести весь словарь
[2] - продолжить.

>> 1

***СЛОВАРЬ***
К: Картофель
в: вино виноград
ж: жимолость
к: курица картофель кефир
л: лимон лапша лайм
м: молоко моцарелла морковь мандарины
н: нектарины
о: огурец
р: руккола

*****
```

Рис. 16: Добавление элемента в словарь


```
C:\Users\acer\Desktop\Лабы\Graph Theory\Graph Theory-20210822T153711Z-001\Graph
***Удаление слова***

Введите слово
>> жимолость

Слово удалено!

Выберите действие:
[1] - вывести весь словарь
[2] - продолжить.

>> 1

***СЛОВАРЬ***
К: Картофель
в: вино виноград
к: курица картофель
л: лимон лапша лайм
м: молоко моцарелла морковь мандарины
н: нектарины
о: огурец
р: руккола
*****
```

Рис. 19: Удаление выбранного элемента и ключа

```
C:\Users\acer\Desktop\Лабы\Graph Theory\Graph Theory-20210822T153
>> 4

***Поиск слова***

Введите слово:
>> курица

Слово найдено!

к: курица картофель
*****
***МЕНЮ***
[1] - построить словарь;
[2] - добавить слово;
[3] - удалить слово;
[4] - найти слово;
[5] - показать словарь;
[0] - выход.

>>
```

Рис. 20: Успешный поиск выбранного элемента

```
C:\Users\acer\Desktop\Лабы\Graph Theory\Graph Theory-2021082

>> 4

***Поиск слова***

Введите слово:
>> дрова

Слово 'дрова' отсутствует в словаре!

*****
***МЕНЮ***
[1] - построить словарь;
[2] - добавить слово;
[3] - удалить слово;
[4] - найти слово;
[5] - показать словарь;
[0] - выход.

>>
```

Рис. 21: Поиск выбранного элемента, который отсутствует в словаре

Заключение

В результате выполнения данной лабораторной работы различными способами было реализовано приложение «Словарь». В качестве реализации было использовано два варианта: **красно-черное дерево** и **хэш-таблица**.

Приложение позволяет выполнять такие операции с элементами словаря, как:

- *составление словаря* посредством чтения текста из файла;
- *добавление* нового элемента в словарь;
- *удаление* выбранного элемента из словаря;
- *поиск* выбранного элемента по словарю;
- *дополнение* словаря элементами из файла;
- *полная очистка* словаря.

Использованные структуры позволяют эффективно реализовывать данные операции. Красно-черному дереву в этом помогает его особая структура раскраски, а хэш-таблице — использование ассоциативной памяти.

Красно-черное дерево имеет более быстрый алгоритм поиска ($O(\log_2 N)$ у красно-черного дерева, $O(1)$ — у хэш-таблицы в лучшем случае и $O(N)$ — в худшем), что очень важно при использовании словаря. Кроме того, элементы красно-черного дерева могут быть легко выведены в порядке возрастания, что в нашем случае является лексикографическим порядком, который и является более эффективным.

Однако, при минимизации коллизий подбором хэш-функции и использование повторного хэширования хэш-таблица может оказаться эффективнее красно-черного дерева в поиске элементов.

Достоинства реализации:

- возможность просмотра полной информации об элементе (родитель, потомок, цвет) за счет реализации класса `Node`;
- реализованный класс, абстрагирующий красно-черное дерево, может быть полезен за счет эффективности реализованных операций;
- возможность масштабирования программы за счет выбора другой хэш-функции (например, полиномиальное хэширование $h(S) = S[0] + S[1]*P + S[2]*P^2 + \dots + S[N]*P^N$).

Недостатки реализации:

- загрузка данных в словарь осуществляется из заранее выбранных файлов;
- выбранная хэш-функция порождает множество коллизий.

Список литературы

- [1] Ф.А.Новиков. Дискретная математика для программистов. СПб: Питер Пресс, 2009г. 364с.
- [2] А.В.Востров. Курс лекций по дискретной математике <https://tema.spbstu.ru/tgraphlect/> (последнее посещение 29.08.2021).
- [3] Хэш-таблицы [электронный ресурс]. URL: http://algotist.ru/ds/s_has.php (последнее посещение 26.08.2021).
- [4] Красно-черные деревья [электронный ресурс]. URL: <https://neerc.ifmo.ru/wiki/index.php?title=> (последнее посещение 25.08.2021).