МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение
высшего образования «Санкт-Петербургский политехнический
университет Петра Великого»

Институт компьютерных наук и кибербезопасности

Высшая школа технологий искусственного интеллекта

Направление: 02.03.01 Математика и компьютерные науки

Отчет о выполнении лабораторной работы №6

«Реализация словаря и сжатие данных»

Дисциплина «Теория графов»

Вариант 8

Выполнил студент группы

№5130201/30001 _____ Мелещенко С.И.

Проверил _____ Востров А. В.

Санкт-Петербург, 2025

Содержание

Введение

Данный отчет содержит в себе описание выполнения 6 лабораторной работы.

1. Для выбранного текста на русском языке реализовать структуру данных - хеш-таблицу, для которой характерны операции: добавления, удаления и поиска. На ее основе реализовать словарь. Хеш-функция выбрать произвольную (в отчете описать ее качество и устойчивость к коллизиям). Минимальная реализация хэш-функции - первая буква слова (невысокий балл), за более сложные хэш-функции - высокие баллы. Добавить функцию полной очистки словаря и загрузки/дополнения словаря из текстового файла.

2. Для этого же текста на русском языке построить словарь на основе красно-черных деревьев. Реализовать функции добавления, удаления и поиска слова. (НЕЛЬЗЯ использовать готовые структуры данных!!!). Добавить функцию полной очистки словаря и загрузки/дополнения словаря из текстового файла.

3. а) Случайно сгенерировать файл в 10 тысяч символов (в соответствии с заданным распределением - Парето), используя русский алфавит, цифры и спецсимволы. Закодировать текстовую информацию, используя указанный в задании алгоритм. Определить цену кодирования (если это возможно). Декодировать информацию, определить коэффициент сжатия. Программно проверить, что декодирование произошло верно.

С - Алгоритм LZW.

б) Аналогично предыдущему пункту закодировать информацию, применив двухступенчатое кодирование и декодирование. Показать, какой из способов более эффективный.

4. В соотвествии с вариантом применить алгоритм сжатия тектового файла и декодирование к исходному текстовому файлу со словарем.

Дополнительная реализация включает возможность оптимизации словаря (например, поиск потенциальных однокоренных слов в разных формах; самый простой критерий - 2/3 длины слова). Выбор слов, которые следует оставить в словаре - за пользователем. Использовать готовые решения из библиотеки STL не разрешается.

Программа была реализована на языке программирования C++ в среде программирования Microsoft Visual Studio.

1 Математическое описание

1.1 Хэш-таблица

Хеш-таблица — структура данных, реализующая интерфейс ассоциативного массива. В отличие от деревьев поиска, реализующих тот же интерфейс, обеспечивают меньшее время отклика в среднем. Представляет собой эффективную структуру данных для реализации словарей, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.

Существует два основных вида хеш-таблиц: с цепочками и открытой адресацией. Хеш-таблица содержит некоторый массив H , элементы которого есть пары (хеш-таблица с открытой адресацией) или списки пар (хеш-таблица с цепочками).

Выполнение операции в хеш-таблице начинается с вычисления хеш-функции от ключа. Хеш-код i=h(key) играет роль индекса в массиве H , а зная индекс, мы можем выполнить требующуюся операцию (добавление, удаление или поиск).

Количество коллизий зависит от хеш-функции; чем лучше используемая хеш-функция, тем меньше вероятность их возникновения.

Пример моей хэш-таблицы (см. [Рис. 1]).

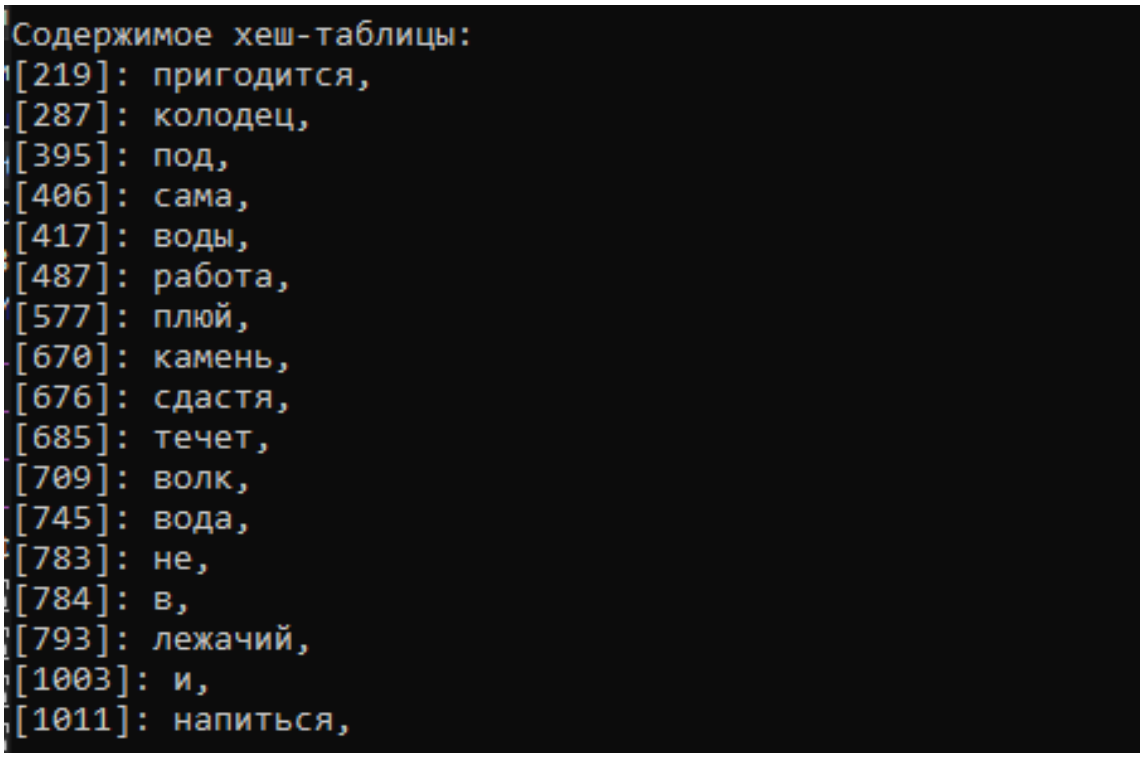

Рис. 1. Хэш-таблица

1.2 Хэш-функция

Хэш-функции — это функции, получающие на входе данные, обычно строку, и возвращающие число. При многократном вызове хэш-функции с одинаковыми входными данными она всегда будет возвращать одно и то же число, и возвращаемое число всегда будет находиться в гарантированном интервале.

Случай, при котором хеш-функция преобразует более чем одни входные данные (один массив входных данных) в одинаковые выходные данные (сводки), называется «коллизией». Вероятность возникновения коллизий используется для оценки качества хеш-функций.

В данной лабораторной работе использовалась хэш-функция FNV-1a.

FNV (Fowler–Noll–Vo) — это семейство хэш-функций, широко используемых для быстрого вычисления хэш-кодов строк и других данных. FNV-1 и FNV-1a — это две основные версии этого алгоритма. Они применяются для создания хэш-таблиц, проверки целостности данных и в других ситуациях, где требуется быстрый и простой способ создания уникальных идентификаторов для данных.

Основные характеристики FNV:

FNV-1: Хэш вычисляется путем последовательного умножения каждого байта данных на константу, а затем выполнения операции побитового XOR с текущим значением хэша.

FNV-1a: Порядок операций изменен: сначала выполняется операция XOR с текущим значением хэша, а затем умножение на константу.

Формула:

hash = offset_basis
for each byte_of_data to be hashed
hash = hash XOR byte_of_data
hash = hash * FNV_prime

Константы 32-bit:

Offset basis: 2166136261
FNV prime: 16777619

Данные константы приняты для небольших объемов данных и случаев, когда количество уникальных значений относительно невелико, используется в системах с ограниченной памятью, где каждый байт на счету, применяется в системах, где небольшие размеры хэш-кодов достаточно для предотвращения большого числа коллизий.

Размера хеш-таблицы - 1021. Простое число, что уменьшает количество коллизий. Для 100-700 слов 1021 — идеален (мало коллизий, память не перегружена).

Пример хэширования слова с помощью хэш-функции.

Слово "предпринимать".

Буквы: п р е д п р и н и м а т ь

Коды байтов (UTF-8): D0 BF D1 80 D0 B5 D0 B4 D0 BF D1 80 D0 B8 D0 BD D0 B8 D0 BC D0 B0 D1 82 D0 BC

Десятичные значения: 208 191 209 128 208 181 208 180 208 191 209 128 208 184 208 189 208 184 208 188 208 176 209 130 208 188

hash XOR 208 = 2166136261 XOR 208 = 2166136053

hash * 16777619 = 2166136053 * 16777619 = 36351108395099207

Обрезка до 32 бит: 36351108395099207 AND 0xFFFFFFFF = 2482448839

hash XOR 191 → 2482448839 XOR 191 = 2482449016

hash * 16777619 → 2482449016 * 16777619 = ...

И так далее для всех 26 байт.

Итоговый хеш: hash % 1021 = 1018

1.3 Кодирование

Устойчивость к коллизиям.

Пример для моего текста на 170 слов:

Всего ячеек: 1021

Пустых ячеек: 871 (85.3085%)

Коллизий: 10

Коэффициент заполнения: 14.6915%

Коэффициент заполнения должен быть в диапазоне 10-70% - получилось.

Пустые ячейки 30-90% - получилось.

Коллизии <15 для 150 слов - получилось.

Максимальная длина цепочки: 2 элемента (все коллизии - парные).

Нет кластеров (коллизии равномерно распределены по таблице).

Следовательно, использованная хэш-функция достаточно устойчива к коллизиям.

1.4 Красно-черное дерево

Красно-черное дерево (Red-black tree, RB-tree) - бинарное дерево, баланс которого достигается за счет поддержания раскраски вершин в два цвета, подчиняющейся следующим правилам:

1. Каждый узел покрашен либо в черный, либо в красный цвет.

2. Листьями объявляются NIL-узлами (т.е. "виртуальные"узлы, наследники узлов, которые обычно называют листьями; на них "указывают"NULL указатели). Листья покрашены в черный цвет.

3. Каждый красный узел имеет черного родителя (если узел красный,

то оба его потомка черны).

4. На всех ветвях дерева, ведущих от его корня к листьям, число черных узлов одинаково.

Все основные операции с красно-черным деревом можно реализовать за O(h), то есть $O(log_2 n)$.

Свойства красно-черных деревьев.

1. Корень и пустые узлы всегда имеют черный цвет.

2. Красная вершина не может иметь красных потомков.

3. Максимальные «черные» длины путей, ведущих из корня к листьям (пустым узлам), одинаковы.

Вставка.

1. Вставка как в бинарное дерево поиска (BST).

2. Новый узел всегда красный.

3. Балансировка для исправления нарушений:

- Если дядя красный, то перекрасить родителя, дядю и деда.

- Если дядя чёрный, то выполнить поворот (левый/правый).

Удаление.

1. Замена узла на потомка.

2. Если удалённый узел был чёрным, происходит балансировка - перекрашивание и повороты для устранения двойной чёрной вершины.

Левый поворот.

1. Правый потомок становится на место узла.

2. Левый потомок становится правым потомком.

Правый поворот выполняется аналогично, но зеркально (для левого потомка).

Замена поддерева - обновляет связи родителя и потомков.

Балансировка.

1. Перекрашивание узлов, чтобы свойства выполнялись.

2. Повороты, чтобы свойства выполнялись.

Сложность операций.

Вставка, удаление, поиск - O(log n). Повороты - O(1).

Пример моего красно-черного дерева (см. [Рис. 2]).

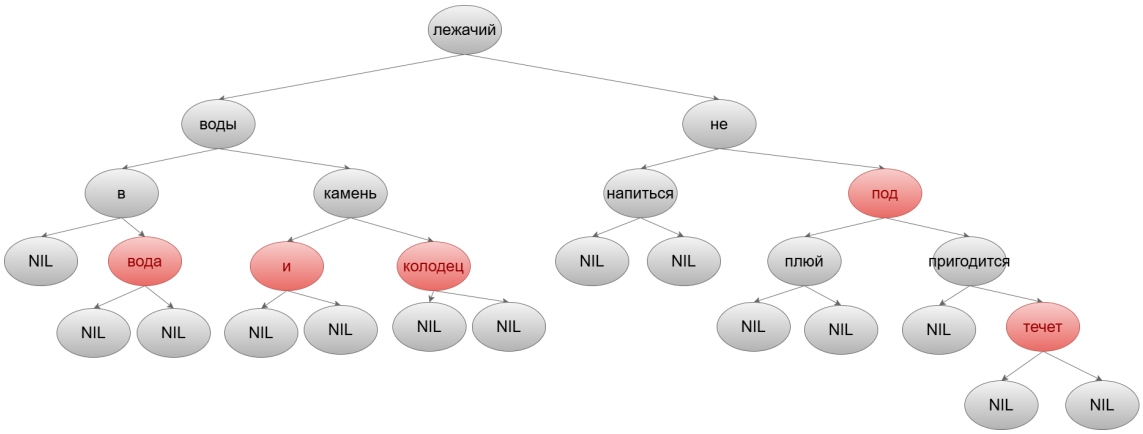

Рис. 2. Красно-черное дерево

1.5 Кодирование

Кодирование информации — отображение данных на кодовые слова.

Обычно в процессе кодирования информация преобразуется из формы, удобной для непосредственного использования, в форму, удобную для передачи, хранения или автоматической обработки. В более узком смысле кодированием информации называют представление информации в виде кода. Средством кодирования служит таблица соответствия знаковых систем, которая устанавливает взаимно однозначное соответствие между знаками или группами знаков двух различных знаковых систем.

Одноступенчатое и двуступенчатое кодирование – это методы преобразования информации в цифровой форме, используемые в системах передачи и обработки данных. Разница между ними заключается в количестве этапов кодирования.

1. Одноступенчатое кодирование.

Происходит за один этап: исходные данные сразу преобразуются в конечный закодированный формат.

2. Двуступенчатое кодирование.

Включает два этапа: сначала данные кодируются в промежуточный формат, затем преобразуются в окончательный.

В некоторых случаях добавление промежуточного этапа может ухудшить сжатие.

Я сжимаю случайно сгенерированный файл по распределению Парето. Одноступенчатое LZW-кодирование будет эффективнее двухступен-

чатого (RLE → LZW) по следующим причинам:

1. Символы распределены по закону Парето (степенное распределение с "тяжелыми хвостами"). Нет длинных повторяющихся последовательностей, так как генератор выдает псевдослучайные комбинации. А алгоритм RLE эффективен только при повторениях от 3–4 одинаковых символов подряд. RLE не сокращает, а увеличивает размер (каждый символ превращается в пару символ:1). Затем LZW вынужден сжимать уже искусственно раздутые данные.

2. RLE разбивает данные на одиночные символы + счетчики, уничтожая естественные повторения. LZW получает на вход менее избыточные данные и сжимает их хуже.

Двухступенчатое кодирование могло бы помочь, если были бы длинные повторяющиеся последовательности (больше 4х одинаковых символов подряд). Для псевдослучайных текстов это не работает.

1.6 Алгоритм LZW

Алгоритм Лемпеля — Зива — Велча (Lempel-Ziv-Welch, LZW) — это универсальный алгоритм сжатия данных. Этот метод позволяет достичь одну из наилучших степеней сжатия среди других существующих методов сжатия графических данных, при полном отсутствии потерь или искажений в исходных файлах.

Кодирование.

Шаг 1. Все возможные символы заносятся в словарь. Во входную фразу X заносится первый символ сообщения.

Шаг 2. Считать очередной символ Y из сообщения.

Шаг 3. Если Y — это символ конца сообщения, то выдать код для X , иначе:

Если фраза XY уже имеется в словаре, то присвоить входной фразе значение XY и перейти к Шагу 2 ,

Иначе выдать код для входной фразы X , добавить XY в словарь и присвоить входной фразе значение Y . Перейти к Шагу 2.

Конец.

Декодирование.

Шаг 1. Все возможные символы заносятся в словарь. Во входную

фразу X заносится первый код декодируемого сообщения.

Шаг 2. Считать очередной код Y из сообщения.

Шаг 3. Если Y — это конец сообщения, то выдать символ, соответствующий коду X , иначе:

Если фразы под кодом XY нет в словаре, вывести фразу, соответствующую коду X , а фразу с кодом XY занести в словарь.

Иначе присвоить входной фразе код XY и перейти к Шагу 2 .

Конец.

LZW полагается на повторяющиеся последовательности.

Если данные случайны LZW может увеличить размер. Также плохо работает с небольшими файлами: из-за накладных расходов на словарь LZW может проигрывать даже простым методам (RLE, Huffman).

1.7 Алгоритм RLE

Алгоритм RLE (Run-Length Encoding) — алгоритм сжатия, заменяющий идущие подряд одинаковые символы парой (повторяющийся символ, количество повторений). Например, строчку aaababbcbbb он переводит в (a, 3) (b, 1) (a, 1) (b, 2) (c, 1) (b, 3). Этот алгоритм эффективен для строк, содержащих много цепочек повторяющихся символов, например, результата преобразования Барроуза — Уилера.

2 Особенности реализации

2.1 Библиотечные классы

<iostream> – базовый ввод-вывод (стандартные потоки cin, cout).

<vector> – динамический массив для хранения данных.

<list> – двусвязный список для эффективных вставок/удалений.

<fstream> – работа с файлами (чтение/запись).

<string> – строковые операции.

<sstream> – строковые потоки.

<algorithm> – стандартные алгоритмы (сортировка, поиск и т. д.).

<set> – контейнер для хранения уникальных элементов.

<map> – ассоциативный массив (красно-черное дерево).

<cmath> – математические функции (логарифмы, степени и др.).

<random> – генерация случайных чисел (использовалось для тестовых данных).

<chrono> – замер времени выполнения.

<iomanip> – форматированный вывод.

2.2 Класс Hash

Класс реализует словарь на основе хеш-таблицы с цепочным методом разрешения коллизий. Также включает функции для сжатия и распаковки данных с использованием алгоритма LZW.

Поля.

vector<list<pair<int, string»> hash – хеш-таблица, где:

int – ключ (хэш слова),

string – само слово,

list – цепочка для коллизий.

2.2.1 Конструктор Hash

Вход: new_size (тип int) – начальный размер хеш-таблицы.

Выход: Инициализированная хеш-таблица типа vector<list<pair<int, string»>, где int – хеш-значение ключа, string – хранимое слово.

Конструктор класса Hash, который инициализирует хеш-таблицу заданного размера.

Листинг 1. Конструктор Hash

```
Hash::Hash(int new_size) {
    hash.resize(new_size);
}
```

2.2.2 Метод HashFunc

Вход: word (тип const string&) – строка, для которой необходимо вычислить хеш.

Выход: int – хеш-значение слова, приведенное к диапазону [0, 1020] (остаток от деления на 1021).

Вычисляет хеш-значение для переданного слова с использованием алгоритма FNV-1a (Fowler–Noll–Vo), который обеспечивает хорошее распределение значений и низкую вероятность коллизий. Полученный хеш используется для определения индекса в хеш-таблице.

Работа метода начинается с инициализации хеш-значения константой FNV_OFFSET_BASIS, которая для 32-битной версии алгоритма обычно равна 2166136261. Затем для каждого символа входной строки последовательно выполняется две операции: сначала применяется побитовое исключающее ИЛИ (XOR) между текущим значением хеша и ASCII-кодом символа, после чего результат умножается на простое число FNV_PRIME (обычно 16777619 для 32-битной версии).

После обработки всех символов полученное большое число приводится к размеру хеш-таблицы с помощью операции вычисления остатка от деления на 1021.

Листинг 2. Метод HashFunc

```
int Hash::HashFunc(const string& word) {
    uint32_t hash = FNV_OFFSET_BASIS;

    for (char c : word) {
        hash ^= static_cast<uint32_t>(static_cast<unsigned char
           >(c));
        hash *= FNV_PRIME;
    }
```

```
8
9       return hash % 1021;
10  }
```

2.2.3 Метод AddWord

Вход: word (string&) - слово для добавления (передается по ссылке)

Выход: хэш-таблица с новым словом, вывод сообщений на экран.

Добавляет новое слово в хеш-таблицу (словарь) с предварительной проверкой на уникальность.

Все символы слова приводятся к нижнему регистру (для регистронезависимого хранения), определяется индекс корзины с помощью функции HashFunc(). Производится поиск слова в соответствующей корзине. Если слово уже существует - выводится предупреждение и метод завершает работу без добавления.

Проверяется, что слово не пустое, слово добавляется в конец цепочки выбранной корзины и выводится подтверждение.

Листинг 3. Метод AddWord

```
1  void Hash::AddWord(string& word) {
2      for (char& c : word) c = tolower(c);
3      int ind = HashFunc(word);
4
5      for (auto& pairkv : hash[ind]) {
6          if (pairkv.second == word) {
7              icout << "Слово '" << word << "' уже есть в словаре\
                   n";
8              return;
9          }
10     }
11
12     if (!word.empty()) {
13         hash[ind].push_back({ 0, word });
14         cout << "Слово '" << word << "' успешно добавлено в слов
               арь\n";
15     }
16 }
```

2.2.4 Метод FindWord

Вход: word (const string&) - слово для поиска (передается по константной ссылке).

flagoutput (int) - флаг управления выводом (1- без вывода сообщений, 2 - с выводом результатов поиска).

Выход: если слово найдено - возвращает индекс в списке коллизий (больше или равно 0), если слово не найдено - возвращает -1.

Создается копия входного слова, все символы приводятся к нижнему регистру для регистронезависимого поиска. Вычисляется хеш слова для определения индекса корзины, инициализируется счетчик позиции в списке (-1). Последовательно перебираются элементы выбранной корзины, для каждого элемента увеличивается счетчик позиции.

При нахождении совпадения в режиме flagoutput=2 выводится информация о найденном слове, возвращается текущая позиция. Иначе, в режиме flagoutput=2 выводится сообщение об отсутствии. Возвращается -1.

Листинг 4. Метод FindWord

```
int Hash::FindWord(const string& word) {
    string w = word;
    for (char& c : w) c = tolower(c); // Приводим слово к нижнему регистру
    int ind = HashFunc(w); // Находим индекс в хеш-таблице
    int count = -1;

    for (auto& pairkv : hash[ind]) {
        count++; // Считаем индекс в списке элементов
        if (pairkv.second == w) cout << "Слово '" << w << "' есть в словаре, его индекс в хеш-таблице: " << ind << endl;
            return count; // Возвращаем индекс найденного слова в списке
        }
    }
        cout << "Слово '" << w << "' отсутствует в словаре" << endl;
    return -1; // Если слово не найдено
}
```

2.2.5 Метод DeleteWord

Вход: word (const string&) - слово для удаления (передается по константной ссылке).

Выход: хэш-таблица без этого слова, вывод в консоль.

Удаляет указанное слово из хеш-таблицы, если оно существует. Реализует безопасное удаление с проверкой наличия слова и информированием о результате операции.

Метод Hash::DeleteWord выполняет удаление слова из хеш-таблицы через несколько этапов. Сначала слово приводится к нижнему регистру для регистронезависимого поиска. Затем вычисляется хеш слова для определения нужной ячейки таблицы.

Метод FindWord в тихом режиме проверяет наличие слова. Если слово найдено, создается итератор, который перемещается к нужной позиции в цепочке коллизий, после чего элемент удаляется. В консоль выводится сообщение об успешном удалении. Если слово не найдено, выводится соответствующее уведомление.

Листинг 5. Метод DeleteWord

```
void Hash::DeleteWord(const string& word) {
    string w = word;
    for (char& c : w) c = tolower(c);
    int ind = HashFunc(w);
    int pair_index = FindWord(w, 1);
    if (pair_index != -1) {
        auto iter = hash[ind].begin();
        advance(iter, pair_index);
        hash[ind].erase(iter);
        cout << "Слово '" << w << "' удалено из словаря\n";
    }
    else cout << "Слово '" << w << "' отсутствует в словаре\n";
}
```

2.2.6 Метод DeleteAll

Вход: хэш-таблица.

Выход: пустая хэш-таблица, вывод в консоль.

Полностью очищает хеш-таблицу (словарь), удаляя все элементы из всех корзин. Перед очисткой проверяет, есть ли вообще данные для удаления, и информирует пользователя о результате операции.

Метод начинает работу с проверки текущего состояния хеш-таблицы. Последовательно перебирая все корзины (цепочки коллизий), он проверяет их на наличие элементов. Если обнаруживается хотя бы одна непустая корзина, метод фиксирует факт непустоты словаря.

При обнаружении данных в таблице выполняется полная очистка: для каждой корзины вызывается метод clear(), который удаляет все элементы из цепочек коллизий. По завершении очистки выводится соответствующее уведомление. Если же на этапе проверки выясняется, что таблица уже пуста, метод просто информирует пользователя об этом, избегая лишних операций.

Листинг 6. Метод DeleteAll

```
void Hash::DeleteAll() {
    bool empty = true;
    for (const auto& bucket : hash) {
        if (!bucket.empty()) {
            empty = false;
            break;
        }
    }
    if (!empty) {
        for (auto& bucket : hash) bucket.clear();
        cout << "Словарь очищен\n";
    }
    else cout << "Словарь пуст\n";
}
```

2.2.7 Метод PrintHash

Вход: хэш-таблица.

Выход: вывод на консоль всех непустых ячеек таблицы.

Выводит в консоль полное содержимое хеш-таблицы и подробную статистику о ее состоянии, включая данные о коллизиях и заполнении.

На первом этапе метод осуществляет обход всех ячеек хеш-таблицы, выводя в консоль только непустые ячейки с их содержимым. Для каждой занятой ячейки выводится её индекс в квадратных скобках, после чего через запятую перечисляются все хранящиеся в этой ячейке слова. Пустые ячейки при этом пропускаются для компактности вывода. После отображения содержимого таблицы метод переходит к сбору и анализу статистических данных. Второй проход по таблице включает подсчёт трёх ключевых показателей: количества пустых ячеек, общего

числа коллизий (определяемых как элементы сверх первого в каждой ячейке) и коэффициента заполнения таблицы. Для каждого показателя вычисляются как абсолютные значения, так и процентные соотношения. В результате формируется развёрнутая статистическая сводка, включающая общее количество ячеек, число и процент пустых ячеек, количество коллизий и процент заполнения таблицы.

Листинг 7. Метод PrintHash

```
void Hash::PrintHash() {
    cout << "\nСодержимое хеш-таблицы:\n";
    for (size_t i = 0; i < hash.size(); i++) {
        if (!hash[i].empty()) {
            cout << "[" << i << "]: ";
            for (const auto& pair : hash[i]) {
                cout << pair.second << ", ";
            }
            cout << endl;
        }
    }

     //Статистика
    int empty = 0, collisions = 0;
    for (const auto& bucket : hash) {
        if (bucket.empty()) empty++;
        else if (bucket.size() > 1) collisions += bucket.size()
           - 1;
    }

    cout << "\nСтатистика:\n";
    cout << "Всего ячеек: " << hash.size() << endl;
    cout << "Пустых ячеек: " << empty << " ("
        << (empty * 100.0 / hash.size()) << "%)" << endl;
    cout << "Коллизий: " << collisions << endl;
    cout << "Коэффициент заполнения: "
        << ((hash.size() - empty) * 100.0 / hash.size()) << "%\n
           ";
}
```

2.2.8 Метод ReadFromFile

Вход: строка с именем файла, который необходимо прочитать.

Выход: заплненная хэш-таблица.

Загружает слова из текстового файла в хеш-таблицу, выполняя предварительную обработку и валидацию данных.

Процесс работы начинается с попытки открытия указанного файла для чтения. Если файл не существует или не может быть открыт, метод немедленно выводит сообщение об ошибке и завершает работу. При успешном открытии файла метод последовательно считывает его содержимое построчно. Каждая строка разбивается на отдельные слова с помощью строкового потока. Для каждого полученного слова выполняется многоэтапная обработка: сначала удаляются все знаки пунктуации по краям слова, затем все символы приводятся к нижнему регистру для обеспечения регистронезависимости. После нормализации слова проверяется его соответствие заданным критериям валидности.

Корректные слова автоматически добавляются в хеш-таблицу с помощью внутреннего метода AddWord, в то время как о невалидных словах выводится предупреждающее сообщение, но они пропускаются. Процесс продолжается до полного чтения всего файла, после чего файловый дескриптор закрывается. Важно отметить, что метод обрабатывает слова в том порядке, в котором они встречаются в файле, но их конечное положение в хеш-таблице определяется хеш-функцией.

Листинг 8. Метод ReadFromFile

```
void Hash::ReadFromFile(const string& file_name) {
    ifstream in(file_name);
    string str;

    if (in.is_open()) {
        while (getline(in, str)) {
            istringstream iss(str);
            string word;
            while (iss >> word) {
                word.erase(remove_if(word.begin(), word.end(),
                   ispunctuation), word.end());
                for (char& c : word) c = tolower(c);

                if (correctWordInput(word)) {
                    AddWord(word);
                }
                else {
                    cout << "Слово '" << word << "' из файла не
                       было добавлено в словарь.\n";
                }
            }
        }
    }
    else {
        cout << "Такой файл отсутствует в директории\n";
    }
```

```
25      in.close();
26  }
```

2.2.9 Метод ExportToFile

Вход: filename — строка, содержащая путь к файлу, в который необходимо экспортировать данные из хеш-таблицы.

Выход: заполненный файл, вывод в консоль.

Метод предназначен для сохранения содержимого хеш-таблицы в текстовый файл. На первом этапе создаётся поток записи в файл с именем, переданным через параметр filename. Если файл не удаётся открыть, выполнение прерывается с выводом сообщения об ошибке.

Далее метод переходит к обходу всей хеш-таблицы, реализованной в виде массива списков (цепочек). Для каждой цепочки (бакета) последовательно просматриваются все элементы — пары, содержащие ключ и соответствующее значение. Из каждой пары извлекается только значение (в данном контексте — слово) и записывается в файл, каждая запись с новой строки. Ключи при этом не сохраняются.

По завершении обхода всех бакетов файл закрывается, и в стандартный поток вывода отправляется сообщение о завершении экспорта.

Листинг 9. Метод ExportToFile

```
1  void Hash::ExportToFile(const string& filename) const {
2      ofstream out(filename);
3      if (!out.is_open()) {
4          cerr << "Не удалось открыть файл для записи: " <<
               filename << endl;
5          return;
6      }
7
8      // Проходим по всем ячейкам хеш-таблицы
9      for (const auto& bucket : hash) {
10         // Проходим по всем словам в текущей ячейке
11         for (const auto& pair : bucket) {
12             out << pair.second << "\n"; // Записываем только сло
                   во
13         }
14     }
15
16     out.close();
17     cout << "Словарь успешно экспортирован в файл " << filename
         << endl;
```

```
18 }
```

2.2.10 Метод CompressDictionary

Вход: inputFile — строка с путем к исходному текстовому файлу, содержащему словарь.

outputFile — строка с путем к файлу, в который будет сохранен сжатый результат.

Выход: сжатый словарь, вывод в консоль статистики.

Метод выполняет сжатие словаря, содержащегося в текстовом файле, с использованием алгоритма LZW.

Сначала производится попытка открыть входной файл inputFile. Если файл открыть не удалось, выводится сообщение об ошибке, и выполнение метода прекращается. После успешного открытия содержимое файла целиком считывается в строку с помощью итераторов потока. Затем эта строка передаётся в функцию LZWEncode, реализующую алгоритм сжатия. Результатом работы этой функции является вектор целых чисел, представляющий сжатые данные. Далее вызывается функция WriteCompressed, которая записывает полученные закодированные данные в указанный выходной файл outputFile. После завершения сжатия метод открывает оба файла — исходный и сжатый — в бинарном режиме и позиционируется в конец файлов, чтобы определить их фактические размеры в байтах. Размеры исходного и сжатого файлов выводятся в консоль, а также рассчитывается и отображается коэффициент сжатия (отношение исходного размера к сжатому).

Листинг 10. Метод CompressDictionary

```
1  void Hash::CompressDictionary(const string& inputFile, const
       string& outputFile) {
2      ifstream in(inputFile);
3      if (!in) {
4          cerr << "Ошибка открытия файла " << inputFile << endl;
5          return;
6      }
7      string content((istreambuf_iterator<char>(in)),
           istreambuf_iterator<char>());
8      in.close();
9      vector<int> encoded = LZWEncode(content);
10     WriteCompressed(encoded, outputFile);
11     ifstream orig(inputFile, ios::ate | ios::binary);
```

```
12      ifstream comp(outputFile, ios::ate | ios::binary);
13      size_t originalSize = orig.tellg();
14      size_t compressedSize = comp.tellg();
15      orig.close();
16      comp.close();
17      cout << "Размеры:\n";
18      cout << "Исходный: " << originalSize << " байт\n";
19      cout << "Сжатый: " << compressedSize << " байт\n";
20      cout << "Коэффициент: " << (double)originalSize /
           compressedSize << "\n";
21  }
```

2.2.11 Метод DecompressDictionary

Вход: inputFile — строка, содержащая путь к файлу, в котором находятся сжатые данные (в бинарном формате).

outputFile — строка, содержащая путь к текстовому файлу, в который будет сохранён восстановленный словарь.

Выход: декодированный словарь, вывод в консоль.

Метод выполняет полное восстановление словаря из файла сжатых данных, закодированных с помощью алгоритма LZW. На первом этапе файл inputFile открывается в бинарном режиме. Затем из файла считывается первый байт, обозначающий количество бит на код. Оставшиеся байты считываются в массив bytes, после чего выполняется распаковка битов в целочисленные коды. Для этого используется буфер, в который побайтово записываются данные, с последующим извлечением кодов фиксированной длины. После распаковки выполняется декодирование по алгоритму LZW. Инициализируется словарь на 256 символов (ASCII), и последовательно обрабатываются коды. На каждом шаге определяется строка по текущему коду, формируется результат, и при необходимости в словарь добавляется новая комбинация. Важно, что словарь ограничен по размеру (65536 элементов), что предотвращает его переполнение.

После завершения декодирования результат сохраняется в выходной файл outputFile. Затем декодированный словарь загружается в новый временный объект Hash, из которого извлекается и выводится количество восстановленных слов, подтверждая успешность операции

Листинг 11. Метод DecompressDictionary

```
1  void Hash::DecompressDictionary(const string& inputFile, const
       string& outputFile) {
```

```
    ifstream in(inputFile, ios::binary);
    vector<uint8_t> bytes(
        (istreambuf_iterator<char>(in)),
        istreambuf_iterator<char>()
    );
    in.close();
    vector<int> encoded;
    uint32_t buffer = 0;
    int bits_in_buffer = 0;
    size_t byte_pos = 0;
    while (byte_pos < bytes.size() || bits_in_buffer >=
       bits_per_code) {
        while (bits_in_buffer < bits_per_code && byte_pos <
           bytes.size()) {
            buffer |= (uint32_t(bytes[byte_pos++]) <<
               bits_in_buffer);
            bits_in_buffer += 8;
        }
        if (bits_in_buffer >= bits_per_code) {
            int code = buffer & ((1 << bits_per_code) - 1);
            encoded.push_back(code);
            buffer >>= bits_per_code;
            bits_in_buffer -= bits_per_code;
        }
    }
    unordered_map<int, string> dictionary;
    for (int i = 0; i < 256; i++) {
        dictionary[i] = string(1, static_cast<char>(i));
    }
    string decoded;
    int dictSize = 256;

    string current = dictionary[encoded[0]];
    decoded = current;

    for (size_t i = 1; i < encoded.size(); i++) {
        int code = encoded[i];
        string entry;

        if (code == dictSize) {
            entry = current + current[0];
        }
        else if (code < dictSize) {
            entry = dictionary[code];
        }

        decoded += entry;
        if (dictSize < 65536) {
```

```
47                dictionary[dictSize++] = current + entry[0];
48            }
49            current = entry;
50        }
51
52        out << decoded;
53        out.close();
54        Hash tempDict;
55        tempDict.ReadFromFile(outputFile);
56        cout << "Словарь успешно восстановлен. Слов: "
57            << tempDict.GetWordCount() << endl;
58    }
```

2.2.12 Метод LZWEncode

Вход: input — строка, представляющая текст, который необходимо сжать.

Выход: Вектор целых чисел (vector<int>), содержащий закодированную последовательность по алгоритму LZW.

Метод реализует алгоритм сжатия данных LZW (Lempel–Ziv–Welch), предназначенный для преобразования входной строки в последовательность целых чисел с сохранением информации, но с уменьшением размера. На первом этапе создается словарь, содержащий все возможные символы ASCII (0–255), каждому из которых сопоставляется уникальный код. Затем метод инициализирует строку current, в которую будет постепенно накапливаться текущая последовательность символов. Алгоритм проходит по каждому символу входной строки input. К текущей строке current добавляется символ c, образуя next. Если строка next уже есть в словаре, она становится новым значением current. Если строки next в словаре нет, то в выходной вектор добавляется код строки current, а затем в словарь добавляется next с новым кодом. При этом ведется контроль размера словаря: если он превышает максимально допустимое значение (4096, т.е. 12-битные коды), словарь полностью сбрасывается и инициализируется заново.

По завершении цикла, если в current осталась непустая строка, она также добавляется в выходной поток кодов. Полученный вектор целых чисел возвращается как результат сжатия.

Листинг 12. Метод LZWEncode

```
1 vector<int> Hash::LZWEncode(const string& input) {
2     unordered_map<string, int> dictionary;
```

```
    // Инициализация словаря ASCII-символами (0-255)
    for (int i = 0; i < 256; i++) {
        dictionary[string(1, (char)i)] = i;
    }

    string current;
    vector<int> encoded;
    int dictSize = 256;
    const int max_dict_size = 4096; // Максимальный размер словаря (12 бит)

    for (char c : input) {
        string next = current + c;
        if (dictionary.count(next)) {
            current = next;
        }
        else {
            encoded.push_back(dictionary[current]);
            // Сброс словаря при переполнении
            if (dictSize >= max_dict_size) {
                dictionary.clear();
                for (int i = 0; i < 256; i++) {
                    dictionary[string(1, (char)i)] = i;
                }
                dictSize = 256;
            }

            dictionary[next] = dictSize++;
            current = string(1, c);
        }
    }

    if (!current.empty()) encoded.push_back(dictionary[current]);
    return encoded;
}
```

2.2.13 Метод ReadCompressed

Вход: filename — строка с именем бинарного файла, содержащего ранее сжатые данные (в формате LZW).

Выход: Вектор целых чисел (vector<int>), представляющий декодированную (извлечённую из потока битов) последовательность кодов.

Метод открывает бинарный файл, содержащий данные, сжатые с по-

мощью алгоритма LZW. Сначала читается первый байт файла, в котором закодировано количество бит на каждый код (bits_per_code), обычно от 8 до 16. После этого оставшиеся байты читаются в буфер. Основная часть метода — это извлечение целых кодов из последовательности битов. Для этого используется битовый буфер (buffer) и счётчик количества битов в нём (bits_in_buffer). По мере накопления достаточного количества битов извлекается очередной код с помощью побитовых операций. Этот код добавляется в результирующий вектор. Цикл продолжается до тех пор, пока не будут прочитаны все данные из файла. По завершении метод возвращает полученный вектор кодов для последующей декомпрессии.

Листинг 13. Метод ReadCompressed

```
vector<int> Hash::ReadCompressed(const string& filename) {
    ifstream in(filename, ios::binary);
    vector<int> codes;
    if (!in) {
        cerr << "Ошибка открытия файла" << endl;
        return codes;
    }
    int bits_per_code = in.get();
    vector<char> bytes(
        (istreambuf_iterator<char>(in)),
        istreambuf_iterator<char>()
    );

    uint32_t buffer = 0;
    int bits_in_buffer = 0;
    size_t byte_pos = 0;

    while (byte_pos < bytes.size() || bits_in_buffer >=
       bits_per_code) {
        while (bits_in_buffer < bits_per_code && byte_pos <
           bytes.size()) {
            buffer |= (uint32_t(bytes[byte_pos++]) <<
               bits_in_buffer);
            bits_in_buffer += 8;
        }

        if (bits_in_buffer >= bits_per_code) {
            int code = buffer & ((1 << bits_per_code) - 1);
            codes.push_back(code);
            buffer >>= bits_per_code;
            bits_in_buffer -= bits_per_code;
        }
    }

```

```
32      return codes;
33  }
```

2.2.14 Метод WriteCompressed

Вход: codes — вектор целых чисел (vector<int>), представляющий LZW-коды для записи в файл.

filename — строка с именем выходного файла, в который будет записан результат в бинарном виде.

Выход: файл с последовательностью битов.

Метод выполняет упаковку последовательности целых чисел в компактный бинарный формат с использованием битовой арифметики. Это часть алгоритма сжатия LZW. Он сначала определяет минимальное количество битов, необходимое для представления каждого кода из входного вектора — от 9 до 16 в зависимости от максимального значения в codes.

В начало выходного файла записывается 1 байт, содержащий bits_per_code — битность всех последующих кодов. Далее каждый код из codes поочерёдно добавляется в 32-битный буфер, пока в нём не наберётся как минимум 8 бит. Тогда младшие 8 бит буфера записываются в файл. Это повторяется, пока не будут обработаны все коды.

Если после завершения цикла в буфере остаются невыгруженные биты, они также дописываются в файл как последний байт. Метод завершает работу, закрывая поток.

Листинг 14. Метод WriteCompressed

```
1  void Hash::WriteCompressed(const vector<int>& codes, const
       string& filename) {
2      ofstream out(filename, ios::binary);
3      if (!out) {
4          cerr << "Ошибка открытия файла для записи" << endl;
5          return;
6      }
7      // Определяем необходимое количество бит на код
8      int max_code = *max_element(codes.begin(), codes.end());
9      int bits_per_code = 9; // Начинаем с 9 бит
10     if (max_code >= 512) bits_per_code = 10;
11     if (max_code >= 1024) bits_per_code = 12;
12     if (max_code >= 4096) bits_per_code = 16;
13
14     out.put(static_cast<char>(bits_per_code));
```

```
15      uint32_t buffer = 0;
16      int bits_in_buffer = 0;
17
18      for (int code : codes) {
19          buffer |= (code << bits_in_buffer);
20          bits_in_buffer += bits_per_code;
21
22          while (bits_in_buffer >= 8) {
23              out.put(buffer & 0xFF);
24              buffer >>= 8;
25              bits_in_buffer -= 8;
26          }
27      }
28
29      if (bits_in_buffer > 0) out.put(buffer & 0xFF);
30
31      out.close();
32  }
```

2.2.15 Метод GetWordCount

Вход: хэш-таблца.

Выход: Целое число — общее количество слов, хранящихся в хеш-таблице.

Метод подсчитывает общее количество элементов, находящихся во всех ячейках хеш-таблицы. В структуре таблицы каждая ячейка (bucket) представляет собой, например, список пар "ключ–значение"(слов). Метод проходит по всем ячейкам и суммирует количество слов в каждой из них с помощью bucket.size().

Листинг 15. Метод GetWordCount

```
1   int Hash::GetWordCount() const {
2       int count = 0;
3       for (const auto& bucket : hash) {
4           count += bucket.size();
5       }
6       return count;
7   }
```

2.3 Класс RBTree

Класс реализует словарь на основе красно-черного дерева.

Поля.

Node* root – указатель на корень дерева.

Node* nil – специальный лист-заглушка (черный узел, заменяющий nullptr для упрощения балансировки).

struct Node – структура узла дерева, содержащая:

string word – хранимое слово,

Color color – цвет узла (RED или BLACK),

Node* left, Node* right, Node* parent – указатели на левого и правого потомков, а также на родителя.

2.3.1 Конструктор RBTree

Вход: пустое красно-черное дерево.

Выход: инициализированное красно-черное дерево.

Конструктор класса RBTree инициализирует пустое красно-черное дерево. В процессе создания экземпляра класса создается узел nil, который инициализируется значением BLACK и представляет собой специальный маркер для пустых листьев дерева. После этого корень дерева (root) устанавливается на nil, что указывает на то, что дерево пусто.

Листинг 16. Конструктор RBTree

```
RBTree::RBTree() {
nil = new Node("", BLACK);
root = nil;
}
```

2.3.2 Деструктор RBTree

Вход: красно-черное дерево.

Выход: пустое красно-черное дерево.

Деструктор класса RBTree выполняет очистку ресурсов, занимаемых объектом дерева. Он вызывает метод clearTree, который рекурсивно освобождает память, занятую всеми узлами дерева, начиная с корня

(root). После этого удаляется узел nil, который был создан в конструкторе. Это гарантирует освобождение всех выделенных динамических объектов и предотвращает утечки памяти.

Листинг 17. Деструктор RBTree

```
RBTree::~RBTree() {
clearTree(root);
delete nil;
}
```

2.3.3 Метод insert

Вход: const std::string& word — слово, которое необходимо вставить в дерево.

Выход: красно-черное дерево с новым словом.

Метод вставляет новое слово в красно-черное дерево (RBTree). Перед вставкой слово переводится в нижний регистр для обеспечения регистрационной независимости. Далее создается новый узел, и с помощью итеративного поиска находится подходящее место для вставки:

Если слово уже существует в дереве, операция прерывается, узел удаляется, и в консоль выводится соответствующее сообщение.

Если место найдено, узел добавляется как левый или правый потомок найденного родителя.

После вставки происходит корректировка дерева с помощью метода insertFixup, чтобы сохранить свойства красно-черного дерева (балансировка и цветовые правила).

Листинг 18. Метод insert

```
void RBTree::insert(const std::string& word) {
std::string w = word;
for (char& c : w) c = tolower(c);
Node* z = new Node(w);
Node* y = nil;
Node* x = root;

while (x != nil) {
    y = x;
    if (z->word < x->word) {
        x = x->left;
    }
    else if (z->word > x->word) {
```

```
14          x = x->right;
15      }
16      else {
17          delete z;
18          std::cout << "Слово '" << w << "' уже есть в словаре\n";
19          return;
20      }
21  }
22
23  z->parent = y;
24  if (y == nil) {
25      root = z;
26  }
27  else if (z->word < y->word) {
28      y->left = z;
29  }
30  else {
31      y->right = z;
32  }
33
34  z->left = nil;
35  z->right = nil;
36  z->color = RED;
37
38  insertFixup(z);
39  std::cout << "Слово '" << w << "' успешно добавлено в словарь\n"
        ;
40  }
```

2.3.4 Метод remove

Вход: const std::string& word — слово, которое необходимо удалить из дерева.

Выход: модифицированное дерево.

Метод удаляет заданное слово из красно-черного дерева (RBTree). Перед началом слово переводится в нижний регистр для унификации. Затем дерево просматривается с помощью метода searchNode, чтобы найти узел с нужным словом. Если такой узел отсутствует, выводится соответствующее сообщение.

При удалении обрабатываются три случая:

У узла нет левого поддерева: производится пересадка правого поддерева.

У узла нет правого поддерева: производится пересадка левого поддерева.

У узла есть оба поддерева: находится минимальный узел в правом поддереве (преемник), происходит его пересадка на место удаляемого.

После удаления производится корректировка структуры с помощью deleteFixup, если удаленный узел был черным, чтобы сохранить свойства красно-черного дерева.

Листинг 19. Метод remove

```
void RBTree::remove(const std::string& word) {
std::string w = word;
for (char& c : w) c = tolower(c);
Node* z = searchNode(w);
if (z == nil) {
    std::cout << "Слово '" << w << "' отсутствует в словаре\n";
    return;
}

Node* y = z;
Node* x;
Color y_original_color = y->color;

if (z->left == nil) {
    x = z->right;
    transplant(z, z->right);
}
else if (z->right == nil) {
    x = z->left;
    transplant(z, z->left);
}
else {
    y = minimum(z->right);
    y_original_color = y->color;
    x = y->right;

    if (y->parent == z) {
        x->parent = y;
    }
    else {
        transplant(y, y->right);
        y->right = z->right;
        y->right->parent = y;
    }

    transplant(z, y);
    y->left = z->left;
```

```
38      y->left->parent = y;
39      y->color = z->color;
40  }
41
42  if (y_original_color == BLACK) {
43      deleteFixup(x);
44  }
45
46  delete z;
47  std::cout << "Слово '" << w << "' удалено из словаря\n";
48  }
```

2.3.5 Метод search

Вход: const std::string& word — искомое слово.

Выход: true, если слово найдено в дереве, иначе false.

Метод ищет слово в красно-черном дереве. Перед поиском оно преобразуется в нижний регистр. Поиск осуществляется с помощью вспомогательной функции searchNode.

Если слово найдено, в консоль выводится информация о нем:

1) само слово;

2) цвет узла (R — красный, B — черный);

3) имя родительского узла (или NIL, если он отсутствует);

4) имена левого и правого потомков (или NIL, если их нет).

Если слово не найдено, выводится сообщение об его отсутствии. Метод возвращает true или false в зависимости от результата поиска.

Листинг 20. Метод search

```
1  bool RBTree::search(const std::string& word) {
2  std::string w = word;
3  for (char& c : w) c = tolower(c);
4  Node* result = searchNode(w);
5  if (result != nil) {
6      std::cout << result->word << " (" << (result->color == RED ?
           "R" : "B") << ")";
7
8      if (result->parent != nil) {
9          std::cout << " [parent: " << result->parent->word << "]"
               ;
10     }
11     else {
```

```
12          std::cout << " [parent: NIL]";
13      }
14
15      std::cout << " {";
16      if (result->left != nil) {
17          std::cout << "L: " << result->left->word;
18      }
19      else {
20          std::cout << "L: NIL";
21      }
22      std::cout << ", ";
23      if (result->right != nil) {
24          std::cout << "R: " << result->right->word;
25      }
26      else {
27          std::cout << "R: NIL";
28      }
29      std::cout << "}";
30
31      std::cout << std::endl;
32      return true;
33  }
34  else {
35      std::cout << "Слово '" << w << "' отсутствует в словаре\n";
36      return false;
37  }
38  }
```

2.3.6 Метод clear

Вход: красно-черное дерево.

Выход: пустое дерево

Если корень дерева указывает на nil, значит дерево уже пусто — в этом случае выводится соответствующее сообщение. Если дерево не пусто, вызывается вспомогательная рекурсивная функция clearTree, которая удаляет все узлы начиная с корня. После завершения удаления корень устанавливается в nil, и в консоль выводится сообщение об успешной очистке.

Листинг 21. Метод clear

```
1  void RBTree::clear() {
2  if (root == nil) {
3  std::cout << "Словарь уже пуст\n";
4  return;
```

```
}
clearTree(root);
root = nil;
std::cout << "Словарь очищен\n";
}
```

2.3.7 Метод print

Вход: красно-черное дерево.

Выход: визуализация дерева в консоли.

Процесс работы программы: Метод используется для отображения всех элементов красно-черного дерева. Если корень дерева указывает на nil, то дерево считается пустым, и выводится сообщение «Словарь пуст». Иначе запускается рекурсивная функция printTree, которая печатает структуру дерева в удобочитаемом виде, отображая иерархию узлов с отступами. Используется символьное форматирование для визуализации глубины вложенности.

Листинг 22. Метод print

```
void RBTree::print() {
if (root == nil) {
std::cout << "Словарь пуст\n";
return;
}
std::cout << "\nСодержимое красно–черного дерева:\n";
printTree(root, 0);
}
```

2.3.8 Метод readFromFile

Вход: filename — строка с именем текстового файла, содержащего слова для добавления в словарь.

Выход: модифицированное дерево.

Процесс работы программы: Метод открывает указанный текстовый файл и построчно считывает его содержимое. Каждая строка разбивается на отдельные слова с помощью потока istringstream. Перед вставкой каждое слово:

1) очищается от знаков препинания,

2) приводится к нижнему регистру,

3) проверяется с помощью функции correctWordInput.

Если слово проходит проверку, оно вставляется в красно-чёрное дерево через метод insert. В конце выводится сообщение об успешной загрузке, либо сообщение об ошибке, если файл не был открыт.

Листинг 23. Метод readFromFile

```
void RBTree::readFromFile(const std::string& filename) {
std::ifstream in(filename);
std::string str;

if (in.is_open()) {
    while (std::getline(in, str)) {
        std::istringstream iss(str);
        std::string word;
        while (iss >> word) {
            word.erase(std::remove_if(word.begin(), word.end(),
               ispunctuation), word.end());
            for (char& c : word) c = tolower(c);

            if (correctWordInput(word)) {
                insert(word);
            }
        }
    }
    std::cout << "Слова из файла '" << filename << "' успешно до
       бавлены в словарь\n";
}
else {
    std::cout << "Файл '" << filename << "' не найден\n";
}
in.close();
}
```

2.3.9 Метод leftRotate

Вход: x — указатель на узел дерева, относительно которого производится левый поворот.

Выход: модифицированное поддерево

Метод выполняет левый поворот в красно-чёрном дереве относительно узла x. Это вспомогательная операция, используемая для балансировки дерева после вставки или удаления узлов.

Узел y устанавливается как правый потомок x. Правое поддерево x заменяется на левое поддерево y. Если левое поддерево y не nil, его родитель обновляется на x. Родитель y становится родителем x. Если x был корнем, y становится новым корнем. Иначе x заменяется на y как левый или правый потомок своего родителя. Левый потомок y становится x, и x обновляет своего родителя на y.

Листинг 24. Метод leftRotate

```
void RBTree::leftRotate(Node* x) {
Node* y = x->right;
x->right = y->left;
if (y->left != nil) {
    y->left->parent = x;
}

y->parent = x->parent;

if (x->parent == nil) {
    root = y;
}
else if (x == x->parent->left) {
    x->parent->left = y;
}
else {
    x->parent->right = y;
}

y->left = x;
x->parent = y;
}
```

2.3.10 Метод rightRotate

Вход: y — указатель на узел дерева, относительно которого производится правый поворот.

Выход: модифицированное поддерево.

Метод выполняет правый поворот красно-чёрного дерева вокруг узла y. Это вспомогательная операция, используемая при балансировке дерева во время вставки или удаления.

Узел x устанавливается как левый потомок y. Левое поддерево y заменяется на правое поддерево x. Если правое поддерево x не nil, его родитель обновляется на y. Родитель x становится родителем y. Если y

— корень дерева, x становится новым корнем. Иначе y заменяется на x как левый или правый потомок своего родителя. Правый потомок x становится y, и y обновляет своего родителя на x.

Листинг 25. Метод rightRotate

```
void RBTree::rightRotate(Node* y) {
Node* x = y->left;
y->left = x->right;

if (x->right != nil) {
    x->right->parent = y;
}

x->parent = y->parent;

if (y->parent == nil) {
    root = x;
}
else if (y == y->parent->right) {
    y->parent->right = x;
}
else {
    y->parent->left = x;
}

x->right = y;
y->parent = x;
}
```

2.3.11 Метод insertFixup

Вход: z — указатель на новый узел, только что вставленный в красно-чёрное дерево.

Выход: сбалансированное дерево.

Функция insertFixup исправляет возможные нарушения свойств красно-черного дерева после добавления нового узла z. Основной цикл выполняется, пока родитель узла z является красным (нарушение свойства, запрещающего два красных узла подряд). В теле цикла сначала определяется, является ли родитель z левым или правым потомком своего родителя (деда z). В зависимости от этого рассматриваются два симметричных случая.

Если родитель z - левый потомок.

1. Находим "дядю"у (правого потомка деда).

2. Если дядя красный (случай 1):

3. Перекрашиваем родителя и дядю в черный.

4. Деда делаем красным.

5. Перемещаем указатель z на деда для продолжения проверок выше по дереву.

Если дядя черный (случай 2):

1. Если z - правый потомок, сначала делаем левый поворот вокруг родителя.

2. Перекрашиваем родителя в черный, деда - в красный.

3. Делаем правый поворот вокруг деда.

Симметричная обработка выполняется, когда родитель z - правый потомок:

Находим "дядю"у (теперь левого потомка деда)

Если дядя красный - аналогично перекрашиваем

Если черный - выполняем правый поворот при необходимости, затем левый

После завершения цикла обязательно устанавливаем цвет корня в черный (на случай, если он был изменен на красный в процессе балансировки). Это гарантирует соблюдение первого свойства красно-черного дерева.

Листинг 26. insertFixup

```
void RBTree::insertFixup(Node* z) {
    while (z->parent->color == RED) {
        if (z->parent == z->parent->parent->left) {
            Node* y = z->parent->parent->right;

            if (y->color == RED) {
                z->parent->color = BLACK;
                y->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;
            }
            else {
                if (z == z->parent->right) {
                    z = z->parent;
                    leftRotate(z);
                }
```

```
17
18                  z->parent->color = BLACK;
19                  z->parent->parent->color = RED;
20                  rightRotate(z->parent->parent);
21              }
22          }
23          else {
24              Node* y = z->parent->parent->left;
25
26              if (y->color == RED) {
27                  z->parent->color = BLACK;
28                  y->color = BLACK;
29                  z->parent->parent->color = RED;
30                  z = z->parent->parent;
31              }
32              else {
33                  if (z == z->parent->left) {
34                      z = z->parent;
35                      rightRotate(z);
36                  }
37
38                  z->parent->color = BLACK;
39                  z->parent->parent->color = RED;
40                  leftRotate(z->parent->parent);
41              }
42          }
43      }
44      root->color = BLACK;
45  }
```

2.3.12 Метод deleteFixup

Вход: Node* x — узел, который требуется восстановить после удаления в красно-чёрном дереве.

Выход: модифицированное дерево.

Метод восстанавливает свойства красно-чёрного дерева после удаления узла. Если x — чёрный узел, то его удаление могло нарушить баланс чёрных высот. Алгоритм рассматривает два симметричных случая (x — левый или правый ребёнок) и выполняет перекрашивание и повороты, чтобы восстановить свойства дерева.

Листинг 27. Метод deleteFixup

```
1 void RBTree::deleteFixup(Node* x) {
2 while (x != root && x->color == BLACK) {
```

```
3  if (x == x->parent->left) {
4  Node* w = x->parent->right;
5
6          if (w->color == RED) {
7              w->color = BLACK;
8              x->parent->color = RED;
9              leftRotate(x->parent);
10             w = x->parent->right;
11         }
12
13         if (w->left->color == BLACK && w->right->color == BLACK)
               {
14             w->color = RED;
15             x = x->parent;
16         }
17         else {
18             if (w->right->color == BLACK) {
19                 w->left->color = BLACK;
20                 w->color = RED;
21                 rightRotate(w);
22                 w = x->parent->right;
23             }
24
25             w->color = x->parent->color;
26             x->parent->color = BLACK;
27             w->right->color = BLACK;
28             leftRotate(x->parent);
29             x = root;
30         }
31     }
32     else {
33         Node* w = x->parent->left;
34
35         if (w->color == RED) {
36             w->color = BLACK;
37             x->parent->color = RED;
38             rightRotate(x->parent);
39             w = x->parent->left;
40         }
41
42         if (w->right->color == BLACK && w->left->color == BLACK)
               {
43             w->color = RED;
44             x = x->parent;
45         }
46         else {
47             if (w->left->color == BLACK) {
48                 w->right->color = BLACK;
```

```
49                  w->color = RED;
50                  leftRotate(w);
51                  w = x->parent->left;
52              }
53
54              w->color = x->parent->color;
55              x->parent->color = BLACK;
56              w->left->color = BLACK;
57              rightRotate(x->parent);
58              x = root;
59          }
60      }
61  }
62  x->color = BLACK;
63  }
```

2.3.13 Метод transplant

Вход: Node* u — узел, который нужно заменить.

Node* v — узел, который будет поставлен вместо ‘u‘ .

Выход: модифицированное дерево (новые связи).

Метод ‘transplant‘ выполняет замену поддерева с корнем в узле ‘u‘ на поддерево с корнем в узле ‘v‘. Основные шаги:

1. Если ‘u‘ является корнем дерева (‘u->parent == nil‘), то ‘v‘ становится новым корнем.

2. Если ‘u‘ был левым потомком своего родителя, то ‘v‘ становится левым потомком. Иначе ‘v‘ становится правым потомком.

3. Устанавливаем родителя ‘v‘ равным родителю ‘u‘.

Листинг 28. Метод transplant

```
1  void RBTree::transplant(Node* u, Node* v) {
2      if (u->parent == nil) {
3          root = v;
4      }
5      else if (u == u->parent->left) {
6          u->parent->left = v;
7      }
8      else {
9          u->parent->right = v;
10     }
11     v->parent = u->parent;
12 }
```

2.3.14 Метод minimum

Вход: Node* node — узел, начиная с которого ищется минимальный элемент в поддереве.

Выход: Node* — указатель на узел с минимальным ключом в поддереве с корнем ‘node‘.

Метод находит узел с минимальным ключом в поддереве с корнем в ‘node‘, следуя левым потомкам до достижения листа (‘nil‘).

Пока существует левый потомок (‘node->left != nil‘), переходим к нему. Возвращаем последний найденный узел, который не имеет левого потомка (с минимальным ключом).

Листинг 29. Метод minimum

```
RBTree::Node* RBTree::minimum(Node* node) {
    while (node->left != nil) {
        node = node->left;
    }
    return node;
}
```

2.3.15 Метод printTree

Вход: Node* node — текущий узел для печати (обычно начинается с корня).

int depth — текущая глубина в дереве (для отступов).

Выход: вывод дерева на консоль.

Метод рекурсивно выводит структуру красно-чёрного дерева в удобном для чтения формате:

Сначала рекурсивно обрабатывается правое поддерево (чтобы дерево выводилось "на боку"с корнем слева). Глубина узла определяет количество отступов (4 пробела на уровень глубины)

Выводится:

1) Ключ узла (‘word‘) ;

2) Цвет (R - красный, B - черный) ;

3) Родительский узел (если есть);

4) Левый и правый потомки (или NIL если их нет).

Затем рекурсивно обрабатывается левое поддерево

Листинг 30. Метод printTree

```
void RBTree::printTree(Node* node, int depth) {
    if (node == nil) return;

    // Сначала правое поддерево (для лучшей визуализации)
    printTree(node->right, depth + 1);

    // Отступы в зависимости от глубины
    for (int i = 0; i < depth; ++i) {
        std::cout << "    ";
    }
    std::cout << node->word << " (" << (node->color == RED ? "R"
        : "B") << ")";
    if (node->parent != nil) {
        std::cout << " [parent: " << node->parent->word << "]";
    }
    std::cout << " {";
    if (node->left != nil) {
        std::cout << "L: " << node->left->word;
    }
    else {
        std::cout << "L: NIL";
    }
    std::cout << ", ";
    if (node->right != nil) {
        std::cout << "R: " << node->right->word;
    }
    else {
        std::cout << "R: NIL";
    }
    std::cout << "}";
    std::cout << std::endl;
    printTree(node->left, depth + 1);
}
```

2.3.16 Метод clearTree

Вход: Указатель на вершину поддерева node (обычно вызывается с корня дерева).

Выход: Пустое дерево.

Листинг 31. Метод clearTree

```
void RBTree::clearTree(Node* node) {
if (node != nil) {
```

```
3 clearTree(node->left); // Рекурсивно удаляем левое поддерево
4 clearTree(node->right); // Рекурсивно удаляем правое поддерево
5 delete node; // Удаляем текущий узел
6 }
7 }
```

2.3.17 Метод searchNode

Вход: ‘const std::string& word‘ — слово для поиска в красно-черном дереве.

Выход: Указатель на узел ‘Node*‘, содержащий заданное слово, если оно найдено. Если слово отсутствует — возвращается ‘nil‘.

Метод searchNode выполняет поиск слова в красно-черном дереве. Сначала переданное слово копируется во временную строку и приводится к нижнему регистру, чтобы обеспечить регистронезависимый поиск. Далее начинается проход по дереву от корня.

На каждом шаге:

1) если текущее слово в узле совпадает с искомым — метод возвращает указатель на этот узел;

2) если искомое слово меньше слова в текущем узле (лексикографически) — переход к левому поддереву;

3) если больше — переход к правому поддереву.

Поиск продолжается, пока не будет найдено совпадение или не достигнут специальный nil-узел, обозначающий конец ветви. Если слово не найдено, метод возвращает nil.

Листинг 32. Метод searchNode

```
1 RBTree::Node* RBTree::searchNode(const std::string& word) {
2 std::string w = word;
3 for (char& c : w) c = tolower(c); // Приведение слова к нижнему
      регистру
4 Node* current = root;
5 while (current != nil) {
6     if (w == current->word) {
7         return current;              // Совпадение найдено
8     }
9     else if (w < current->word) {
10        current = current->left;    // Идем в левое поддерево
11    }
12    else {
```

```
13            current = current->right;   // Идем в правое поддерево
14        }
15    }
16    return nil;                         // Слово не найдено
17
18    }
```

2.4 Функции

2.4.1 Функция correctWordInput

Вход: const string& word — строка, представляющая введённое пользователем слово.

Выход: true, если слово корректно (содержит только строчные русские буквы и дефисы). false, если в слове обнаружены цифры или недопустимые символы. В этом случае выводится сообщение об ошибке.

Метод проверяет, состоит ли строка только из строчных русских букв и символов дефиса. Для каждого символа строки:

1) если это цифра, или если символ не входит в диапазон русских строчных букв ('а'–'я') и не является дефисом, метод выводит сообщение об ошибке и возвращает false;

2) если все символы допустимы, метод возвращает true.

Листинг 33. Метод correctWordInput

```
1  bool correctWordInput(const string& word) {
2  for (int i = 0; i < word.length(); i++) {
3  char c = word[i];
4  bool digitflag = isdigit(word[i]); // Проверка, является ли симв
       ол цифрой
5  bool rusflag = (c >= 'а' and c <= 'я') or (c == '-'); // Проверк
       а, является ли символ русской буквой или дефисом
6  if (digitflag or !rusflag) {
7  cout << "Слово содержит цифру или другой неподходящий символ. По
       вторите ввод.\n";
8  return false; // Найден недопустимый символ
9  }
10 }
11 return true; // Все символы допустимы
12 }
```

2.4.2 Функция ispunctuation

Вход: char c — символ, который необходимо проверить.

Выход: true, если символ относится к перечню знаков препинания: . , "; : ! ? « » — false — если символ не входит в список.

Метод проверяет, является ли переданный символ одним из заранее заданных знаков препинания. Для этого он перебирает массив знаков и сравнивает каждый с символом c. Если совпадение найдено, возвращается true. Если перебор завершается без совпадения — возвращается false.

Листинг 34. Метод ispunctuation

```
1
2 bool ispunctuation(char c) {
3 char arr[] = { '.', ',', '"', ';', ':', '!', '?', '  ', '  ', '
      ' };
4 for (char punct : arr) {
5 if (c == punct) return true; // Совпадение найдено
6 }
7 return false; // Символ не найден среди знаков препинания
8 }
```

2.4.3 Функция readEncodedFromFile

Вход:

const string& filename — путь к бинарному файлу с закодированными данными

Выход:

vector<int> — вектор раскодированных значений

Метод читает бинарный файл и декодирует его содержимое. Файл открывается в бинарном режиме. При ошибке выводится сообщение и возвращается пустой вектор.

Первый байт файла содержит bits_per_code — количество бит на одно закодированное значение Остальные данные файла считываются в вектор байт.

Декодирование:

Используется буфер (buffer) для накопления битов. Постепенно из буфера извлекаются значения по bits_per_code бит. Каждое извлечен-

ное значение добавляется в результирующий вектор.

Листинг 35. Метод readEncodedFromFile

```
vector<int> readEncodedFromFile(const string& filename) {
ifstream file(filename, ios::binary);
vector<int> encoded;
if (!file) {
cerr << "Ошибка открытия файла!" << endl;
return encoded;
}

// Читаем bits_per_code (1 байт)
int bits_per_code = file.get();

// Читаем остальные данные
vector<char> bytes(
  (istreambuf_iterator<char>(file)),
  istreambuf_iterator<char>()
);

uint64_t buffer = 0;
int bits_in_buffer = 0;
size_t byte_pos = 0;

while (byte_pos < bytes.size() || bits_in_buffer >=
   bits_per_code) {
  // Наполняем буфер
  while (bits_in_buffer < bits_per_code && byte_pos < bytes.size
     ()) {
    buffer |= (uint64_t(bytes[byte_pos++]) << bits_in_buffer);
    bits_in_buffer += 8;
  }

  // Извлекаем код
  if (bits_in_buffer >= bits_per_code) {
    int code = buffer & ((1 << bits_per_code) - 1);
    encoded.push_back(code);
    buffer >>= bits_per_code;
    bits_in_buffer -= bits_per_code;
  }
}

return encoded;
}
```

2.4.4 Функция writeEncodedToFile

Вход: const string& filename — путь к файлу для записи.

const string& decoded — декодированные данные для сохранения.

Выход: файл с декодированными данными.

Метод сохраняет декодированные данные в текстовый файл, предварительно очищая его.

Создается временный поток clear_file с флагом ios::trunc. Это гарантирует, что файл будет пустым перед записью новых данных.

Файл открывается в текстовом режиме (по умолчанию). При ошибке выводится сообщение и метод завершается. Все содержимое строки decoded записывается в файл, и поток закрывается для гарантии сохранения данных.

Листинг 36. Метод saveDecodedToFile

```
void saveDecodedToFile(const string& filename, const string&
    decoded) {
// Очищаем файл перед записью
ofstream clear_file(filename, ios::trunc);
clear_file.close();

ofstream file(filename);
if (!file) {
    cerr << "Не удалось открыть файл для записи." << endl;
    return;
}
file << decoded;
file.close();
}
```

2.4.5 Функция LZWDecode

Вход: const vector<int>& encoded — вектор закодированных кодов LZW.

Выход: string — декодированная строка.

Метод реализует алгоритм декодирования LZW. Создается начальный словарь, содержащий все 256 символов ASCII, каждому символу соответствует его однобайтовое представление. Первый код из входного вектора декодируется через словарь. Результат добавляется в выходную строку.

Если код есть в словаре — берется соответствующая строка. Если код равен текущему размеру словаря — обрабатывается специальный случай LZW. Иначе генерируется ошибка.

Найденная строка добавляется к результату. В словарь добавляется новая комбинация: текущая строка + первый символ новой строки.

После обработки всех кодов возвращается полная декодированная строка.

Листинг 37. Метод LZWDecode

```
string LZWDecode(const vector<int>& encoded) {
// Инициализация словаря ASCII символами
unordered_map<int, string> dictionary;
for (int i = 0; i < 256; ++i) {
dictionary[i] = string(1, static_cast<char>(i));
}

string decoded;
int dictSize = 256;

if (encoded.empty()) return decoded;

// Обработка первого кода
string current = dictionary[encoded[0]];
decoded += current;

for (size_t i = 1; i < encoded.size(); ++i) {
    int code = encoded[i];
    string entry;

    if (dictionary.count(code)) {
        entry = dictionary[code];
    }
    else if (code == dictSize) {
        // Специальный случай LZW
        entry = current + current[0];
    }
    else {
        throw runtime_error("Invalid LZW code");
    }
    decoded += entry;
    // Добавление новой комбинации в словарь
    dictionary[dictSize++] = current + entry[0];
    current = entry;
}

return decoded;
```

```
}
```

2.4.6 Функция LZWEncode

Вход:

const string& input — строка для кодирования.

Выход:

vector<int> — вектор закодированных кодов LZW.

Метод реализует алгоритм кодирования LZW.

Создается начальный словарь, содержащий все 256 символов ASCII. Каждому символу соответствует его числовой код (0-255).

Для каждого символа входной строки:

1) Формируется строка next = current + текущий символ;

2) Если next есть в словаре — переходим к обработке следующего символа;

Иначе в выходной вектор добавляется код для current, next добавляется в словарь с новым кодом, current сбрасывается на текущий символ.

Если после обработки всех символов current не пуст — его код добавляется в результат. Возвращается вектор закодированных значений.

Листинг 38. Метод LZWEncode

```
vector<int> LZWEncode(const string& input) {
// Инициализация словаря ASCII символами
unordered_map<string, int> dictionary;
for (int i = 0; i < 256; i++) {
dictionary[string(1, char(i))] = i;
}

string current;
vector<int> encoded;
int dictSize = 256;

for (char c : input) {
    string next = current + c;
    if (dictionary.find(next) != dictionary.end()) {
        current = next;
    } else {
        encoded.push_back(dictionary[current]);
        dictionary[next] = dictSize++;
        current = string(1, c);
```

```
    }
}

// Добавление последнего кода
if (!current.empty()) {
    encoded.push_back(dictionary[current]);
}

return encoded;
}
```

2.4.7 Функция generateParetoFile

Вход: const string& filename — имя файла для генерации

int size — размер генерируемого файла в символах

Выход: сгенерированный файл.

Метод генерирует файл со случайными символами, распределенными по закону Парето.

Листинг 39. Функция generateParetoFile

```
void generateParetoFile(const string& filename, int size) {
// Очистка файла
ofstream clear_file(filename, ios::trunc);
clear_file.close();

// Открытие файла для записи
ofstream file(filename);
if (!file.is_open()) {
    cerr << "Не удалось открыть файл для записи." << endl;
    return;
}

// Набор символов для генерации
string chars = "абвгдеёжзийклмнопрстуфхцчшщъыьэюя
   0123456789.,!?;:-()\" ";

// Инициализация генератора
random_device rd;
mt19937 gen(rd());

// Генерация файла
for (int i = 0; i < size; ++i) {
    double value = pareto_distribution(20.0, 1.0, gen);
```

```
23      int index = static_cast<int>(value * chars.size()) % chars.
            size();
24      file << chars[index];
25  }
26
27  file.close();
28  }
```

2.4.8 Функция pareto_distribution

Вход: double alpha - параметр формы распределения.

double xm - параметр масштаба.

std::mt19937& gen - генератор случайных чисел.

Выход: double - случайное значение, распределенное по закону Парето.

Функция генерирует случайные числа с распределением Парето.

Листинг 40. Функция pareto$_d$*istribution*

```
1  double pareto_distribution(double alpha, double xm, std::mt19937
       & gen) {
2  // Генерация равномерно распределенного числа [0, 1)
3  std::uniform_real_distribution<> dis(0.0, 1.0);
4  double u = dis(gen);
5
6  // Преобразование в распределение Парето
7  return xm / pow(u, 1.0 / alpha);
8  }
```

2.4.9 Функция LZWEncodeTwoStage

Вход: const string& input — строка для двухэтапного кодирования

Выход: vector<int> — вектор закодированных кодов LZW после RLE-преобразования

Функция реализует двухэтапное кодирование (RLE + LZW).

Листинг 41. Функция LZWEncodeTwoStage

```
1  vector<int> LZWEncodeTwoStage(const string& input) {
2  // Этап 1: RLE–кодирование исходного текста
3  vector<int> rawInput(input.begin(), input.end());
4  vector<pair<int, int>> rleEncoded = advancedRLEEncode(rawInput);
```

```
5
6 // Этап 2: Подготовка строки для LZW
7 string rleString;
8 for (const auto& pair : rleEncoded) {
9     rleString += to_string(pair.first) + ":" + to_string(pair.
      second) + " ";
10 }
11
12 // Этап 3: LZW-кодирование RLE-строки
13 return LZWEncode(rleString);
14 }
```

2.4.10 Функция LZWDecodeTwoStage

Вход: const vector<int>& encoded — вектор кодов LZW, полученных после двухэтапного кодирования.

Выход: string — декодированная исходная строка.

Входные коды LZW преобразуются в строку формата "число:количество" (например, "65:3 66:2"). Строка разбивается на токены по пробелам. Каждый токен разделяется на число и количество повторений по символу ':'. Формируется вектор пар RLE (значение, количество).

Пары преобразуются в последовательность чисел. Числа конвертируются в символы ASCII.

Листинг 42. LZWDecodeTwoStage

```
1 string LZWDecodeTwoStage(const vector<int>& encoded) {
2 string rleString = LZWDecode(encoded);
3 rleDecoded.reserve(rleString.size() / 4); // Оптимизация выделен
    ия памяти
4 vector<pair<int, int>> rleDecoded;
5 istringstream iss(rleString);
6 string token;
7 while (getline(iss, token, ' ')) {  // Явное разделение по пробе
    лам
8     size_t colonPos = token.find(':');
9     if (colonPos != string::npos) {
10         try {
11             int value = stoi(token.substr(0, colonPos));
12             int count = stoi(token.substr(colonPos + 1));
13             rleDecoded.emplace_back(value, count);
14         } catch (...) {
15             // Обработка ошибок преобразования чисел
16             continue;
```

```
17          }
18      }
19  }
20  vector<int> decodedInts = advancedRLEDecode(rleDecoded);
21  return string(decodedInts.begin(), decodedInts.end());
22  }
```

2.4.11 Функция advancedRLEEncode

Вход: const vector<int>& input — вектор целых чисел для кодирования.

Выход: vector<pair<int, int» — вектор пар (значение, количество повторений).

Проверка на пустой входной вектор. Установка первого элемента как текущего значения, счетчик повторений устанавливается в 1.

Если совпадает с текущим значением — увеличиваем счетчик. Если отличается — сохраняем текущую пару (значение, количество) и начинаем новую серию.

Сохранение последней накопленной серии. Возвращается вектор пар RLE.

Листинг 43. Метод advancedRLEEncode

```
1  vector<pair<int, int>> advancedRLEEncode(const vector<int>&
       input) {
2  vector<pair<int, int>> encoded;
3  if (input.empty()) return encoded;
4
5  int current = input[0];
6  int count = 1;
7
8  for (size_t i = 1; i < input.size(); ++i) {
9      if (input[i] == current) {
10         count++;
11     } else {
12         encoded.emplace_back(current, count);
13         current = input[i];
14         count = 1;
15     }
16 }
17 encoded.emplace_back(current, count);
18
19 return encoded;
20 }
```

2.4.12 Функция advancedRLEDecode

Вход: const vector<pair<int, int»& encoded — вектор пар RLE (значение, количество повторений)

Выход: vector<int> — восстановленный вектор данных

Создается пустой вектор для результатов. Для каждой пары (значение, количество):

1) Значение добавляется в результат указанное количество раз,

2) Используется вложенный цикл для повторений.

Возвращается полностью восстановленный вектор.

Листинг 44. advancedRLEDecode

```
vector<int> advancedRLEDecode(const vector<pair<int, int>>&
    encoded) {
vector<int> decoded;
// Предварительное вычисление размера результата
size_t total_size = 0;
for (const auto& pair : encoded) {
total_size += pair.second;
}
decoded.reserve(total_size); // Оптимизация выделения памяти

// Основной цикл декодирования
for (const auto& pair : encoded) {
    decoded.insert(decoded.end(), pair.second, pair.first);
}

return decoded;
}
```

2.4.13 Функция getFileSize

Вход: const string& filename — путь к файлу, размер которого нужно определить

Выход: size_t — размер файла в байтах (0 в случае ошибки)

Файл открывается в бинарном режиме (ios::binary). Указатель сразу устанавливается в конец файла (ios::ate). Если файл не открылся, воз-

вращается 0. Позиция указателя в конце файла (tellg()) соответствует размеру файла в байтах.

Листинг 45. Функция getFileSize

```
size_t getFileSize(const string& filename) {
// Открываем файл и сразу ставим указатель в конец
ifstream file(filename, ios::binary | ios::ate);
if (!file.is_open()) {
return 0; // Файл не существует или ошибка доступа
}
return static_cast<size_t>(file.tellg());
}
```

3 Результаты работы программы

При запуске программы мы видим стартовое окно, предлагающее выбрать с чем мы хотим работать (см. [Рис. 3]).

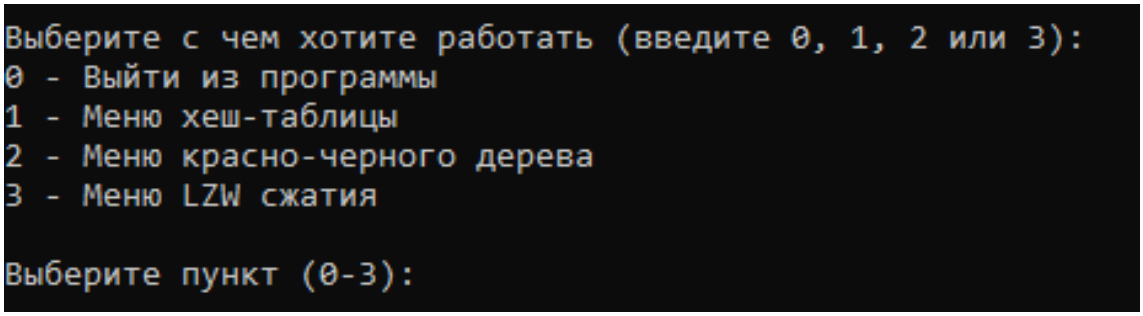

Рис. 3. Главное меню

Предусмотрена защита от некорректного пользовательского ввода в каждом меню. При вводе 1, мы переходим в меню работы с хэш-таблицей (см. [Рис. 4]).

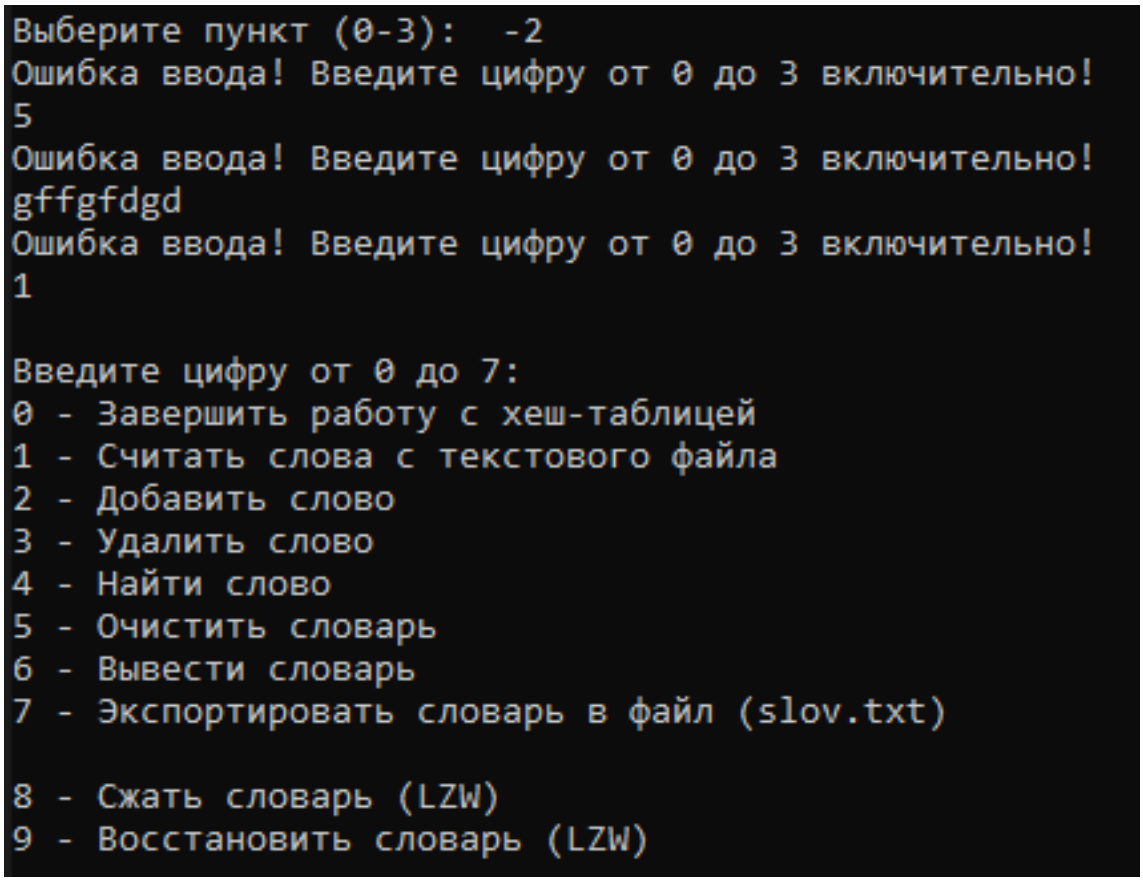

Рис. 4. Меню хэш-таблицы

Предусмотрена недоступность пунктов, требующих предыдущих действий (пункты 7, 8, 9) (см. [Рис. 5]).

```
Выберите пункт (0-9):  9
Ошибка: сначала сожмите словарь (пункт 8).

Введите цифру от 0 до 7:
0 - Завершить работу с хеш-таблицей
1 - Считать слова с текстового файла
2 - Добавить слово
3 - Удалить слово
4 - Найти слово
5 - Очистить словарь
6 - Вывести словарь
7 - Экспортировать словарь в файл (slov.txt)

8 - Сжать словарь (LZW)
9 - Восстановить словарь (LZW)

Выберите пункт (0-9):  8
Ошибка: сначала экспортируйте словарь (пункт 7).
```

Рис. 5. Защита от пользования методами без выполнения предыдущих пунктов

При вводе 1, можно выбрать способ загрузки файлов. Есть 3 заготовленных файла с текстами разного объема для работы (пункт 1-3) (см. [Рис. 6]).

```
Выберите пункт (0-9):  1

Выберите способ загрузки файла:
1 - Быстрая загрузка (dictionary1.txt - 'Светлая полоса')
2 - Быстрая загрузка (dictionary2.txt - 'Белые ночи')
3 - Быстрая загрузка (dictionary3.txt - 'Пословицы')
4 - Ручной ввод имени файла
Ваш выбор: 2
```

Рис. 6. Загрузка файла

При вводе 5 в меню происходит очистка словаря (см. [Рис. 7]).

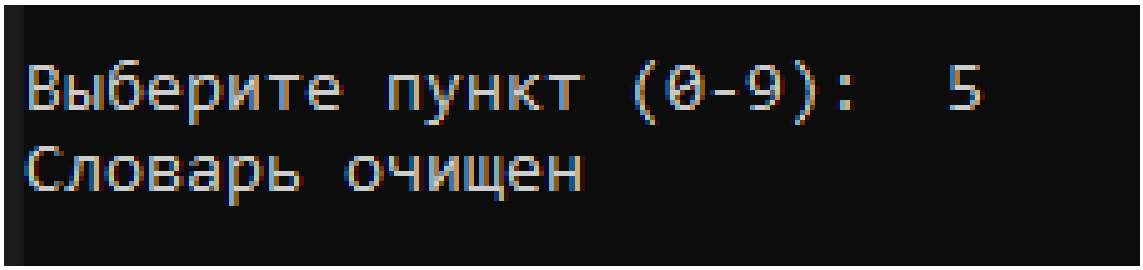

Рис. 7. очистка словаря

В пункте 1 предусмотрена защита от некорректного ввода имени файла при выборе четвертой опции (см. [Рис. 8]).

```
Выберите пункт (0-9):  1

Выберите способ загрузки файла:
1 - Быстрая загрузка (dictionary1.txt - 'Светлая полоса')
2 - Быстрая загрузка (dictionary2.txt - 'Белые ночи')
3 - Быстрая загрузка (dictionary3.txt - 'Пословицы')
4 - Ручной ввод имени файла
Ваш выбор: dict
Ошибка ввода! Введите цифру от 1 до 4 включительно!
4
Введите название файла: dict
Файл не найден. Пожалуйста, попробуйте еще раз.
Введите название файла: dictionary0.txt
```

Рис. 8. Ручной ввод имени файла

Пункт 2 позволяет добавлять слова в словарь, также предусмотрена защита от некорректного ввода (см. [Рис. 9]).

```
Выберите пункт (0-9):  2
Введите слово: efwefwefw
Слово содержит цифру или другой неподходящий символ. Повторите ввод.
папб,
Слово содержит цифру или другой неподходящий символ. Повторите ввод.
кот
Слово 'кот' успешно добавлено в словарь
```

Рис. 9. Добавление слова

Если слово уже есть в словаре, программа сообщит об этом (см. [Рис. 10]).

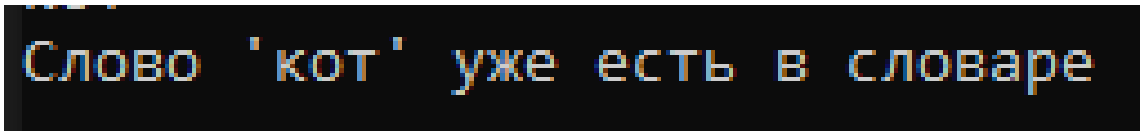

Рис. 10. Добавление слова, которое уже есть

Пункт 3 служит для удаления слов. Если слово отсутствует в словаре выведется сообщение (см. [Рис. 11]).

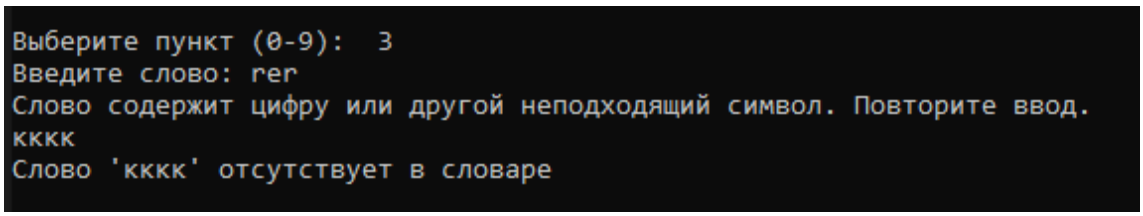

Рис. 11. Удаление слова, которого нет в словаре

Если слово есть в словаре, то оно будет успешно удалено (см. [Рис. 12]).

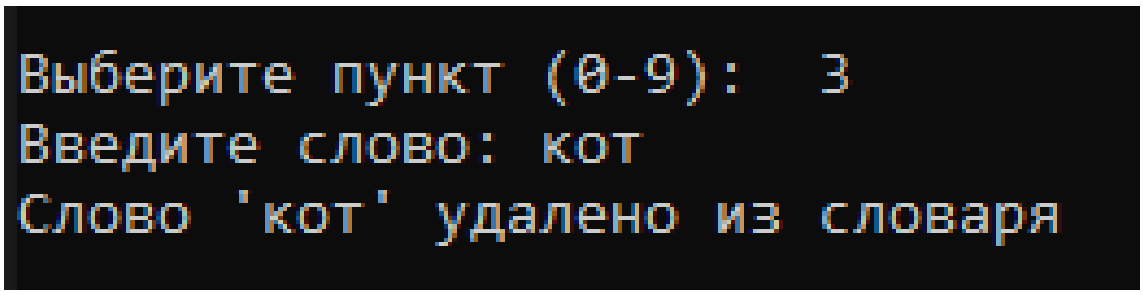

Рис. 12. Удаление слова

4 - поиск слова. Если слово не найдено - выведется сообщение (см. [Рис. 13]).

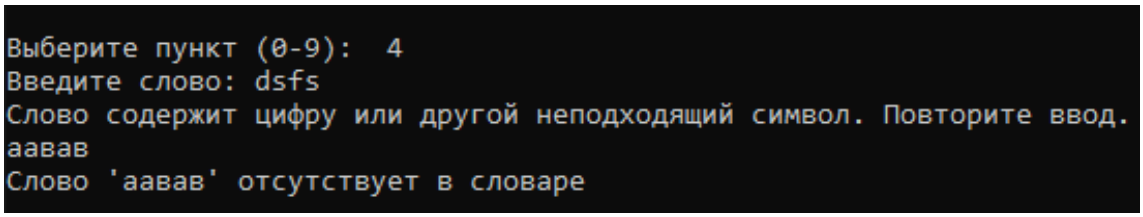

Рис. 13. Поиск слова

Если слово найдено, то также будет выведен его индекс в хэш-таблице (см. [Рис. 14]).

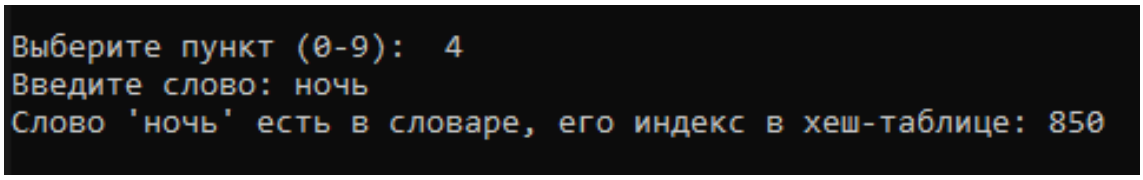

Рис. 14. Успешный поиск слова

Вывод содержимого хэш-таблицы выглядит следующим образом (пункт 6) (см. [Рис. 15]).

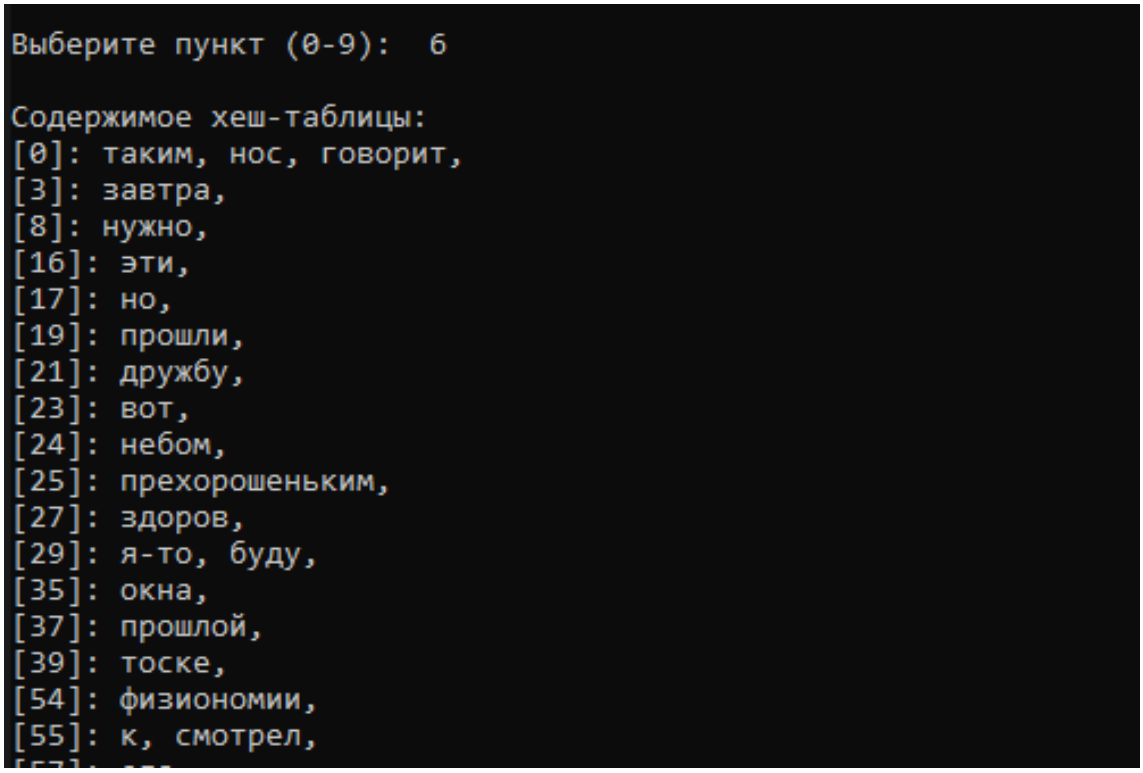

Рис. 15. Вывод хэш-таблицы

Также внизу после таблицы будет статистика (см. [Рис. 16]).

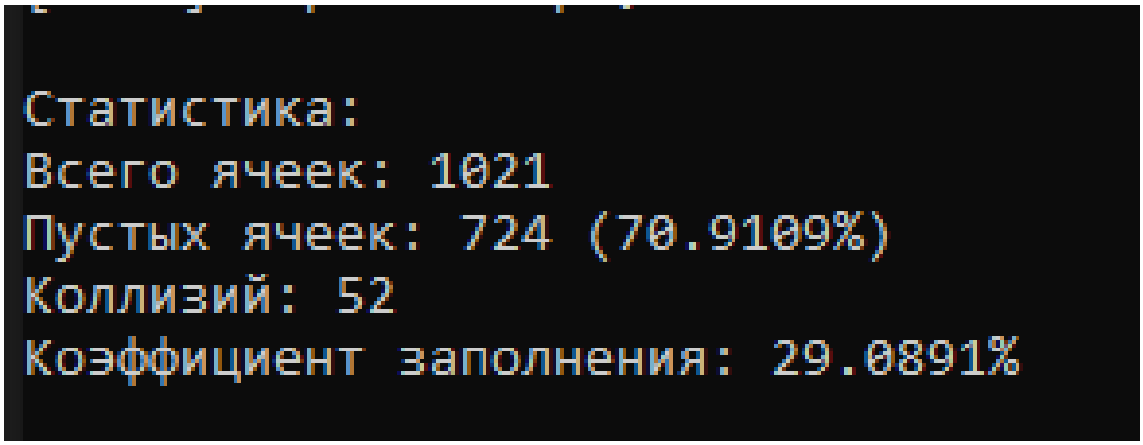

Рис. 16. Статистика

Пункт 7 позволяет экспортировать словарь в файл (см. [Рис. 17]).

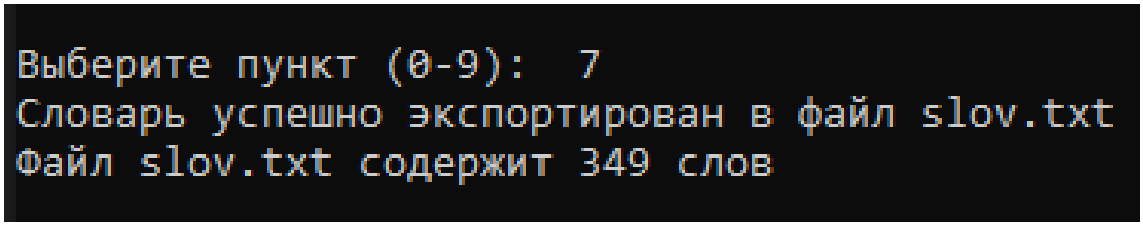

Рис. 17. Экспорт словаря в файл

8 - кодирование LZW (см. [Рис. 18]).

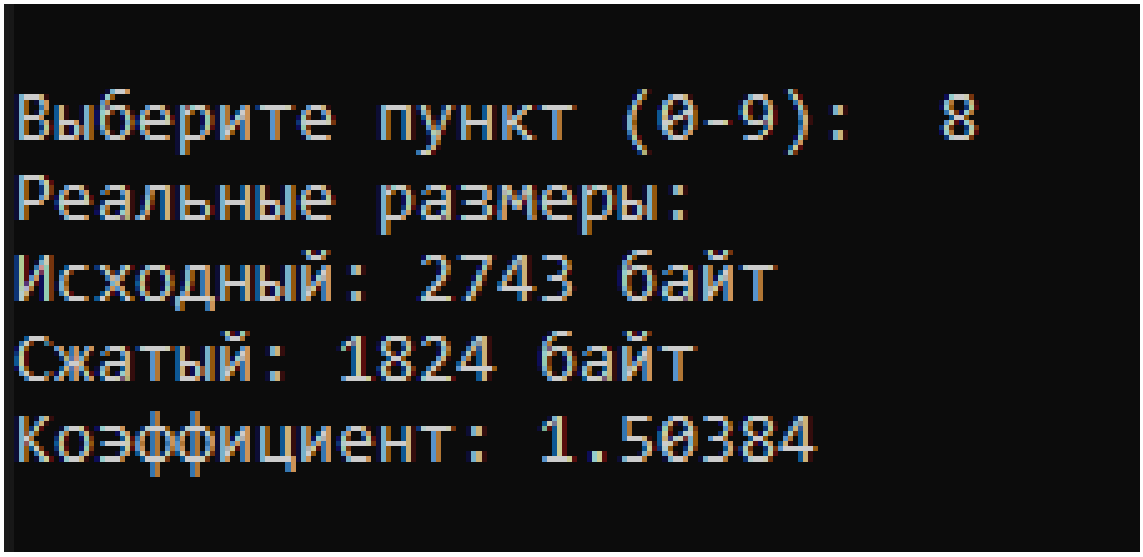

Рис. 18. Кодирование LZW

9 - декодирование и проверка идентичности файлов до кодирования и после (см. [Рис. 19]).

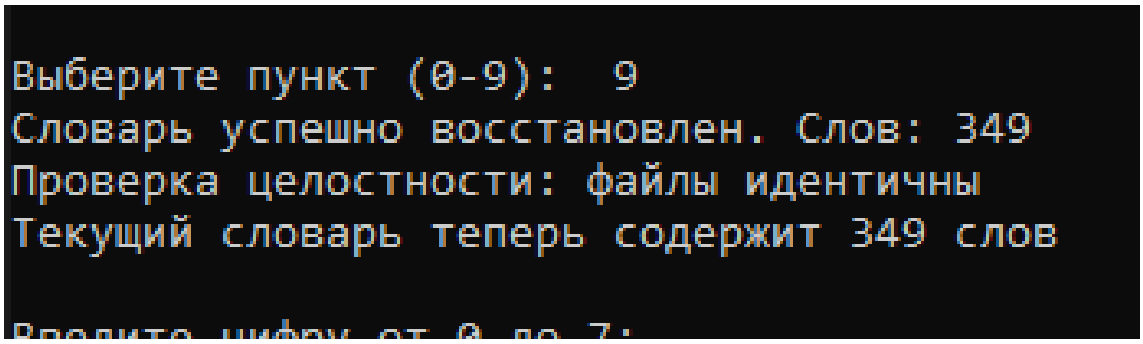

Рис. 19. Декодирование

При вводе нуля мы возвращаемся в главное меню (см. [Рис. 20]).

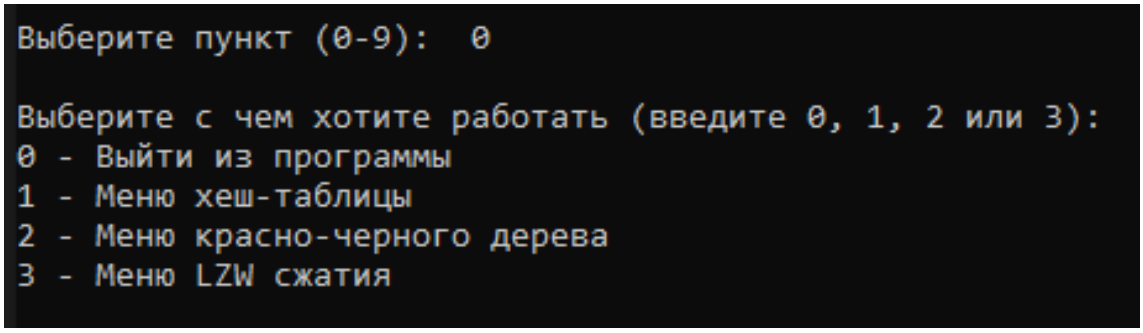

Рис. 20. Выход из меню хэш-таблицы

При вводе 2 мы попадаем в меню красно-черного дерева (см. [Рис. 21]).

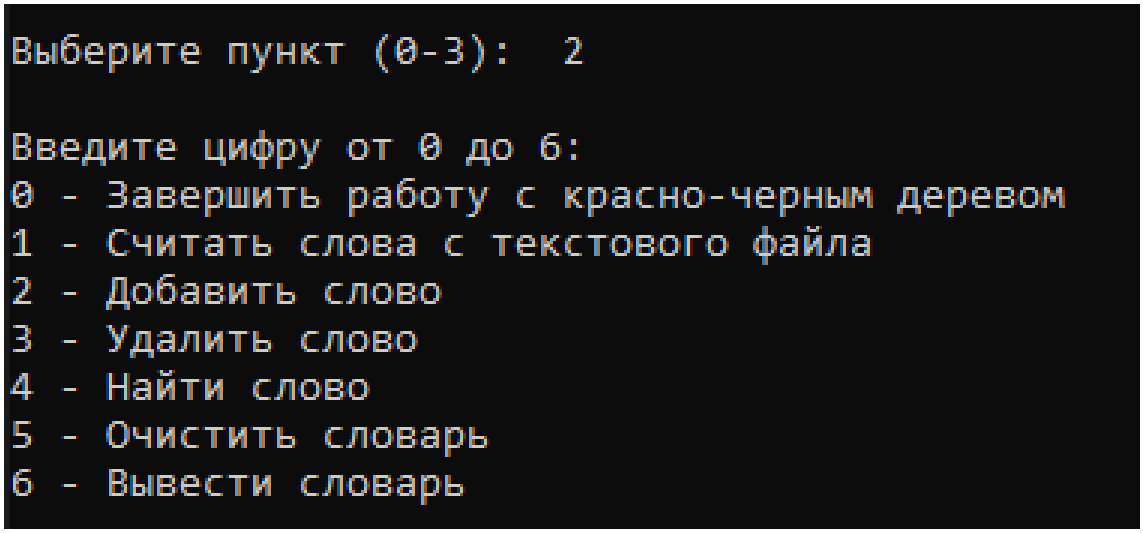

Рис. 21. Меню красно-черного дерева

Вводим 1 и загружаем файл так же как и с хэш-таблицей (см. [Рис. 22]).

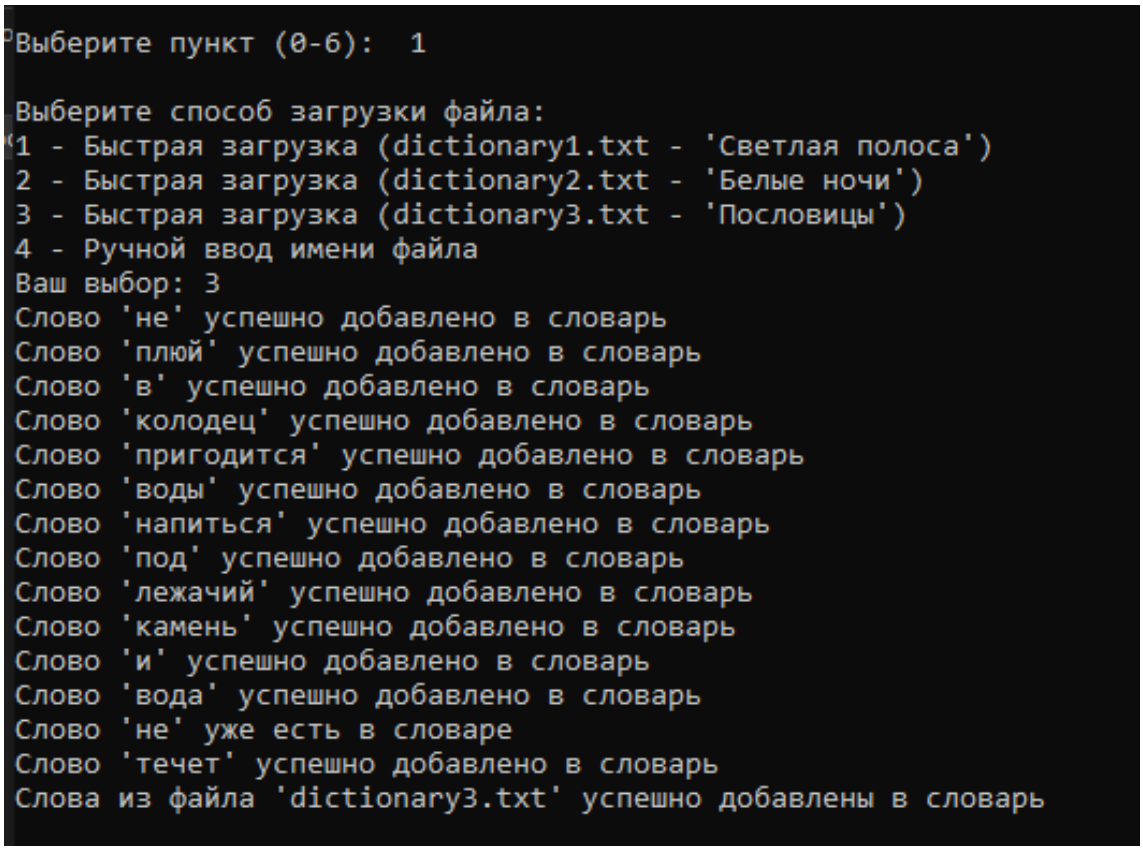

Рис. 22. Загрузка из файла

При вводе 2 мы можем добавить слово. Предусмотрена защита от некорректного ввода. Если слово уже есть, то мы получаем сообщение об этом (см. [Рис. 23]).

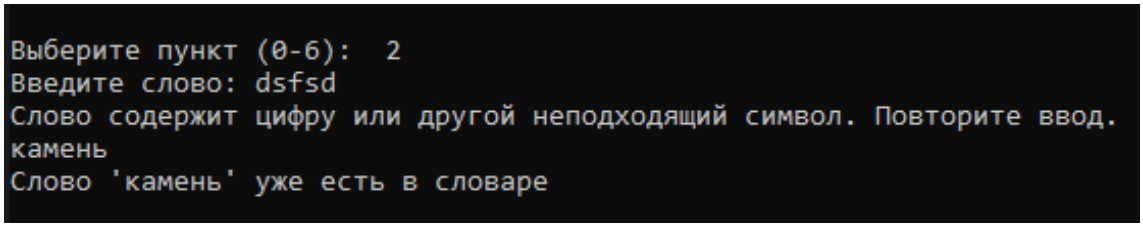

Рис. 23. Добавление слова, которое уже есть

Если слова нет, то оно успешно добавляется (см. [Рис. 24]).

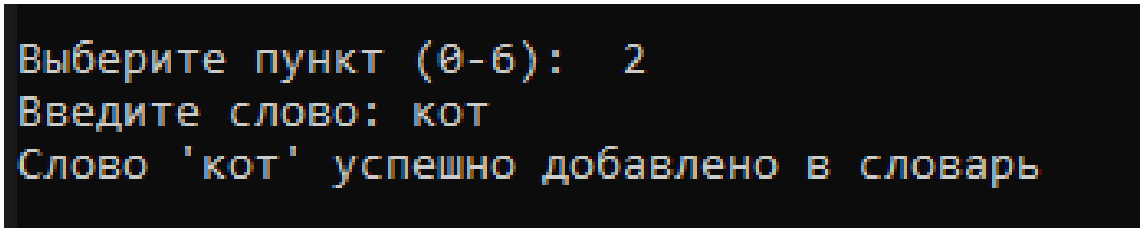

Рис. 24. Добавление слова

Пункт 3 служит для удаления слова. Если слова нет в словаре - оно не будет удалено (см. [Рис. 25]).

```
Выберите пункт (0-6):  3
Введите слово: fdf
Слово содержит цифру или другой неподходящий символ. Повторите ввод.
аааааaa
Слово 'аааааaa' отсутствует в словаре
```

Рис. 25. Удаления слова, которого нет

Если слово есть - успешно удалится (см. [Рис. 26]).

```
Выберите пункт (0-6):  3
Введите слово: кот
Слово 'кот' удалено из словаря
```

Рис. 26. Удаление слова

Поиск слова - пункт 4. Если слова нет в словаре, то мы получим сообщение об этом (см. [Рис. 27]).

```
Выберите пункт (0-6):  4
Введите слово: dfdfdf
Слово содержит цифру или другой неподходящий символ. Повторите ввод.
кот
Слово 'кот' отсутствует в словаре
```

Рис. 27. Поиск слова, которого нет

Если слово есть, то мы получим вывод этого узла с информацией (см. [Рис. 28]).

```
Выберите пункт (0-6):  4
Введите слово: колодец
колодец (B) [parent: камень] {L: NIL, R: NIL}
```

Рис. 28. Успешный поиск слова

5 - очищение словаря (см. [Рис. 29]).

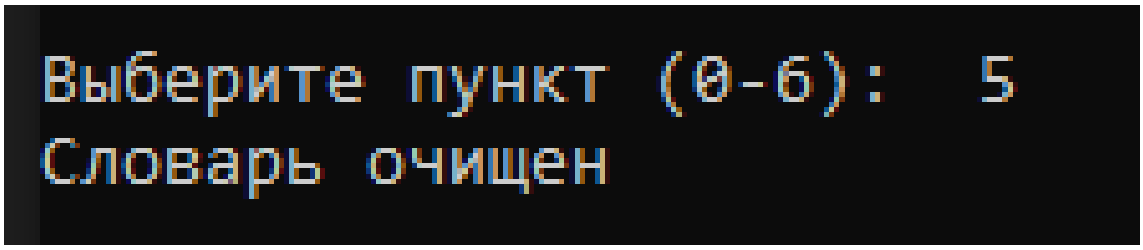

Рис. 29. очищение словаря

Пункт 6 - вывод дерева. Если слов нет - то вывод сообщения (см. [Рис. 30]).

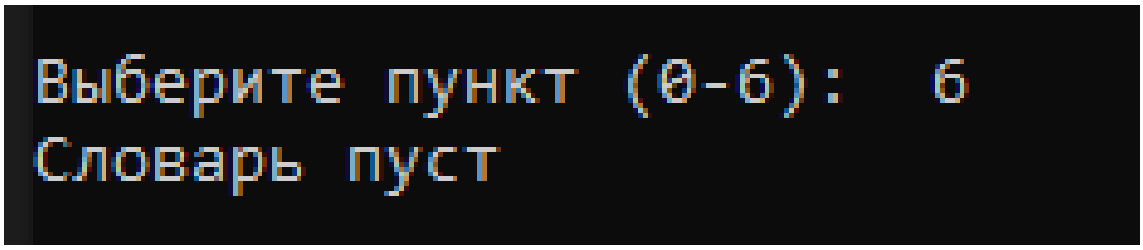

Рис. 30. Вывод пустого дерева

Иначе выводим дерево (см. [Рис. 31]).

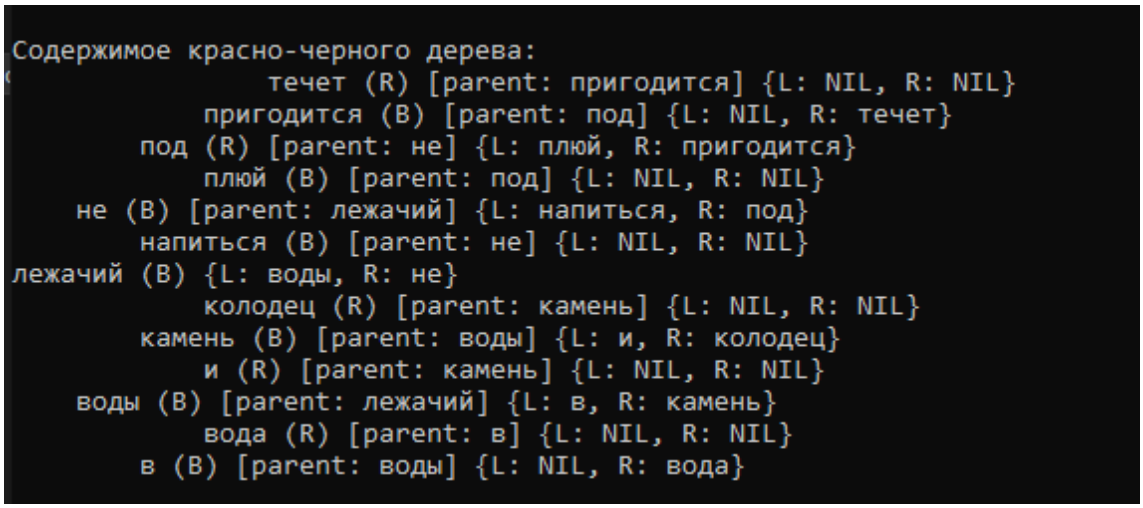

Рис. 31. Вывод дерева

Пункт 3 в главном меню - работа с LZW. Выводится меню (см. [Рис. 32]).

```
Выберите пункт (0-3):  3

Введите цифру от 0 до 5:
0 - Завершить работу с LZW
1 - Сгенерировать случайный файл (10,000 символов)
2 - Закодировать файл (LZW)
3 - Декодировать файл (LZW)
4 - Проверить корректность декодирования
5 - Показать статистику сжатия
```

Рис. 32. Меню работы с LZW

Нельзя кодировать файл, декодировать файл, посмотреть статистику и проверить корректность, не сгенерировав файл и не выполня предыдущие пункты (см. [Рис. 33]).

```
Выберите пункт (0-6):  4
Ошибка! Сначала выполните декодирование (пункт 3).

Введите цифру от 0 до 5:
0 - Завершить работу с LZW
1 - Сгенерировать случайный файл (10,000 символов)
2 - Закодировать файл (LZW)
3 - Декодировать файл (LZW)
4 - Проверить корректность декодирования
5 - Показать статистику сжатия

Выберите пункт (0-6):  5
Ошибка! Для статистики нужно:
1. Сгенерировать файл (пункт 1)
2. Выполнить кодирование (пункт 2)

Введите цифру от 0 до 5:
0 - Завершить работу с LZW
1 - Сгенерировать случайный файл (10,000 символов)
2 - Закодировать файл (LZW)
3 - Декодировать файл (LZW)
4 - Проверить корректность декодирования
5 - Показать статистику сжатия
```

Рис. 33. Защита от декодирования до кодирования

Пункт 1 - генерация файла (см. [Рис. 34]).

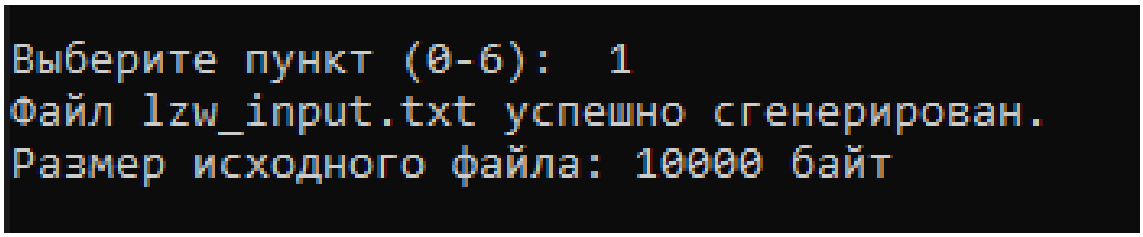

Рис. 34. Генерация файла

Пункт 2 - кодирование (и одноступенчатое, и двуступенчатое) (см. [Рис. 35]).

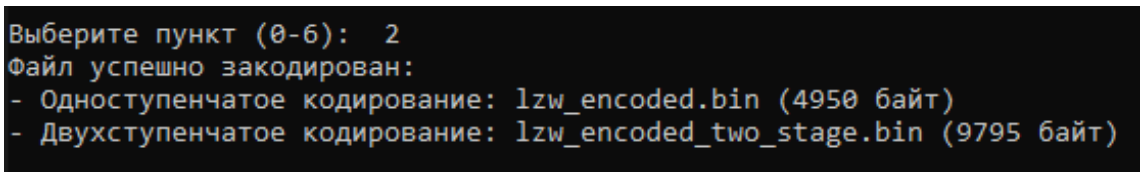

Рис. 35. Кодирование

Пункт 3 - декодирование файлов (см. [Рис. 36]).

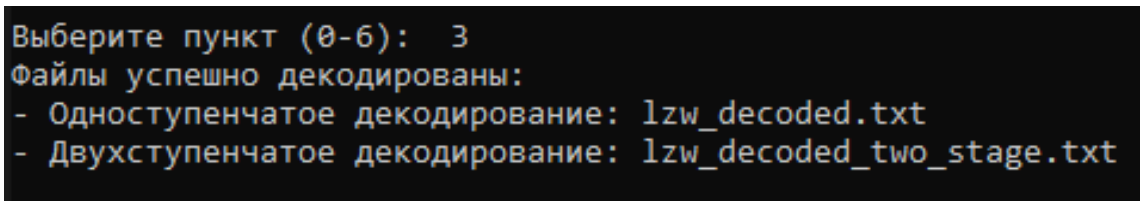

Рис. 36. Декодирование

Пункт 4 - результаты проверки декодированных файлов на соответствие изначальному (см. [Рис. 37]).

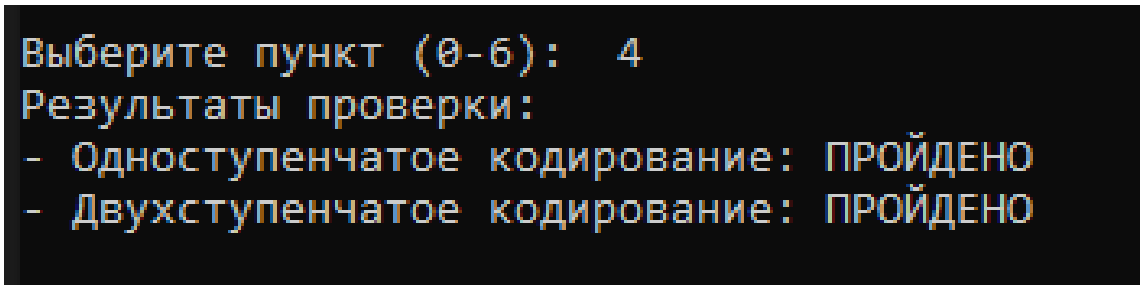

Рис. 37. Проверка кодирования и декодирования

Пункт 5 - статистика сжатия (см. [Рис. 38]).

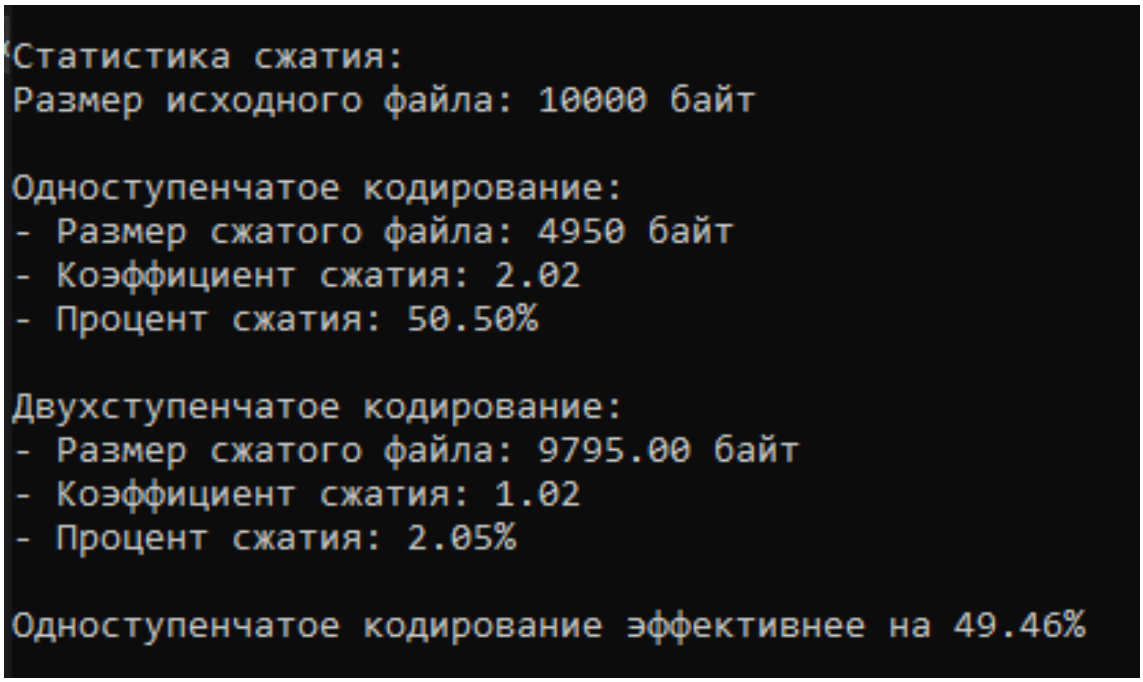

Рис. 38. Статистика

При вводе нуля в главном меню программа завершает работу (см. [Рис. 39]).

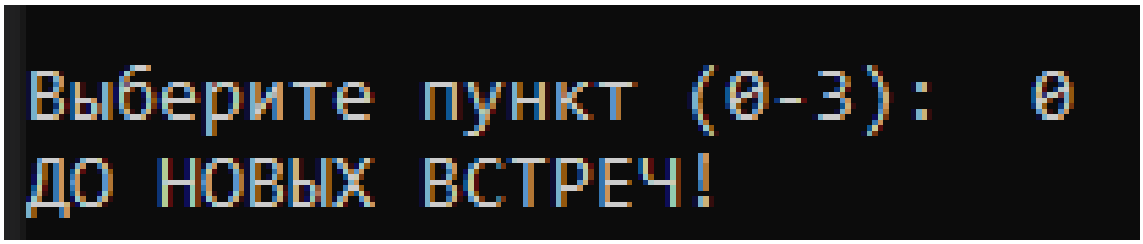

Рис. 39. Завершение работа

Заключение

В результате проделанной работы была успешно реализована программа, состоящая из нескольких ключевых модулей. Первым этапом стала реализация хеш-таблицы для хранения словаря с поддержкой базовых операций: добавления, удаления и поиска элементов. В качестве хеш-функции была выбрана FNV-1a, которая продемонстрировала хорошую устойчивость к коллизиям при равномерном распределении ключей. Дополнительно были реализованы функции полной очистки таблицы и загрузки данных из внешнего текстового файла.

Вторым этапом работы стало построение альтернативной версии словаря на основе красно-черного дерева с аналогичным набором операций. Особое внимание было уделено правильной реализации механизмов балансировки дерева при вставке и удалении узлов.

Третья часть работы включала генерацию тестовых данных объемом 10 тысяч символов с распределением Парето и последующее их сжатие с использованием алгоритма LZW. В ходе экспериментов были получены результаты эффективности кодировани. Дополнительно исследовалась схема двухступенчатого сжатия (LZW+RLE), которая показала результаты хуже по сравнению с базовым LZW. Символы распределены по закону Парето. Нет длинных повторяющихся последовательностей, а алгоритм RLE эффективен только при повторениях от 3–4 одинаковых символов подряд. RLE не сокращает, а увеличивает размер. Затем LZW вынужден сжимать уже искусственно раздутые данные. Также RLE разбивает данные на одиночные символы + счетчики, уничтожая естественные повторения. LZW получает на вход менее избыточные данные и сжимает их хуже.

Финальным этапом стало применение реализованного алгоритма сжатия к исходному текстовому файлу со словарными данными.

Недостатком данной программы является четко ограниченный размер хэш-таблицы (1021) загрузка слишком короткого текста даст много пустых ячеек, а слишком большого увеличит количество коллизий. Красно-черное дерево недостаточно наглядно, сложно воспринимать информацию в таком виде.

Преимуществом программы является вывод статистики как и для хэш-таблицы, так и для кодирования, что позволяет судить о результатах операций. В операциях поиска, удаления при вводе предусмотрена защита от некорректных символов, что позволяет не просматривать весь

словарь в поисках слова, а сразу выдать ошибку ввода, это ускоряет работу программы.

Дополнительно могут быть реализованы и другие алгоритмы кодирования для сравнения их эффективностей (например, Фано, Хаффмана). Может быть расширен функционал красно-черного дерева добавлением операций разрезания дерева и объединения красно-чёрных деревьев. Для хэш-функции можно сделать расчет "идеального"размера таблице на основе количества слов в загружаемом тексте.

Список использованной литературы

[1] Графы и алгоритмы. Новиков Ф. А. Дискретная математика для программистов. - Санкт-Петербург : Питер Пресс, 2000 - 304 с.

[2] Красно-черное дерево, кодирование информации, алгоритм LZW.
// URL: https://neerc.ifmo.ru/wiki/
(дата обращения: 9.05.2025).

[3] Как работает хэширование.
// URL: https://habr.com/en/companies/ruvds/articles/747084/
(дата обращения: 11.05.2025).

[4] Hash.
// URL: https://wiki.rakovets.by/v0/hashing/fnv/
(дата обращения: 11.05.2025).