МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

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

Институт Компьютерных Наук и Технологий

Высшая школа Искуственного Интелекта

Направление 02.03.01 Математика и Компьютерные Науки

Отчёт по дисциплине:

«Теория графов»

Лабораторная работа № 6

Реализация словаря на основе красно - чёрного дерева и хэш - таблицы.

Обучающийся: _____ Черепанов Михаил Дмитриевич

Руководитель: _____ Востров Алексей Владимирович

«____» _____ 20__ г.

Санкт-Петербург 2023

Содержание

3

Термины и определения **4**

1 Математическое описание **5**
1.1 Красно-Черное Дерево 5
1.1.1 Определение и свойства 5
1.1.2 Балансировка при вставке 6
1.1.3 Балансировка при удалении 7
1.2 Хэш-Таблица 8
1.2.1 Определение и свойства 9

2 Особенности реализации **9**
2.1 Красно-Черное Дерево 9
2.1.1 Класс Node 9
2.1.2 Класс Tree 11
2.1.3 Вставка элемента. Метод Insert. 11
2.1.4 Удаление элемента. Метод Remove 12
2.1.5 Поиск элемента 15
2.2 Хэш-таблица 16
2.2.1 Реализация хэш-функции 16
2.2.2 Добавление элемента 16
2.2.3 Удаление элемента 18
2.2.4 Методы Resize и Rehash 18

3 Результаты работы программы **20**

Заключение **22**

Список используемой литературы **23**

Введение

В лабораторной работе требуется реализовать приложение с функционалом словаря. Словарь реализовать на основе красно-черного дерева и хэш-таблицы. Необходимо реализовать функции добавления, удаления и поиска ключа, функцию полной очистки словаря и загрузки/дополнения словаря из текстового файла.

Термины и определения

В настоящем отчете о применяют следующие термины в контексте бинарных деревьев:

- Сын(Потомок, Ребенок): Сын - это узел, находящийся ниже (по уровню) другого узла в дереве, который непосредственно связан с этим узлом в направлении отца (родителя). Узел может иметь ноль, одного или двух сыновей.
- Отец(Родитель): Отец - это узел, который находится выше (по уровню) другого узла в дереве и связан напрямую с этим узлом в направлении сына (детей). Каждый узел, за исключением корневого узла, имеет ровно одного родителя.
- Дед: Дед - это узел, который является родителем родителя. То есть, если узел X имеет родителя Y, и Y имеет родителя Z, то Z будет дедушкой для X.
- Дядя: Дядя - это брат родителя узла. Если узел X имеет родителя Y, а Y имеет брата Z, то Z будет дядей для X.
- Внук: Внук - это сын сына. Если узел X имеет сына Y, а Y имеет собственного сына Z, то Z будет внуком для X.
- Брат: Брат - это узел, который имеет общего родителя с данным узлом. Если узел X имеет родителя Y, и Y имеет другого сына (не X), то этот другой сын будет братом для X.

1 Математическое описание

1.1 Красно-Черное Дерево

1.1.1 Определение и свойства

Красно-черное дерево (Red-Black Tree) - это бинарное дерево поиска, которое удовлетворяет следующим свойствам:

1. Каждый узел дерева имеет цвет, который может быть либо красным, либо черным (то есть отмечены либо нулем либо единицей);
2. Корень дерева всегда черный;
3. Каждый лист (NIL-узел) дерева, который не содержит данных (как пустые листья в обычных бинарных деревьях), также является черным;
4. Если узел красный, то оба его потомка должны быть черными (это означает, что красные узлы не могут идти друг за другом);
5. Для каждого узла в дереве, любой простой путь от этого узла к любому из его листьев должен содержать одинаковое количество черных узлов. Это свойство обеспечивает, что дерево сбалансировано и гарантирует логарифмическую сложность операций вставки, удаления и поиска;

Для поддержания указанных выше свойств красно-черного дерева используют перекрашивание узлов и повороты (левый и правый относительно одного из узлов). Алгоритм выполнения левого поворота:

1. Переместите правого потомка черного узла в позицию родителя.
2. Обновите ссылки у родителя и правого потомка, а также у их родителей.
3. Установите правого потомка как нового родителя черного узла.

На рис.: 1 показан пример красно-черного дерева.

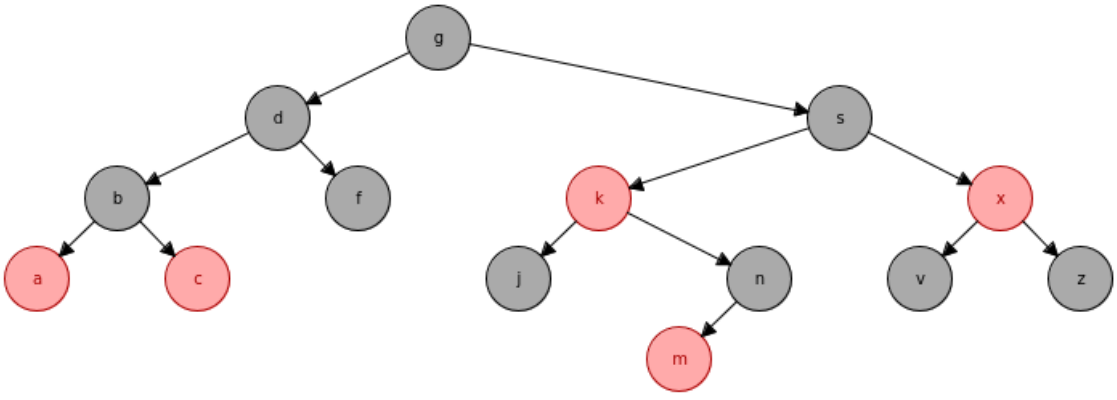

Рис. 1: Пример красно-черного дерева.

На рис.: 2 и 3 графически показаны положения дерева до и после левого поворота, при вставке нового элемента.

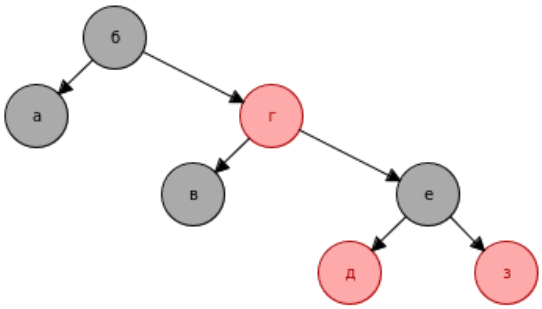

Рис. 2: Дерево до левого поворота.

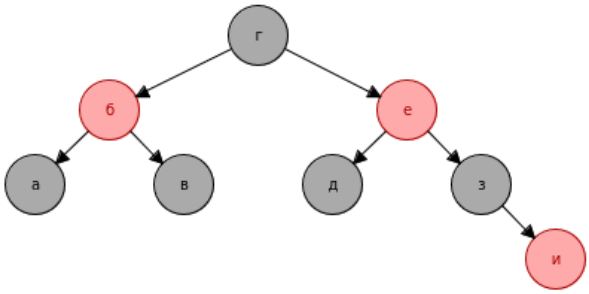

Рис. 3: Дерево после левого поворота.

1.1.2 Балансировка при вставке

Алгоритм вставки нового узла:

1. Вставить новый узел N в дерево как в обычное бинарное дерево поиска, сохраняя порядок элементов.
2. Окрасить новый узел в красный цвет.
3. Если родитель черный, ни одно из свойств Красно-Черного дерева не нарушается. Выйти из алгоритма. Иначе:
4. Если дядя - красный: Красим отца и дядю в черный,деда в красный(если он не является корнем дерева);
   Считаем что вставляемый узел является дедом текущего, переходим к пункту 3.
5. Если дядя - черный: выполняем левый поворот, если N - левый сын, правый поворот, если N - правый сын.
   Красим отца N в черный, брата в красный.

Считаем что вставляемый узел является дедом текущего, переходим к пункту 3.

1.1.3 Балансировка при удалении

Алгоритм удаления узла:

1. Пусть N - удаляемый узел.
2. N - узел с двумя детьми:
   Находим следующий за ним по возрастанию элемент (у него не будет левого сына), меняем значения между узлами, приравниваем N следующему элементу переходим к следующему пункту;
3. N - черный узел с одним ребёнком:
   Переносим значение существующего потомка в узел N. Удаляем потомка.
4. N - красный узел без детей:
   Удаляем узел N. Ни одно из свойств RBT не нарушилось.
5. N - красный узел без детей:
6. N - черный узел без детей (самый напряженный случай):
   Для наглядности предположим, что узел N - правый сын (балансировка работает зеркально, то есть просто все слова «левый» и «правый» заменяем на противоположные.
   (a) Родитель N - красный, брат черный, 2 сына брата черные:
   Меняем местами цвета у отца и брата.
   (b) Родитель N - красный, брат черный, левый сын брата красный:
   Меняем местами цвета у отца и брата;
   Выполняем правый поворот вокруг родителя, красим левого сына в черный;
   (c) Родитель N - чёрный, левый сын красный, у правого внука чёрные дети;
   Красим правого внука в красный, а красного правнука в черный. Выполняем правый поворот вокруг родителя;
   (d) родитель N - чёрный, левый сын красный, у правого внука левый сын красный:
   Выпоняем правый поворот вокруг брата, затем левый вокруг родителя;
   Перекрашиваем красного правнука в черный.
   (e) Родитель N - чёрный, левый сын черный, у дяди правый сын - красный:
   Выпоняем правый поворот вокруг брата, затем левый вокруг родителя;
   Перекрашиваем красного внука в черный.

(f) Родитель N - чёрный, левый сын чёрный, его внуки тоже чёрные
Красим брата в красный.
Приравниваем N деду удаляемого узла, переходим к началу алгоритма.

1.2 Хэш-Таблица

Хэш-таблица - это структура данных, которая используется для хранения и организации данных с использованием преобразования информации с помощью особых математических формул. Она позволяет эффективно выполнять операции вставки, удаления и поиска элементов по ключу. Основная идея заключается в том, что каждому элементу (значению) назначается уникальный индекс (хэш-код) с использованием хэш-функции, и этот индекс используется для быстрого доступа к элементу в таблице.

Хэш-функция - это функция, которая преобразует ключ элемента в числовой индекс (хэш-код). Хорошо разработанная хэш-функция должна минимизировать коллизии (ситуации, когда два разных ключа дают один и тот же хэш-код).

Когда два или более ключа дают один и тот же хэш-код, необходим механизм разрешения коллизий.

В данной лабораторной работе был использован метод двойного хэширования для разрешения колизий. Суть метода заключается в том, что при хэшировании элемента хэш-функция считает два значения: место в массиве, в которое необходимо поместить элемент и шаг, с которым этот индекс будет сдвигаться, если место уже занято.

Формула моей хэщ-функции:

$result = mod(((((key * result + \|s[i]\|) * table\_size) * 2) + 1), table\_size)$,

где:

table_size - размер таблицы;
key- параметр, параметр хеширования, зависимый от размеров таблицы
s[i] - код i-ого символа кодируемой строки, 0<i<string.size() - размер строки;

На рис.: 4 показан пример хэш-таблицы, сгенерированной данной хэш-функцией.

Рис. 4: Пример хэш-таблицы.

2 Особенности реализации

2.1 Красно-Черное Дерево

В данной реализации использованы были разработаны два класса: Node, описывающий узел дерева и Tree, в котором реализованы операции с деревом. Все дейсвтвия с пользователем описаны в main.

2.1.1 Класс Node

В данном классе хранятся данные, хранимые в узле, его цвет, указатели на родителя и потомков узла. Используется шаблонный класс для хранения информации, отличной от типа string. Перегружены операторы равенства и присваивания.

```
template <typename T>
class Node
{
public:
    T data;
    COLOR color;
    Node* parent;
    Node*left;
    Node* right;
    friend class Tree;

     Node(){
          data=0;
          color=BLACK;
          parent=nullptr;
          left=nullptr;
          right=nullptr;
     };
    Node(const T& d,COLOR c=BLACK){
         data=d;
         color=c;
         parent=nullptr;
         left=nullptr;
         right=nullptr;
    }
    Node(const T& d,COLOR c,Node* p){
         data=d;
         color=c;
         parent=p;
         left=nullptr;
         right=nullptr;
    }
```

```
    Node& operator=(const Node& other) {
        if (this != &other) {
            data = other.data;
            color = other.color;
            parent = other.parent;
            left = other.left;
            right = other.right;
        }
        return *this;
    }
    bool operator==(const Node& other) const {
        return (data == other.data &&
                color == other.color &&
                parent == other.parent &&
                left == other.left &&
                right == other.right);
    }
};
```

2.1.2 Класс Tree

Данный класс содержит поля: size - количество элементов в дереве и root - указатель на корневой элемент, а так же функции для работы с красно-черным деревом.

2.1.3 Вставка элемента. Метод Insert.

вход: Дерево с количеством элементов N и значением элемента, который необходимо добавить в дерево.

выход: Дерево с количесвтом элементов N+1.

Вставка элемента реализована в методе Insert. Алгоритм вставки подробно описан в математическом описании.

```
  void Insert(const T& val) {
        int ty=0;
        ty++;
        Node<T>* new_node;
        bool OK=1;
        InsertBin(root,val,new_node,OK);
        if(OK==0)
            return;
        size++;
        if(size==1){
            root=new_node;
```

```
            new_node->color=BLACK;
            return;
        }
        new_node->color=RED;
        while(new_node->parent!=nullptr)
        {//GetGrandFather(new_node)!=NULL&&
         GetGrandFather(new_node)!=root
            Node<T>*p=new_node->parent;
            if (new_node->color == RED && p->color == RED) {
                Node<T>* u = GetUncle(new_node);
                if(u != nullptr && u->color==RED){
                    p->color=BLACK;
                    u->color=BLACK;
                    p->parent->color = RED;
                    p = p->parent;
                }
                else{
                    bool p_is_r_s=IsRightSon(p);
                    if(IsRightSon(p)^
                     IsRightSon(new_node)){
                        p=p_is_r_s? RightRotation(p)
                        : LeftRotation(p);
                    }
                    p=p_is_r_s?LeftRotation(p->parent)
                    : RightRotation(p->parent);
                    p->color = BLACK;
                    if(p->left!= nullptr)
                     p->left->color = RED;
                    if(p->right!= nullptr)
                     p->right->color = RED;
                }
            }
            new_node = p;
        }
        SetRoot(root);
        root->color=BLACK;
    };
```

2.1.4 Удаление элемента. Метод Remove

вход: Дерево с количеством элементов N и значением элемента, который необходимо удалить из структуры.

выход: Дерево с количесвтом элементов N-1.

Удаление элеменета реализовано в методе Remove. Если цвет удаляемого узла черный, то после удаления вызывается функция deleteFixup, выполняющая баланси-

ровку дерева.
Алгоритм балансировки дерева при удалении подробно описан в математическом описании.

```
void Remove(const T& word)
    {
        Node<T>* node = SearchRec(root,word );

        if (node == nullptr)
            return;
        size--;
        Node<T>* toDelete;
        Node<T>* toFix;

        if (node->left == nullptr || node->right == nullptr)
            toDelete = node;
        else
            toDelete = successor(node);
        if (toDelete->left != nullptr)
            toFix = toDelete->left;
        else
            toFix = toDelete->right;

        if (toFix != nullptr)
            toFix->parent = toDelete->parent;
        if (toDelete->parent == nullptr)
            root = toFix;
        else if (toDelete == toDelete->parent->left)
            toDelete->parent->left = toFix;
        else
            toDelete->parent->right = toFix;
        if (toDelete != node)
            node->data = toDelete->data;
        if (toDelete->color == BLACK) {
            deleteFixup(toFix, toDelete->parent);
        }
        delete toDelete;
        SetRoot(root);
    }
```

Функция deleteFixup
Вход Дерево с удаленным узлом, с некоторыми нарушенными свойствами красно-черного дерева, узел, вставший на место удаленного, родитель удаленного узла.
Выход Сбалансированное красно-черное дерево.

Алгоритм работы этой функции также описан в мат. описании.

```
void deleteFixup(Node<T>* node, Node<T>* parent)
    {
        Node<T>* sibling;
        while (node != root && (node == nullptr
        || node->color == BLACK)) {
            if (node == parent->left) {
                sibling = parent->right;

                if (sibling->color == RED) {
                    sibling->color = BLACK;
                    parent->color = RED;
                    LeftRotation(parent);
                    sibling = parent->right;

                }
                if ((sibling->left == nullptr
                || sibling->left->color == BLACK)
                 && (sibling->right == nullptr
                 || sibling->right->color == BLACK)) {
                    sibling->color = RED;
                    node = parent;
                    parent = node->parent;

                }
                else {
                    if (sibling->right == nullptr
                    || sibling->right->color == BLACK)
                     {
                        sibling->left->color = BLACK;
                        sibling->color = RED;
                        RightRotation(sibling);
                        sibling = parent->right;
                    }
                    sibling->color = parent->color;
                    parent->color = BLACK;
                    sibling->right->color = BLACK;
                    LeftRotation(parent);
                    node = root;
                    break;
                }
            }
            else {
                sibling = parent->left;
```

```
                if (sibling->color == RED) {
                    sibling->color = BLACK;
                    parent->color = RED;
                    RightRotation(parent);
                    sibling = parent->left;
                }


                if ((sibling->left == nullptr
                 || sibling->left->color == BLACK)
                  && (sibling->right == nullptr
                   || sibling->right->color == BLACK)) {
                    sibling->color = RED;
                    node = parent;
                    parent = node->parent;

                }
                else {
                    if (sibling->left == nullptr
                    || sibling->left->color == BLACK) {
                        sibling->right->color = BLACK;
                        sibling->color = RED;
                        LeftRotation(sibling);
                        sibling = parent->left;
                    }
                    sibling->color = parent->color;
                    parent->color = BLACK;
                    sibling->left->color = BLACK;
                    RightRotation(parent);
                    node = root;
                    break;
                }
            }
        }
        if (node != nullptr) {
            node->color = BLACK;
        }
    }
```

2.1.5 Поиск элемента

Поиск элемента осуществляется в функции SearchRec и реализован рекурсивно.

Вход Слово для поиска.
Выход Указатель на найденный элемент или nullptr, если элемент не найден.

```
Node<T>* SearchRec(Node<T>* node, const T& value) {
    if (node == nullptr)
        return nullptr;
    if (value == node->data)
        return node;
    else if (value < node->data)
        return SearchRec(node->left, value);
    else
        return SearchRec(node->right, value);
}
```

2.2 Хэш-таблица

Для реализации хэш-таблицы был использован массив структур, содержащих хранящиеся данные и флаг того, удалена ли эта информация из таблицы.

```
struct Node
{
    T value;
    bool state;
    Node(const T& value_) : value(value_), state(true) {}
};
```

2.2.1 Реализация хэш-функции

Вход Строка для вставки в таблицу, размер таблицы, ключ(вспомогательное число для кеширования.

Выход Результат хэширования(целочисленный индекс элемента).

Итоговое число получается путем прибавления целочисленного кода символа к значению (ключ * размер таблицы) и применения к этому значению операции остаток от деления на каждой итерации.

```
static int HashFunction
(const std::string& s, int table_size, const int key)
{
    int hash_result = 0;
    for (int i = 0; i< s.size(); ++i) {
        hash_result = (key * hash_result + s[i]) % table_size;
    }
    hash_result = (hash_result * 2 + 1) % table_size;
```

```
    return hash_result;
}
```

2.2.2 Добавление элемента

Вход Слово, которое необходимо добавить.

Выход Хэш-таблица с добавленным элементом.

В теле функции считаются значения индекса для добавления элемента и значение шага смещения. Далее перебираются все элементы массива, начиная с указанного индекса и с нужным шагом. Если в массиве уже есть данный элемент, выходим из алгоритма (в хэш-таблице не может быть двух одинаковых элементов). Как только находим свободное место, выходим из алгоритма и вставляем на свободное место нужный нам элемент.

Кроме этого, если в таблице больше половины удаленных элементов (как мы помним при удалении элементы просто помечаются флагом) происходит перехеширование данных (об этом дальше). При заполнении 0.75 от всей таблицы, ее размер увеличивается в два раза.

```
bool Add(const T& value)
{
    THash1 hash1 = THash1();
    THash2 hash2 = THash2();
    if (size + 1 > int(rehash_size * buffer_size))
        Resize();
    else if (size_all_non_nullptr > 2 * size)
        Rehash(); //   ,      deleted-
    int h1 = hash1(value, buffer_size);
    int h2 = hash2(value, buffer_size);
    int i = 0;
    int first_deleted = -1; //
    while (arr[h1] != nullptr && i < buffer_size)
    {
        if (arr[h1]->value == value && arr[h1]->state)
            return false; //     ,
        if (!arr[h1]->state && first_deleted == -1)
        {
            first_deleted = h1;
            break;                  //
        }
        h1 = (h1 + h2) % buffer_size;
        ++i;
    }
    if (first_deleted == -1)
```

```
        {
            arr[h1] = new Node(value);
            ++size_all_non_nullptr;
        }
        else
        {
            arr[first_deleted]->value = value;
            arr[first_deleted]->state = true;
        }
        ++size; //
        return true;
    }
```

2.2.3 Удаление элемента

Вход Слово, которое необходимо удалить.

Выход true - если элемент удален, если элемента не существовало - false.

В теле функции считаются значения индекса для удаления элемента и значение шага смещения. Проверяется каждый элемент таблицы начиная с посчитанного индекса с данным шагом. Если найденный элемент равен удаляемому, флаг элементы переходит в false, выходим из алгоритма, элемент удален, возврати true. Если были проверены все элементы таблицы и не нашли подходящего, то возвращяем false.

```
bool Remove(const T& value )
  {
      THash1 hash1 = THash1();
      THash2 hash2 = THash2();
      int h1 = hash1(value, buffer_size);
      int h2 = hash2(value, buffer_size);
      int i = 0;
      while (arr[h1] != nullptr && i < buffer_size)
      {
          if (arr[h1]->value == value && arr[h1]->state)
          {
              arr[h1]->state = false;
              --size;
              return true;
          }
          h1 = (h1 + h2) % buffer_size;
          ++i;
      }
      return false;
```

```
}
```

2.2.4 Методы Resize и Rehash

Вход Текущая хэш-таблица.

Выход Новая хэш-таблица с отсутсвием удаленных элементов.

В обоих функциях создается новый массив элементов (при Resize новый массив в два раза больше). Далее для каждого ненулевого и неудаленного элемента текущего массива вызывается функция добавления в новый массив. Таким образом мы избавляемся от всех удаленных элементов. Старый массив удаляется.

```
void Resize()
    {
        int past_buffer_size = buffer_size;
        buffer_size *= 2;
        size_all_non_nullptr = 0;
        size = 0;
        Node** arr2 = new Node * [buffer_size];
        for (int i = 0; i < buffer_size; ++i)
            arr2[i] = nullptr;
        std::swap(arr, arr2);
        for (int i = 0; i < past_buffer_size; ++i)
        {
            if (arr2[i] && arr2[i]->state)
                Add(arr2[i]->value);
        }

        for (int i = 0; i < past_buffer_size; ++i)
            if (arr2[i])
                delete arr2[i];
        delete[] arr2;
    }
    void Rehash()
    {
        size_all_non_nullptr = 0;
        size = 0;
        Node** arr2 = new Node * [buffer_size];
        for (int i = 0; i < buffer_size; ++i)
            arr2[i] = nullptr;
        std::swap(arr, arr2);
        for (int i = 0; i < buffer_size; ++i)
        {
            if (arr2[i] && arr2[i]->state)
```

```
            Add(arr2[i]->value);
        }

        for (int i = 0; i < buffer_size; ++i)
            if (arr2[i])
                delete arr2[i];
        delete[] arr2;
    }
```

---

3 Результаты работы программы

На рис. 5 - 8 показаны результаты выполнения операций с красно-черным деревом.

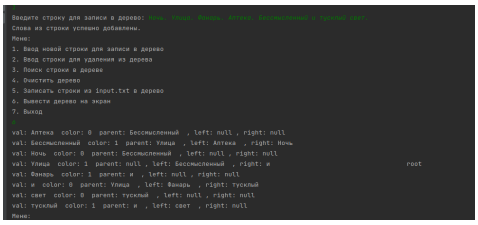

Рис. 5: Добавление строки.

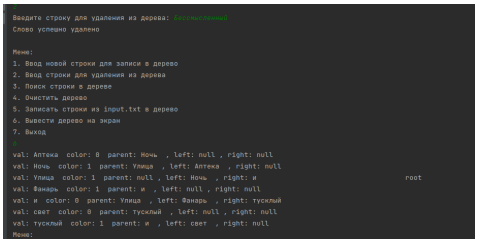

Рис. 6: Удаление элемента.

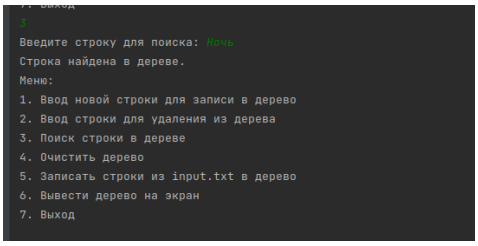

Рис. 7: Поиск элемента.

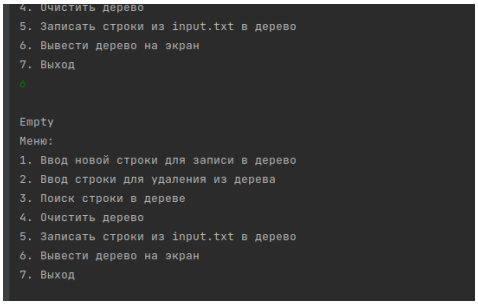

Рис. 8: Очистка дерева.

На рис. 9 - 12 показаны результаты выполнения операций с хэш-таблицей.

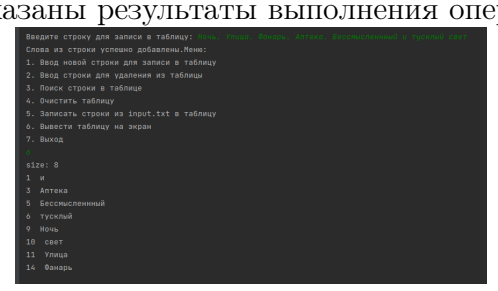

Рис. 9: Добавление строки.

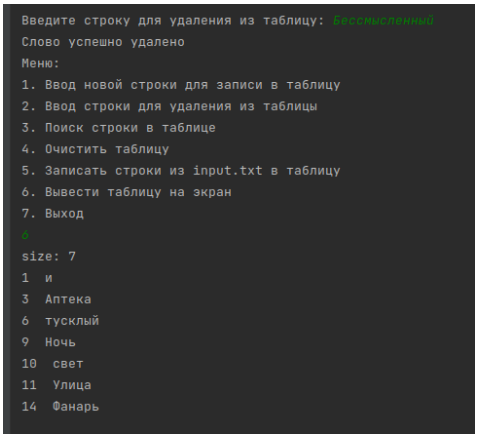

Рис. 10: Удаление нескольких элементов.

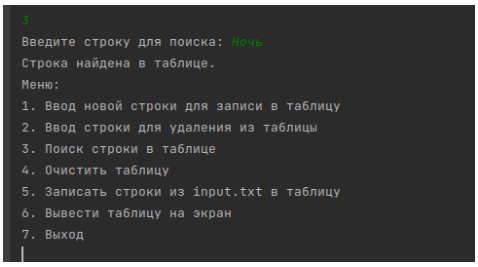

Рис. 11: Поиск элемента.

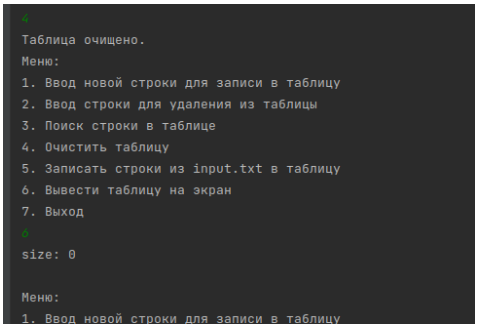

Рис. 12: Очистка дерева.

Заключение

В результате работы были реализованы такие структуры данных как красно-черное дерево и хэш-таблица.

Достоинства реализации:

Использование шаблонных классов при реализации красно-черного дерева, из-за чего в нем можно хранить не только строки;

Использование хэш-функции, которая создает код элемента в зависимости от каждого символа, что уменьшает вероятность возникновения колизий;

Использование двойного хеширования для избежания коллизий;

Недостатки реализации:

Использование метода SetRoot, который востанавливает указатель на корневого пользователя каждый раз при изменении красно-черного дерева, в отличие от того, чтобы в методах, балансирующих дерево контролировать указатель на корень. Это добовляет ненужные итерации.

Масштабирование:

В работе реализованы основыне методы для работы с Красно-черным деревом и хэш-таблицей. С помощью реализованных функций возможно расширение функционала добавлением поиска по критериям, например фильтр по значениям.

Список литературы

[1] Ф.А.Новиков. Дискретная математика для программистов. СПб: Питер Пресс, 2009г. 364с.

[2] КОМПЬЮТЕРНАЯ МАТЕМАТИКА. Д.Кук, Г.Бейз. М.: Наука. Гл. ред. физ.-мат. лит.,1990, 384 с.