МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт Компьютерных наук и кибербезопастности

Направление 02.03.01 Математика и компьютерные науки

Высшая школа технологий искусственного интеллекта

ОТЧЕТ ПО РАБОТЕ ПО ТЕОРИИ ГРАФАХ:

Реализация словаря на основе хэш-таблицы и красно-черного дерева, кодирование данных

Вариант 29

Обучающийся: _____ Санько В. В.

Руководитель: _____ Востров А. В.

«_____»_____ 20___г.

Санкт-Петербург, 2025

Содержание

Введение

Одной из основных задач при работе с текстами является создание словарей. В рамках данной работы рассматриваются два подхода к построению словаря: на основе хеш-таблицы и на основе самобалансирующегося красно-черного дерева. Также необходимо уделить особое внимание выбору хеш-функции для хеш-таблицы, так как необходимо минимизировать количество коллизий. Помимо этого применить алгоритм сжатия данных (продвинутый RLE и алгоритм Фано) на текстовой информации на русском языке.

1 Математическое описание

1.1 Хеш-таблица

Хеш-таблица представляет собой структуру данных, предназначенную для эффективной реализации интерфейса ассоциативного массива, то есть отображения ключей K на значения V. Пусть имеется множество ключей U (универсум ключей). Хеш-таблица использует массив T размера m, называемый таблицей, и хеш-функцию $h : U \to \{0, 1, \ldots, m-1\}$. Хеш-функция отображает ключ $k \in U$ в индекс $h(k)$ в массиве T. Элемент $T[j]$ называется корзиной (bucket).

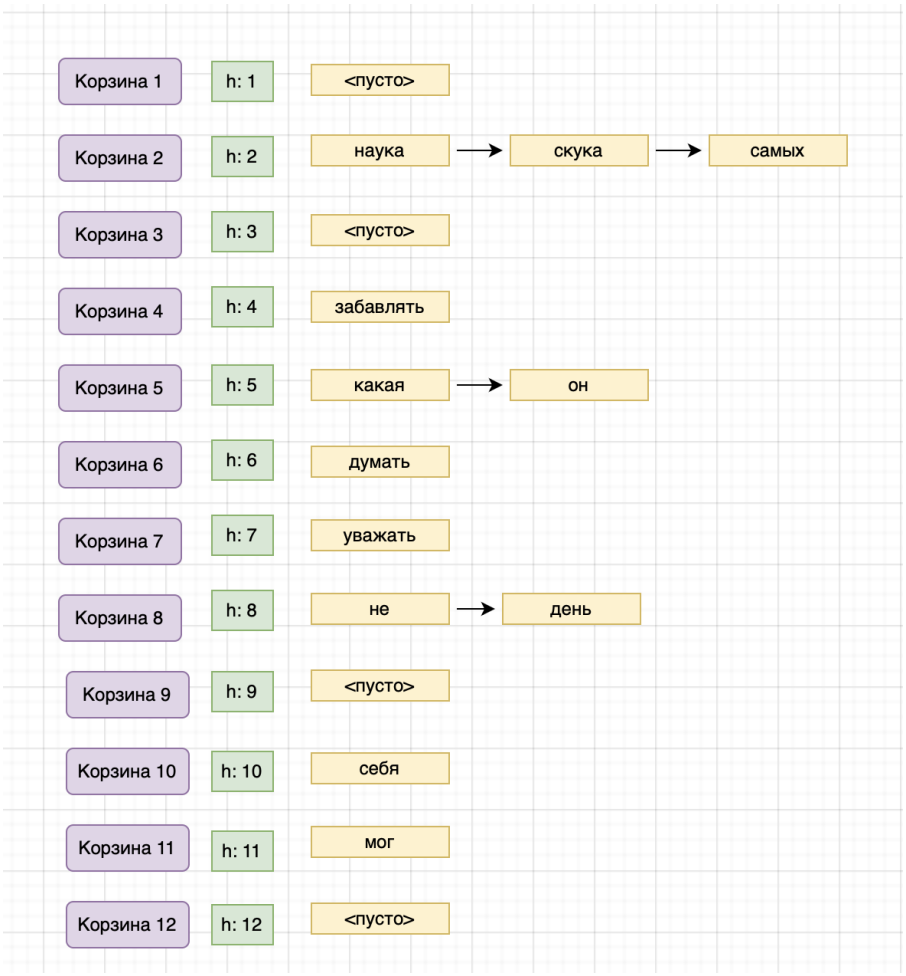

Рис. 1. Хеш-таблица.

Хеш-функция. В данной работе для строковых ключей $S = c_0 c_1 \ldots c_{L-1}$ (где c_i — байтовое представление i-го символа строки, L — длина строки в байтах) применяется полиномиальная хеш-функция. Значение хеш-функции вычисляется следующим образом:

$$h(S) = \left(\sum_{i=0}^{L-1} \mathrm{ord}(c_i) \cdot p^i \right) \pmod{m}$$

где $\text{ord}(c_i)$ — числовое значение байта c_i (например, его ASCII-код или значение как ‘unsigned char‘), p — некоторое простое число, выбранное в качестве основания полинома ($p = 31$), и m — текущий размер хеш-таблицы (количество корзин). Операция взятия по модулю m обеспечивает отображение результата в диапазон индексов таблицы $[0, m-1]$. Для предотвращения переполнения при вычислении больших степеней p^i и больших сумм, модульная арифметика применяется на каждом шаге итеративного вычисления:

1. Инициализация: $hash\_value = 0$, $p\_power = 1$.
2. Для каждого символа c_i из S (от $i = 0$ до $L - 1$):
   (a) В LaTeX это будет:
   $$hash\_value = (hash\_value + \text{ord}(c_i) \cdot p\_power) \pmod{m}.$$
   (b) В LaTeX это будет:
   $$p\_power = (p\_power \cdot p) \pmod{m}.$$
3. Результат: $h(S) = hash\_value$.

В реализации значение $\text{value}(c_i)$ (эквивалент $\text{ord}(c_i)$ в формуле) бралось как ‘static_cast<unsigned char>(c_byte)‘.

Разрешение коллизий. Коллизия возникает, если для двух различных ключей $k_1 \neq k_2$ их хеш-значения совпадают: $h(k_1) = h(k_2)$. В данной реализации для разрешения коллизий используется **метод цепочек (separate chaining)**. Каждая корзина $T[j]$ хеш-таблицы является указателем на начало связанного списка (или другой динамической структуры), хранящего все пары (k, v), для которых $h(k) = j$. Операции вставки, поиска и удаления элемента с ключом k требуют:

1. Вычисления индекса $j = h(k)$.
2. Выполнения соответствующей операции (поиск, вставка, удаление) в связанном списке, ассоциированном с корзиной $T[j]$.

Средняя длина цепочки при равномерном распределении ключей и коэффициенте загрузки $\alpha = N/m$ (где N — количество элементов в таблице) составляет α. Таким образом, ожидаемое время выполнения операций составляет $O(1 + \alpha)$ без учета времени вычисления хеш-функции.

Коэффициент загрузки и рехеширование. Коэффициент загрузки $\alpha = N/m$ является важным показателем производительности хеш-таблицы с методом цепочек. При увеличении α средняя длина цепочек растет, что приводит к увеличению времени выполнения операций. Для поддержания эффективности, когда α достигает некоторого порогового значения α_{max} (в реализации $\alpha_{max} = 0.75$), выполняется операция рехеширования. Она заключается в создании новой

таблицы большего размера m' (обычно $m' \approx 2m$), и все N элементов из старой таблицы переносятся в новую с пересчетом их хеш-кодов по новой хеш-функции $h' : U \to \{0, 1, \ldots, m' - 1\}$. Амортизированная стоимость операции вставки при использовании рехеширования остается $O(1 + \alpha)$ (при условии, что стоимость рехеширования $O(m + N)$ "размазывается"по достаточному количеству вставок).

1.2 Красно-черное дерево

Красно-черное дерево (КЧ-дерево) — это тип самобалансирующегося двоичного дерева поиска. Каждому узлу в дереве приписывается дополнительный атрибут — цвет (красный или черный). За счет поддержания инвариантов, связанных с цветами узлов и структурой дерева, КЧ-деревья гарантируют, что ни один путь от корня до любого листа не будет более чем в два раза длиннее любого другого такого пути. Это обеспечивает логарифмическую высоту дерева $h = O(\log N)$, где N — количество узлов.

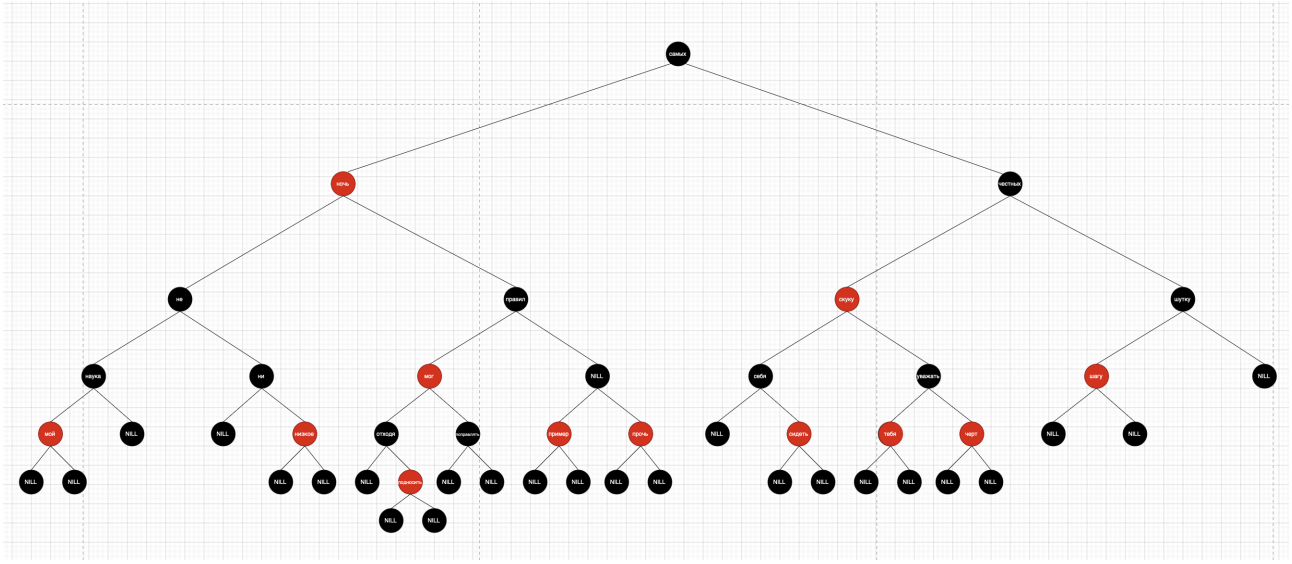

Рис. 2. Красно-черное дерево.

Свойства (инварианты) красно-черного дерева:

1. **Свойство цвета:** Каждый узел является либо красным (RED), либо черным (BLACK).
2. **Свойство корня:** Корень дерева всегда черный.
3. **Свойство листьев:** Все внешние узлы (листья, представляемые в реализации одним стражевым NIL-узлом) являются черными.
4. **Свойство красных узлов:** Если узел красный, то оба его дочерних узла (если они существуют и не являются NIL) должны быть черными. Это означает, что не может быть двух красных узлов, следующих друг за другом на пути от корня к листу.

5. **Свойство черной высоты:** Для любого узла x все простые пути от x до листовых узлов (NIL) в его поддереве содержат одинаковое количество черных узлов. Это количество называется черной высотой узла x, обозначается $bh(x)$.

Операции и балансировка. Для поддержания свойств КЧ-дерева при вставке или удалении узлов используются две основные операции: перекрашивание узлов и вращения.

1.2.1 Перекрашивание узлов

Это простейшая операция, заключающаяся в изменении цвета узла с красного на черный или с черного на красный. Сама по себе эта операция не изменяет структуру дерева, но используется в комбинации с вращениями для восстановления свойств КЧ-дерева.

1.2.2 Вращения (Rotations)

Вращения — это локальные структурные преобразования в двоичном дереве, которые изменяют связи «родитель-потомок», сохраняя при этом свойство упорядоченности двоичного дерева поиска (т.е. для любого узла x все ключи в его левом поддереве меньше $key[x]$, а все ключи в его правом поддереве больше $key[x]$). Вращения используются для изменения высот поддеревьев и помогают сбалансировать дерево. Существует два типа вращений: левое и правое. Они выполняются за константное время $O(1)$.

Левое вращение (Left Rotate) Левое вращение выполняется вокруг узла x, у которого есть правый дочерний узел y (не NIL). В результате вращения y становится новым корнем поддерева, которое ранее возглавлял x. Узел x становится левым дочерним узлом y. Левое поддерево y становится правым поддеревом x.

Правое вращение (Right Rotate) Правое вращение симметрично левому. Оно выполняется вокруг узла y, у которого есть левый дочерний узел x (не NIL). В результате x становится новым корнем поддерева, а y — его правым дочерним узлом. Правое поддерево x становится левым поддеревом y.

1.2.3 Операция Вставки (Insertion)

Вставка нового узла в КЧ-дерево включает два этапа:

1. Стандартная вставка узла как в обычном двоичном дереве поиска. Новый узел z всегда окрашивается в **красный** цвет. Его дочерние узлы (NIL) являются черными.

2. Вызов процедуры `RB-Insert-Fixup(T, z)` для восстановления свойств КЧ-дерева, если они были нарушены.

Вставка красного узла может нарушить только свойство 2 (если z — корень и он красный) или свойство 4 (если родитель z также красный). Свойство 5 (черная высота) не нарушается, так как мы добавляем красный узел, который не меняет количество черных узлов на путях.

RB-Insert-Fixup(T, z) Эта процедура восстанавливает свойства КЧ-дерева. Основной цикл продолжается, пока родитель узла z ($z.parent$) красный (что нарушает свойство 4). Обозначим $P = z.parent$ и $G = z.parent.parent$ (дедушка z). Рассматриваются три случая (и их симметричные аналоги).

Сценарий: Родитель P узла z является левым ребенком дедушки G. (Если P является правым ребенком G, случаи симметричны, с заменой «лево» на «право» и наоборот).

- **Случай 1: Дядя U узла z (правый ребенок G) красный.**
  - $P.color \leftarrow BLACK$
  - $U.color \leftarrow BLACK$
  - $G.color \leftarrow RED$
  - $z \leftarrow G$ (Проблема «два красных подряд» перемещается на два уровня вверх. Цикл продолжается для нового $z = G$).

  Логика: Мы «проталкиваем» красноту дедушки вверх. Черная высота не нарушается, так как на путях через P и U один красный узел заменяется черным, а на путях через G один черный узел (старый G) заменяется красным, но его дети (P, U) стали черными, компенсируя это.

- **Случай 2: Дядя U узла z черный, и z является правым ребенком P (конфигурация «треугольник»).**
  - $z \leftarrow P$
  - `Left-Rotate(T, z)` (т.е. вокруг старого P)

  Логика: Это преобразование сводит ситуацию к Случаю 3. Узел z (теперь это старый P) и его новый родитель (старый z) по-прежнему красные, но теперь они образуют «линию».

- **Случай 3: Дядя U узла z черный, и z является левым ребенком P (конфигурация «линия»).**
  - $P.color \leftarrow BLACK$
  - $G.color \leftarrow RED$
  - `Right-Rotate(T, G)`

Логика: Это преобразование устраняет нарушение свойства 4. Правое вращение вокруг G делает P новым корнем этого поддерева. P становится черным, его ребенок z остается красным, а другой ребенок G (бывший родитель P) становится красным. Поскольку P теперь черный, свойство 4 не нарушено. Цикл завершается.

После завершения цикла свойство 2 (корень должен быть черным) восстанавливается принудительным окрашиванием $T.root.color \leftarrow BLACK$.

1.2.4 Операция Удаления (Deletion)

Удаление узла из КЧ-дерева является более сложной операцией.

1. Сначала выполняется стандартная процедура удаления узла из двоичного дерева поиска. Это сводится к удалению узла y, который имеет не более одного не-NIL ребенка. Пусть x — это единственный ребенок y (или $T.NIL$, если у y нет детей), который занимает место y.
2. Сохраняется исходный цвет удаляемого узла y ($y\_original\_color$).
3. Если $y\_original\_color = RED$, то свойства КЧ-дерева не нарушаются. Красные узлы не влияют на черную высоту, и их удаление не может создать два красных узла подряд или сделать корень красным (если он не был единственным узлом).
4. Если $y\_original\_color = BLACK$, то удаление черного узла нарушает свойство 5 (черная высота) для всех путей, проходивших через y. Также могут быть нарушены свойства 2 или 4. Для восстановления свойств вызывается процедура `RB-Delete-Fixup(T, x)`. Узел x считается несущим «дополнительный черный цвет».

RB-Delete-Fixup(T, x) Эта процедура восстанавливает свойства КЧ-дерева после удаления черного узла. Цикл продолжается, пока x не является корнем и $x.color = BLACK$. Обозначим $P = x.parent$ и w — брат узла x.

Сценарий: Узел x является левым ребенком своего родителя P.

- **Случай 1: Брат w узла x красный.**
  - $w.color \leftarrow BLACK$
  - $P.color \leftarrow RED$
  - `Left-Rotate(T, P)`
  - $w \leftarrow P.right$ (Обновляем w на нового брата x, который теперь будет черным)

  Логика: Это преобразование приводит к одному из следующих случаев (2, 3 или 4), где брат x уже черный. «Дополнительный черный» на x сохраняется.

- **Случай 2: Брат w узла x черный, и оба ребенка w черные.**
  - $w.color \leftarrow RED$
  - $x \leftarrow P$ (Перемещаем «дополнительный черный» вверх к родителю P. Цикл продолжается для нового $x = P$).

  Логика: Черный цвет «снимается» с x и w и «добавляется» к P. Пути через w теперь имеют на один черный узел меньше (сам w), что компенсирует удаленный черный узел на путях через x.

- **Случай 3: Брат w узла x черный, левый ребенок w красный, а правый ребенок w черный.**
  - $w.left.color \leftarrow BLACK$
  - $w.color \leftarrow RED$
  - `Right-Rotate(T, w)`
  - $w \leftarrow P.right$ (Обновляем w на нового брата x)

  Логика: Это преобразование приводит к Случаю 4. «Дополнительный черный» на x сохраняется.

- **Случай 4: Брат w узла x черный, и правый ребенок w красный.**
  - $w.color \leftarrow P.color$
  - $P.color \leftarrow BLACK$
  - $w.right.color \leftarrow BLACK$
  - `Left-Rotate(T, P)`
  - $x \leftarrow T.root$ (Устанавливаем x на корень, чтобы завершить цикл. «Дополнительный черный» устранен).

  Логика: Это преобразование полностью устраняет «дополнительный черный». Цвета перераспределяются, и выполняется вращение для восстановления черных высот.

После завершения цикла, если x не NIL, его цвет устанавливается в черный ($x.color \leftarrow BLACK$) для удаления любого оставшегося «дополнительного черного» и обеспечения свойства корня.

1.3 Алгоритм сжатия RLE (Run-Length Encoding)

Алгоритм RLE — это простой метод сжатия данных без потерь, который эффективен для данных, содержащих много последовательностей повторяющихся значений.

Основной принцип RLE: Заменить последовательность одинаковых символов на пару, состоящую из самого символа и количества его повторений.

Продвинутый RLE: Для повышения эффективности на данных, где длинные серии повторов редки, используется модифицированный подход, который различает два типа последовательностей:

1. **Серия повторов (Run):** Если символ c повторяется N раз, и $N \geq$ MIN_RUN_LENGTH, то эта серия кодируется как пара: счетчик N и сам символ c.

2. **Литеральная последовательность (Literal Run):** Если последовательность символов S_{lit} длиной M не содержит серий повторов длиной MIN_RUN_LENGTH или более, то она кодируется как: отрицательное значение длины $-M$, за которым следует специальный символ-разделитель '#', и затем сама последовательность S_{lit}.

Такой подход позволяет избежать увеличения размера данных для коротких неповторяющихся последовательностей, что характерно для простого RLE. Использование символа-разделителя '#' между счетчиком/длиной и данными помогает декодеру однозначно парсить закодированный поток, особенно когда сами данные могут содержать цифры.

Коэффициент сжатия (CR): Определяется как отношение размера исходных данных $Size_{orig}$ к размеру сжатых данных $Size_{comp}$:

$$CR = \frac{Size_{orig}}{Size_{comp}}$$

Если $CR > 1$, данные сжаты. Если $CR < 1$, размер данных увеличился. Если $CR = 1$, размер не изменился.

Цена кодирования: Для RLE "цена кодирования"не является статистической величиной, как средняя длина кодового слова в энтропийных кодах. Фактически, это размер сжатых данных $Size_{comp}$. Он полностью зависит от структуры входных данных.

1.4 Математическое и алгоритмическое описание кодирования Фано

Алгоритм Фано предназначен для построения префиксных кодов переменной длины для символов источника с известными вероятностями их появления. Целью является минимизация средней длины кодового слова.

Пусть дан алфавит источника $S = \{s_1, s_2, \ldots, s_N\}$ и соответствующие им вероятности $P = \{p_1, p_2, \ldots, p_N\}$, где p_i – вероятность символа s_i, и $\sum_{i=1}^{N} p_i = 1$.

Процесс построения кодов Фано включает следующие шаги:

1. **Упорядочивание символов.** Все символы $s_i \in S$ сортируются в порядке невозрастания их вероятностей p_i: $p_1 \geq p_2 \geq \cdots \geq p_N$.

2. **Рекурсивное разделение.** Процесс построения кодов рекурсивен. Для текущего (отсортированного) списка символов $S_{sub} = \{s_j, s_{j+1}, \ldots, s_k\}$:

   (a) Если список S_{sub} содержит только один символ ($j = k$), то процесс для этой ветви завершен. Накопленная последовательность битов является кодом для этого символа.

   (b) Если список S_{sub} содержит более одного символа, он разделяется на два непустых подсписка $S_{sub}^{(0)} = \{s_j, \ldots, s_m\}$ и $S_{sub}^{(1)} = \{s_{m+1}, \ldots, s_k\}$ ($j \leq m < k$). Разделение производится таким образом, чтобы сумма вероятностей символов в $S_{sub}^{(0)}$ была как можно ближе к сумме вероятностей символов в $S_{sub}^{(1)}$. То есть, ищется такой индекс m, который минимизирует величину:

   $$\left| \sum_{l=j}^{m} p_l - \sum_{l=m+1}^{k} p_l \right|$$

   Или, что эквивалентно, $\sum_{l=j}^{m} p_l$ должна быть максимально близка к $\frac{1}{2} \sum_{l=j}^{k} p_l$.

   (c) Всем символам из подсписка $S_{sub}^{(0)}$ к их текущему кодовому префиксу дописывается бит '0'.

   (d) Всем символам из подсписка $S_{sub}^{(1)}$ к их текущему кодовому префиксу дописывается бит '1'.

   (e) Процесс рекурсивно повторяется для подсписков $S_{sub}^{(0)}$ и $S_{sub}^{(1)}$.

Начальный вызов рекурсивной процедуры происходит для полного отсортированного списка символов S с пустым начальным префиксом кода.

Свойства кодов: Коды, полученные методом Фано, являются префиксными, что обеспечивает однозначность декодирования. Однако они не всегда являются оптимальными с точки зрения минимизации средней длины кодового слова. Средняя длина кодового слова L_{avg} для кодов Фано удовлетворяет неравенству Шеннона: $H(S) \leq L_{avg} < H(S) + 1$, где $H(S)$ – энтропия источника.

Декодирование: Декодирование осуществляется путем последовательного чтения битов из сжатого потока. Накопленная битовая последовательность сравнивается с префиксными кодами из построенной таблицы. При обнаружении совпадения с одним из кодов, соответствующий символ восстанавливается, и процесс продолжается для оставшейся части битового потока.

2 Реализация словаря на основе хеш-таблицы

2.1 Структура узла HashNode

```
namespace DictionaryWithHashTable {
struct HashNode {
    std::string key;
    int value;
    HashNode(std::string k, int v) : key(std::move(k)), value(v) {}
}; }
```

Листинг 1: Структура HashNode

Описание членов структуры HashNode:

- key: Тип std::string. Хранит ключ элемента хеш-таблицы, а именно – слово из текста.
- value: Тип int. Хранит значение, ассоциированное с ключом, а именно – частоту встречаемости слова.

2.2 Класс HashTable

Хеш-таблица была реализована с использованием метода цепочек для разрешения коллизий. Основным хранилищем является вектор списков (‘std::vector<std::list<HashNode»‘), где каждый элемент вектора (корзина) содержит список узлов с одинаковым хеш-значением.

```
class HashTable {
private:
    std::vector<std::list<HashNode>> table;
    size_t num_elements;
    size_t table_size;
    static constexpr double MAX_LOAD_FACTOR = 0.75;
    size_t hashFunction(const std::string& key) const;
    void rehash();
public:
    HashTable(size_t initial_size = 101) : table_size(initial_size), num_elements(0);
    void add(const std::string& key, int value);
    int* get(const std::string& key);
    const int* get(const std::string& key) const;
    bool remove(const std::string& key);
    void clear();
    void print(std::ostream& os = std::cout) const;
    void visualize(std::ostream& os = std::cout) const;
};
```

Листинг 2: Класс HashTable

Описание членов класса `HashTable`:

- `table`: Тип `std::vector<std::list<HashNode>>`. Основное хранилище данных, массив корзин, где каждая корзина - список узлов.
- `num_elements`: Тип `size_t`. Текущее количество элементов (пар ключ-значение) в таблице.
- `table_size`: Тип `size_t`. Текущий размер таблицы.
- `MAX_LOAD_FACTOR`: Статическая константа типа `double`, равная 0.75. Пороговое значение коэффициента загрузки, при превышении которого происходит рехеширование.

2.3 Хеш-функция

- **Вход:** `key` – ключ.
- **Выход:** Значение типа `size_t`, представляющее собой хеш-код (индекс в хеш-таблице) для входного ключа.
- **Описание:** Данная функция реализует полиномиальный алгоритм хеширования для строковых ключей. Она итеративно обрабатывает каждый байт входной строки `key`.

```
size_t hashFunction(const std::string& key) const {
    size_t hash_val = 0;
    size_t p = 31;
    size_t p_pow = 1;
    for (char c_byte : key) {
        hash_val = (hash_val + static_cast<unsigned char>(c_byte) * p_pow) %
            table_size;
        p_pow = (p_pow * p) % table_size;
    }
    return hash_val;
}
```

Листинг 3: Хеш-функция

2.4 Функция `add(const std::string& key, int value)`

- **Вход:** `key` – ключ (слово), `value` – значение (частота).
- **Выход:** занесенная пара ключ-значение в таблицу.
- **Описание:** Добавляет пару ключ-значение в таблицу. Если ключ уже существует, обновляет его значение. При необходимости (превышение `MAX_LOAD_FACTOR` при добавлении нового элемента) выполняет рехеширование.

```cpp
void add(const std::string& key, int value) {
    if (static_cast<double>(num_elements + 1) / table_size >= MAX_LOAD_FACTOR) {
        rehash();
    }

    size_t index = hashFunction(key);
    for (auto& node : table[index]) {
        if (node.key == key) {
            node.value = value;
            return;
        }
    }
    table[index].emplace_back(key, value);
    num_elements++;
}
```

Листинг 4: Функция add в HashTable

2.5 Функция `get(const std::string& key)`

- **Вход:** `key` – ключ для поиска.
- **Выход:** Указатель на значение (`int*`), если ключ найден; `nullptr` в противном случае.
- **Описание:** Ищет элемент по ключу и возвращает указатель на его значение.

```cpp
int* get(const std::string& key) {
    size_t index = hashFunction(key);
    for (auto& node : table[index]) {
        if (node.key == key) {
            return &node.value;
        }
    }
    return nullptr;
}
```

Листинг 5: Функция get в HashTable

2.6 Функция `remove(const std::string& key)`

- **Вход:** `key` – ключ удаляемого элемента.
- **Выход:** `true`, если элемент удален; `false`, если элемент не найден.
- **Описание:** Удаляет элемент с заданным ключом из таблицы.

```cpp
bool remove(const std::string& key) {
    size_t index = hashFunction(key);
    auto& bucket = table[index];
    for (auto it = bucket.begin(); it != bucket.end(); ++it) {
        if (it->key == key) {
            bucket.erase(it);
            num_elements--;
            return true;
        }
    }
    return false;
}
```

Листинг 6: Функция remove в HashTable

2.7 Функция `clear()`

- **Вход:** Текущая хеш-таблица.
- **Выход:** Очищенная хеш-таблица.
- **Описание:** Удаляет все элементы из хеш-таблицы, очищая все корзины и сбрасывая счетчик элементов.

```cpp
void clear() {
    for (auto& bucket : table) {
        bucket.clear();
    }
    num_elements = 0;
}
```

Листинг 7: Функция clear в HashTable

2.8 Приватная функция `rehash()`

- **Вход:** Текущая хеш-таблица.
- **Выход:** Увеличенная хеш-таблица.
- **Описание:** Увеличивает размер таблицы (`table_size`), создает новую таблицу и переносит в нее все существующие элементы с пересчетом их хеш-значений.

```
void rehash() {
    size_t old_table_size = table_size;
    table_size = table_size * 2 + 1;
    std::vector<std::list<HashNode>> old_table = std::move(table);

    table.assign(table_size, std::list<HashNode>());
    num_elements = 0;
    for (size_t i = 0; i < old_table_size; ++i) {
        for (const auto& node : old_table[i]) {
            add(node.key, node.value);
        }
    }
}
```

Листинг 8: Функция rehash в HashTable

2.9 Публичная функция `print`

Прототип: `void print(std::ostream& os = std::cout) const`

- **Вход:** `os`: `std::ostream&` – ссылка на выходной поток (по умолчанию `std::cout`).
- **Выход:** вывод осуществляется в поток `os`.
- **Описание:** Выводит все элементы хеш-таблицы в указанный поток. Итерирует по каждой корзине таблицы и по каждому узлу в непустых корзинах. Пары ключ-значение выводятся в формате `'ключ': значение`, разделенные запятыми, и всё содержимое заключается в фигурные скобки.

```
void print(std::ostream& os = std::cout) const {
    os << "{";
    bool first_item = true;
    for (const auto& bucket : table) {
        for (const auto& node : bucket) {
            if (!first_item) {
                os << ", ";
            }
            os << "'" << node.key << "': " << node.value;
            first_item = false;
        }
    }
    os << "} ";
}
```

Листинг 9: Функция print в HashTable

2.10 Публичная функция `visualize`

Прототип: `void visualize(std::ostream& os = std::cout) const`

- **Вход:** `os`: `std::ostream&` – ссылка на выходной поток (по умолчанию `std::cout`).
- **Выход:** осуществляется вывод в поток `os`.
- **Описание:** Выводит детальное представление хеш-таблицы. Для каждой корзины отображается ее номер и содержимое. Если корзина пуста, выводится `<пусто>`. Если в корзине есть элементы (цепочка коллизий), они выводятся последовательно в формате `("ключ": значение)`, разделенные символом `->`. Также выводится общий размер таблицы и количество элементов.

```
void visualize(std::ostream& os = std::cout) const {
    os << "Визуализация Хеш-таблицы (размер: " << table_size
       << ", элементы: " << num_elements << "):" << std::endl;
    for (size_t i = 0; i < table.size(); ++i) {
        os << "Корзина [" << i << "]: ";
        if (table[i].empty()) {
            os << "<пусто>";
        } else {
            bool first_in_bucket = true;
            for (const auto& node : table[i]) {
                if (!first_in_bucket) {
                    os << " -> ";
                }
                os << "(\"" << node.key << "\": " << node.value << ")";
                first_in_bucket = false;
            }
        }
        os << std::endl;
    }
}
```

Листинг 10: Функция visualize в HashTable

2.11 Класс `Dictionary`

Класс `Dictionary` инкапсулирует объект `HashTable` и предоставляет интерфейс для работы со словарем.

```
class Dictionary {
private:
    HashTable ht;
    std::string toLowerASCII(std::string s) const;

public:
    Dictionary(size_t initial_capacity = 101) : ht(initial_capacity) {}

    void addWord(const std::string& word_raw);
    void removeWord(const std::string& word_raw);
    bool findWord(const std::string& word_raw) const;
    void clear();
    void loadFromFile(const std::string& filepath, bool append = false);
    void print(std::ostream& os = std::cout) const;
    void visualizeStructure(std::ostream& os = std::cout) const;
};
```

Листинг 11: Функция print в HashTable

2.12 Функция `addWord(const std::string& word_raw)`

- **Вход:** `word_raw` – слово для добавления или обновления частоты.
- **Выход:** слово добавлено в словарь.
- **Описание:** Приводит слово к нижнему регистру (ASCII-символы). Если слово уже есть в словаре, увеличивает его частоту. В противном случае добавляет слово с частотой 1.

```
void addWord(const std::string& word_raw) {
    if (word_raw.empty()) return;
    std::string word = toLowerASCII(word_raw);

    int* current_val_ptr = ht.get(word);
    if (current_val_ptr) {
        (*current_val_ptr)++;
    } else {
        ht.add(word, 1);
    }
}
```

Листинг 12: Функция addWord в Dictionary (HashTable)

2.13 Функция `removeWord(const std::string& word_raw)`

- **Вход:** `word_raw` – слово для удаления из словаря.
- **Выход:** слово удалено из словаря.
- **Описание:** Приводит слово к нижнему регистру (ASCII) и удаляет его из внутренней хеш-таблицы.

```
void removeWord(const std::string& word_raw) {
    if (word_raw.empty()) return;
    std::string word = toLowerASCII(word_raw);
    ht.remove(word);
}
```

Листинг 13: Функция removeWord в Dictionary (HashTable)

2.14 Функция `findWord(const std::string& word_raw) const`

- **Вход:** `word_raw` – слово для поиска.
- **Выход:** `true`, если слово найдено; `false` в противном случае. Также выводит информацию о частоте слова, если оно найдено.
- **Описание:** Приводит слово к нижнему регистру (ASCII), ищет его в хеш-таблице и сообщает о результате.

```
bool findWord(const std::string& word_raw) const {
    if (word_raw.empty()) return false;
    std::string word = toLowerASCII(word_raw);

    const int* count_ptr = ht.get(word);
    if (count_ptr) {
        std::cout << "Слово '" << word_raw << "' (ключ: '" << word
                  << "') найдено, частота: " << *count_ptr << std::endl;
        return true;
    } else {
        std::cout << "Слово '" << word_raw << "' (ключ: '" << word
                  << "') не найдено." << std::endl;
        return false;
    }
}
```

Листинг 14: Функция findWord в Dictionary (HashTable)

2.15 Функция `loadFromFile(const std::string& filepath, bool append = false)`

- **Вход:** `filepath` – путь к текстовому файлу, `append` (опционально, по умолчанию `false`) – флаг, указывающий, нужно ли добавлять слова к существующему словарю или очистить его перед загрузкой.
- **Выход:** текст загружен из файла.
- **Описание:** Читает текстовый файл, разбивает его на слова и добавляет их в словарь с подсчетом частот. Использует глобальные функции `readFileToString` и `processTextToWords`.

```
void loadFromFile(const std::string& filepath, bool append = false) {
    if (!append) {
        clear();
    }
    try {
        std::string content = readFileToString(filepath);
        std::vector<std::string> words = processTextToWords(content);
        for (const std::string& word : words) {
            if (!word.empty()) addWord(word);
        }
        std::cout << "Словарь загружен/дополнен из файла '" << filepath
                  << "' (хеш-таблица)." << std::endl;
    } catch (const std::runtime_error& e) {
        std::cerr << "Ошибка при загрузке из файла (хеш-таблица): "
                  << e.what() << std::endl;
    }
}
```

Листинг 15: Функция loadFromFile в Dictionary (HashTable)

3 Реализация словаря на основе Красно-Черного дерева

Для альтернативной реализации словаря было использовано Красно-Черное дерево (КЧ-дерево) — самобалансирующееся бинарное дерево поиска. Это обеспечивает гарантированную логарифмическую сложность O(log N) для операций вставки, удаления и поиска в худшем случае.

3.1 Структура данных: Красно-Черное дерево

```
namespace DictionaryWithRBTree {

enum Color { RED, BLACK };

struct Node {
    std::string key;
    int value;
    Color color;
    Node *parent, *left, *right;

    Node(std::string k, int v, Color c = RED, Node* p = nullptr, Node* l = nullptr,
        Node* r = nullptr)
        : key(std::move(k)), value(v), color(c), parent(p), left(l), right(r) {}
};
```

Листинг 16: Перечисление Color

Описание членов структуры `Node`:

- `key`: Тип `std::string`. Хранит ключ (слово).
- `value`: Тип `int`. Хранит значение (частоту слова).
- `color`: Тип `Color`. Хранит цвет узла.
- `parent`: Указатель на родительский узел `Node*`.
- `left`: Указатель на левого дочернего узла `Node*`.
- `right`: Указатель на правого дочернего узла `Node*`.

3.2 Класс `RBTree`

```
class RBTree {
private:
    Node* root;
    Node* NIL;
```

```
5       void leftRotate(Node* x);
6       void rightRotate(Node* y);
7       void insertFixup(Node* z);
8       Node* findNode(const std::string& key) const;
9       void transplant(Node* u, Node* v);
10      Node* minimum(Node* node);
11      void deleteFixup(Node* x);
12      void destroyRecursive(Node* node) ;
13      void inorderPrintRecursive(Node* node, std::ostream& os, bool& first_item) const;
14      bool printGivenLevel(Node* node, int level, std::ostream& os, bool first_on_level
            ) const;
15      int getMaxDepth(Node* node) const;
16      void printTreeRecursive(Node* node, int space_increment, int current_space, std::
            ostream& os) const ;
17
18  public:
19      RBTree();
20      ~RBTree() {
21          destroyRecursive(root);
22          delete NIL;
23      }
24      RBTree(const RBTree&) = delete;
25      RBTree& operator=(const RBTree&) = delete;
26      void insert(const std::string& key, int value);
27      int* search(const std::string& key);
28      const int* search(const std::string& key) const;
29      bool remove(const std::string& key);
30      void clear();
31      void print(std::ostream& os = std::cout) const;
32      void visualize(std::ostream& os = std::cout) const;
33  };
```

Листинг 17: Перечисление Color

Описание членов класса `RBTree`:

- `root`: Указатель `Node*` на корень дерева.
- `NIL`: Указатель `Node*` на специальный стражевой (sentinel) NIL-узел. Все "отсутствующие"листья и родитель корня указывают на этот узел. Он всегда черный.

3.3 Конструктор `RBTree()`

- **Вход:** пустой объект `RBTree`.
- **Выход:** создается объект `RBTree`.

- **Описание:** Инициализирует дерево: создает стражевой `NIL`-узел (черного цвета, все указатели которого ссылаются на него же) и устанавливает `root` на этот `NIL`-узел, обозначая пустое дерево.

```
RBTree() {
    NIL = new Node("", 0, BLACK);
    NIL->parent = NIL;
    NIL->left = NIL;
    NIL->right = NIL;
    root = NIL;
}
```

Листинг 18: Конструктор RBTree

3.4 Функция `insert(const std::string& key, int value)`

- **Вход:** `key` – ключ (слово), `value` – значение (частота).
- **Выход:** новый узел.
- **Описание:** Вставляет новый узел с заданным ключом и значением в дерево. Новый узел изначально красный. Если ключ уже существует, обновляет его значение. После вставки вызывается процедура `insertFixup` для восстановления свойств КЧ-дерева.

```
void insert(const std::string& key, int value) {
        Node* z = new Node(key, value, RED, NIL, NIL, NIL);
        Node* y = NIL;
        Node* x = root;

        while (x != NIL) {
            y = x;
            if (z->key < x->key) {
                x = x->left;
            } else if (z->key > x->key) {
                x = x->right;
            } else {
                x->value = value;
                delete z;
                return;
            }
        }

        z->parent = y;
        if (y == NIL) {
```

```
            root = z;
        } else if (z->key < y->key) {
            y->left = z;
        } else {
            y->right = z;
        }
        insertFixup(z);
    }
```

Листинг 19: Функция insert в RBTree

3.5 Функция `search(const std::string& key)`

- **Вход:** `key` – ключ для поиска.
- **Выход:** Указатель на значение (`int*`), если ключ найден; `nullptr` в противном случае.
- **Описание:** Выполняет поиск узла по ключу, как в стандартном бинарном дереве поиска.

```
int* search(const std::string& key) {
    Node* node = findNode(key);
    return (node == NIL) ? nullptr : &node->value;
}
const int* search(const std::string& key) const {
    Node* node = findNode(key);
    return (node == NIL) ? nullptr : &node->value;
}
```

Листинг 20: Функция search в RBTree

3.6 Функция `remove(const std::string& key)`

- **Вход:** `key` – ключ удаляемого элемента.
- **Выход:** `true`, если элемент удален; `false`, если элемент не найден.
- **Описание:** Удаляет узел с заданным ключом из дерева. После физического удаления узла (или замены его преемником) вызывается процедура `deleteFixup`, если был удален черный узел, для восстановления свойств КЧ-дерева.

```
bool remove(const std::string& key) {
        Node* z = findNode(key);
        if (z == NIL) return false;

        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) {
                if (x != NIL) x->parent = y;
            } else {
                transplant(y, y->right);
                y->right = z->right;
                y->right->parent = y;
            }
            transplant(z, y);
            y->left = z->left;
            y->left->parent = y;
            y->color = z->color;
        }
        delete z;

        if (y_original_color == BLACK) {
            deleteFixup(x);
        }
        return true;
    }
```

Листинг 21: Функция remove в RBTree

3.7 Приватные функции `leftRotate` `rightRotate`

- **Вход:** `x`: `Node*` – указатель на узел, относительно которого выполняется вращение.
- **Выход:** модифицированная структура дерева.
- **Описание:** Осуществляет левое вращение дерева вокруг узла `x`. Правый дочерний узел `y` узла `x` становится новым корнем данного поддерева. Узел `x` становится левым дочерним узлом `y`. Левое поддерево `y` становится правым поддеревом `x`. Функция корректно обновляет

все указатели (`parent`, `left`, `right`) затронутых узлов и, при необходимости, указатель `root` всего дерева.

```
    void 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;
    }

    void 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;
    }
```

Листинг 22: Функция leftRotate в RBTree

3.8 Приватная функция `insertFixup`

- **Вход:** `z`: `Node*` – указатель на вставленный узел (изначально красный).

- **Выход:** модифицированная структура дерева.
- **Описание:** Восстанавливает свойства красно-черного дерева после вставки узла `z`. Если родитель `z` также красный (нарушение), функция итеративно применяет вращения и перекрашивания, рассматривая три основных случая, основанных на цвете "дяди"узла `z` и конфигурации узлов (родитель `z` как левый/правый ребенок "дедушки"`z`; `z` как левый/-правый ребенок своего родителя). Цель — устранить последовательность из двух красных узлов и сохранить черную высоту. В конце корень дерева всегда окрашивается в черный.

```
void 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);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                rightRotate(z->parent->parent);
            }
        } else {
            Node* y = z->parent->parent->left;
            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->left) {
                    z = z->parent;
                    rightRotate(z);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                leftRotate(z->parent->parent);
            }
        }
```

```
            if (z == root) break;
        }
        root->color = BLACK;
}
```

Листинг 23: Функция insertFixup в RBTree

3.9 Приватная функция `findNode`

- **Вход:** `key`: `const std::string&` – ключ (слово) для поиска.
- **Выход:** `Node*` – указатель на найденный узел, либо на `NIL`, если узел не найден.
- **Описание:** Осуществляет поиск узла с заданным ключом в красно-черном дереве. Поиск выполняется аналогично стандартному бинарному дереву поиска: начиная с корня, происходит сравнение ключей и переход в левое или правое поддерево до тех пор, пока узел не будет найден или не будет достигнут `NIL`-узел.

```
Node* findNode(const std::string& key) const {
    Node* current = root;
    while (current != NIL && current->key != key) {
        if (key < current->key) {
            current = current->left;
        } else {
            current = current->right;
        }
    }
    return current;
}
```

Листинг 24: Функция findNode в RBTree

3.10 Приватная функция `transplant`

Прототип: `void transplant(Node* u, Node* v)`

- **Вход:** `u`: `Node*` – указатель на узел, который заменяется и `v`: `Node*` – указатель на узел, который ставится на место `u` (может быть `NIL`).
- **Выход:** модифицированная структура дерева.
- **Описание:** Вспомогательная функция для операции удаления. Заменяет поддерево с корнем `u` поддеревом с корнем `v`. Обновляет соответствующий дочерний указатель у родителя узла `u` (или указатель `root`, если `u` был корнем) так, чтобы он указывал на `v`. Если `v` не `NIL`, его указатель на родителя устанавливается на бывшего родителя `u`.

```
void transplant(Node* u, Node* v) {
    if (u->parent == NIL) {
        root = v;
    } else if (u == u->parent->left) {
        u->parent->left = v;
    } else {
        u->parent->right = v;
    }
    v->parent = u->parent;
}
```

Листинг 25: Функция transplant в RBTree

3.11 Приватная функция `deleteFixup`

Прототип: `void deleteFixup(Node* x)`

- **Вход:** `x`: `Node*` – узел, который условно несет "дополнительный черный"цвет после удаления другого черного узла.
- **Выход:** модифицирует цвета и структуру дерева.
- **Описание:** Восстанавливает свойства красно-черного дерева после удаления черного узла. Функция итеративно обрабатывает узел `x`, который считается "дважды черным для устранения нарушения черной высоты. Рассматриваются четыре случая (и их симметричные варианты) в зависимости от цвета брата `x` и его детей, применяя вращения и перекрашивания для перераспределения черных узлов или "проталкивания"проблемы вверх по дереву. Цикл завершается, когда "дополнительный черный"цвет устранен или достигнут корень. Корень дерева в конце всегда окрашивается в черный.

```
void deleteFixup(Node* x) {
    while (x != root && x->color == BLACK) {
        if (x == x->parent->left) {
            Node* w = x->parent->right;
            if (w->color == RED) {
                w->color = BLACK;
                x->parent->color = RED;
                leftRotate(x->parent);
                w = x->parent->right;
            }
            if (w->left->color == BLACK && w->right->color == BLACK) {
                w->color = RED;
                x = x->parent;
```

```
            } else {
                if (w->right->color == BLACK) {
                    w->left->color = BLACK;
                    w->color = RED;
                    rightRotate(w);
                    w = x->parent->right;
                }
                w->color = x->parent->color;
                x->parent->color = BLACK;
                w->right->color = BLACK;
                leftRotate(x->parent);
                x = root;
            }
        } else {
            Node* w = x->parent->left;
            if (w->color == RED) {
                w->color = BLACK;
                x->parent->color = RED;
                rightRotate(x->parent);
                w = x->parent->left;
            }
            if (w->right->color == BLACK && w->left->color == BLACK) {
                w->color = RED;
                x = x->parent;
            } else {
                if (w->left->color == BLACK) {
                    w->right->color = BLACK;
                    w->color = RED;
                    leftRotate(w);
                    w = x->parent->left;
                }
                w->color = x->parent->color;
                x->parent->color = BLACK;
                w->left->color = BLACK;
                rightRotate(x->parent);
                x = root;
            }
        }
    }
    x->color = BLACK;
}
```

Листинг 26: Функция deleteFixup в RBTree

3.12 Приватная функция `destroyRecursive`

Прототип: `void destroyRecursive(Node* node)`

- **Вход:** `node`: `Node*` – указатель на корень поддерева, которое нужно удалить.
- **Выход:** освобождает память.
- **Описание:** Рекурсивно удаляет все узлы в поддереве, корнем которого является `node`. Использует обратный обход (post-order): сначала рекурсивно обрабатывает левое и правое поддеревья, а затем удаляет сам узел `node`. Базовый случай рекурсии – достижение `NIL`-узла. Эта функция используется для полной очистки дерева в методе `clear()` и в деструкторе.

```
1 void destroyRecursive(Node* node) {
2     if (node != NIL) {
3         destroyRecursive(node->left);
4         destroyRecursive(node->right);
5         delete node;
6     }
7 }
```

Листинг 27: Функция destroyRecursive в RBTree

3.13 Приватная функция `inorderPrintRecursive`

Прототип: `void inorderPrintRecursive(Node* node, std::ostream& os, bool& first_item) const`

- **Вход:**
  - `node`: `Node*` – текущий узел для обработки.
  - `os`: `std::ostream&` – поток для вывода.
  - `first_item`: `bool&` – флаг, указывающий, является ли текущий выводимый элемент первым в общем списке (для корректной расстановки запятых).
- **Выход:** осуществляется вывод в поток `os`).
- **Описание:** Рекурсивно обходит дерево в инфиксном (симметричном) порядке и выводит информацию о каждом узле (ключ, значение, цвет) в поток `os`. Благодаря инфиксному обходу бинарного дерева поиска, ключи выводятся в лексикографическом (отсортированном) порядке. Флаг `first_item` используется для корректного форматирования вывода с запятыми-разделителями.

```
void inorderPrintRecursive(Node* node, std::ostream& os, bool& first_item) const {
    if (node != NIL) {
        inorderPrintRecursive(node->left, os, first_item);
        if (!first_item) {
            os << ", ";
        }
        os << "'" << node->key << "': " << node->value << " (" << (node->color == RED
            ? "R" : "B") << ")";
        first_item = false;
        inorderPrintRecursive(node->right, os, first_item);
    }
}
```

Листинг 28: Функция inorderPrintRecursive в RBTree

3.14 Приватная функция `getMaxDepth`

Прототип: `int getMaxDepth(Node* node) const`

- **Вход:** `node`: `Node*` – указатель на корень поддерева, для которого вычисляется глубина.
- **Выход:** `int` – максимальная глубина поддерева (количество узлов на самом длинном пути от `node` до листового `NIL`-узла).
- **Описание:** Рекурсивно вычисляет максимальную глубину поддерева с корнем `node`. Для `NIL`-узла глубина равна 0. Для остальных узлов глубина вычисляется как $1 + \max$ (глубина левого поддерева, глубина правого поддерева). Эта функция используется для определения количества уровней при визуализации дерева.

```
int getMaxDepth(Node* node) const {
    if (node == NIL) return 0;
    int left_depth = getMaxDepth(node->left);
    int right_depth = getMaxDepth(node->right);
    return std::max(left_depth, right_depth) + 1;
}
```

Листинг 29: Функция getMaxDepth в RBTree

3.15 Класс `Dictionary` (на основе `RBTree`)

Аналогичен словарю на хеш-таблице, но использует `RBTree` для хранения данных.

```
class Dictionary {
private:
```

```
    RBTree rbt;
    std::string toLowerASCII(std::string s) const {
        std::transform(s.begin(), s.end(), s.begin(),
                       [](unsigned char c){ return std::tolower(c); });
        return s;
    }

public:
    Dictionary() = default;
    void addWord(const std::string& word_raw);
    void removeWord(const std::string& word_raw);
    bool findWord(const std::string& word_raw) const;
    void clear();
    void loadFromFile(const std::string& filepath, bool append = false);
    void print(std::ostream& os = std::cout) const;
    void visualizeStructure(std::ostream& os = std::cout) const;
};

}
```

Листинг 30: Класс `Dictionary` (на основе `RBTree`)

Основные функции класса `Dictionary` Функции `Dictionary()` (конструктор), `addWord()`, `removeWord()`, `findWord()`, `clear()`, `loadFromFile()` имеют то же назначение, входные и выходные параметры, что и в версии словаря на хеш-таблице, но для операций со словами они обращаются к соответствующим методам объекта `rbt`.

- **addWord**: вызывает `rbt.search()` и затем `rbt.insert()`.
- **removeWord**: вызывает `rbt.remove()`.
- **findWord**: вызывает `rbt.search()`.
- **clear**: вызывает `rbt.clear()`.
- **loadFromFile**: аналогично версии для хеш-таблицы, но использует `addWord` данного класса.

Код этих функций по структуре очень похож на версию для хеш-таблицы, с заменой вызовов `ht` на `rbt`.

4 Алгоритм сжатия RLE

4.1 Функция генерации случайного текста

- **Вход:** `filename`: `const std::string&` – имя файла для сохранения.
- **Выход:** `std::string` – сгенерированный текст.
- **Описание:** Генерирует псевдослучайный текст, имитирующий заданное распределение символов русского алфавита, цифр и спецсимволов. Для задания распределения используется весовой подход: частым символам (например, 'о', 'е', 'а', пробел) присваиваются большие веса. Выбор символа осуществляется с помощью `std::discrete_distribution`. С определенной вероятностью (в реализации 20%) функция генерирует серии повторяющихся символов (длиной от 3 до 10), что позволяет создавать данные, подходящие для тестирования RLE-сжатия. Сгенерированный текст сохраняется в файл и возвращается функцией.

```
const double CHAMPER_A = 1.57;
const double CHAMPER_M = 4.0;

const int MIN_RUN_LENGTH = 3;

double champernownePDF(double x){
    return CHAMPER_A / (M_PI * std::cosh(CHAMPER_A * (x - CHAMPER_M)));
}

std::string generateRandomText(size_t approx_target_chars, const std::string&
    filename = "random_text_custom_dist.txt") {
    std::vector<std::string> base_char_pool;
    std::vector<std::string> RUS_LOWER = {
        "а", "б", "в", "г", "д", "е", "ж", "з", "и", "й", "к", "л", "м",
        "н", "о", "п", "р", "с", "т", "у", "ф", "х", "ц", "ч", "ш", "щ",
        "ы", "э", "ю", "я"
    };
    std::vector<std::string> DIGITS = {"0","1","2","3","4","5","6","7","8","9"};
    std::vector<std::string> SPECIALS = {" ", ".", ",", "!", "?", "-", ":"};

    for (const auto& s : RUS_LOWER) base_char_pool.push_back(s);
    for (const auto& s : DIGITS) base_char_pool.push_back(s);
    for (const auto& s : SPECIALS) base_char_pool.push_back(s);

    std::vector<double> weights;
    weights.reserve(base_char_pool.size());
    double total_calculated_weight = 0.0;

```

```
    for (size_t i = 0; i < base_char_pool.size(); ++i) {
        double x_for_char = static_cast<double>(i);
        double weight = champernownePDF(x_for_char);
        weight = std::max(weight, 1e-9);
        weights.push_back(weight);
        total_calculated_weight += weight;
    }

    std::random_device rd;
    std::mt19937 rng(rd());
    std::discrete_distribution<size_t> char_dist(weights.begin(), weights.end());

    std::uniform_real_distribution<double> run_probability_dist(0.0, 1.0);
    double probability_of_run = 0.20;
    std::uniform_int_distribution<size_t> run_length_dist(MIN_RUN_LENGTH, 10);

    std::string result_text;
    result_text.reserve(approx_target_chars * 2);
    size_t current_char_count = 0;

    while (current_char_count < approx_target_chars) {
        size_t char_idx = char_dist(rng);
        const std::string& selected_char_utf8 = base_char_pool[char_idx];

        if (run_probability_dist(rng) < probability_of_run && (current_char_count +
            MIN_RUN_LENGTH < approx_target_chars) ) {
            size_t length_of_run = run_length_dist(rng);
            length_of_run = std::min(length_of_run, approx_target_chars -
                current_char_count);

            if (length_of_run >= MIN_RUN_LENGTH ){
                for (size_t k_run = 0; k_run < length_of_run; ++k_run) {
                    result_text += selected_char_utf8;
                }
                current_char_count += length_of_run;
            } else {
                 result_text += selected_char_utf8;
                 current_char_count++;
            }
        } else {
            result_text += selected_char_utf8;
            current_char_count++;
        }
    }
```

```
70
71      std::ofstream outfile(filename, std::ios::binary);
72      if (outfile.is_open()) {
73          outfile.write(result_text.data(), result_text.length());
74          outfile.close();
75          std::cout << "Сгенерирован файл '" << filename << "' (сгенерировано " <<
                current_char_count
76                    << " UTF-8 симв., размер файла " << result_text.length() << " байт)
                         с использованием champernownePDF." << std::endl;
77      } else {
78          std::cerr << "Не удалось создать файл для генерации: " << filename << std::
                endl;
79      }
80      return result_text;
81  }
```

Листинг 31: Функция generateRandomText в RLE (ключевые моменты)

4.2 Функция `RLE::advancedRleEncode`

- **Вход:** `input`: `const std::string&` – исходная строка байт для кодирования.
- **Выход:** `std::string` – RLE-закодированная строка.
- **Описание:** Реализует продвинутый алгоритм RLE. Функция последовательно обрабатывает входную строку. Если обнаруживается серия одинаковых байт длиной не менее `MIN_RUN_LENGTH`, она кодируется в формате `«счетчик>#<байт>"`. Если длинная серия отсутствует, функция определяет максимальную возможную последовательность литералов (байтов, не образующих длинных серий) и кодирует ее в формате `<длина>#<байты_литералов>"`. Символ '#' используется как разделитель между числовым префиксом и данными.

```
1   const int MIN_RUN_LENGTH = 3;
2
3   std::string advancedRleEncode(const std::string& input) {
4       if (input.empty()) return "";
5
6       std::stringstream encoded_ss;
7       size_t i = 0;
8       const size_t n = input.length();
9       const char SEPARATOR = '#';
10
11      while (i < n) {
12          char current_char = input[i];
13          size_t count = 1;
```

```
        size_t j = i + 1;
        while (j < n && input[j] == current_char) {
            count++;
            j++;
        }

        if (count >= MIN_RUN_LENGTH) {
            encoded_ss << std::to_string(count) << SEPARATOR << current_char;
            i = j;
        } else {
            size_t literal_start = i;
            size_t k = i;
            while(k < n) {
                char check_char_literal = input[k];
                size_t run_len_ahead = 0;
                size_t temp_k = k;
                while(temp_k < n && input[temp_k] == check_char_literal) {
                    run_len_ahead++;
                    temp_k++;
                }
                if (run_len_ahead >= MIN_RUN_LENGTH) {
                    break;
                }
                k += run_len_ahead;
            }
            size_t literal_length = k - literal_start;
            if (literal_length > 0) {
                encoded_ss << "-" << std::to_string(literal_length) << SEPARATOR;
                encoded_ss << input.substr(literal_start, literal_length);
            }
            i = k;
        }
    }
    return encoded_ss.str();
}
```

Листинг 32: Функция advancedRleEncode в RLE

4.3 Функция `RLE::advancedRleDecode`

- **Вход:** `encoded_input`: `const std::string&` – RLE-закодированная строка.
- **Выход:** `std::string` – раскодированная (исходная) строка.
- **Описание:** Выполняет декодирование строки, сжатой функцией `advancedRleEncode`. Ите-

рирует по закодированной строке, ожидая определенный формат: «счетчик>#<байт>" для серий повторов или <длина>#<байты_литералов>" для последовательностей литералов. Функция считывает управляющие символы ('-' или цифру), затем число (счетчик/длину), затем символ-разделитель '#'. После этого, в зависимости от типа сегмента, либо повторяет следующий байт указанное число раз, либо считывает указанное количество байт как литералы. Содержит проверки на корректность формата и наличие данных, выбрасывая исключения `std::runtime_error` при ошибках.

```
std::string advancedRleDecode(const std::string& encoded_input) {
    if (encoded_input.empty()) return "";

    std::stringstream decoded_ss;
    size_t i = 0;
    const size_t n = encoded_input.length();
    const char SEPARATOR = '#';

    while (i < n) {
        char first_byte_of_seq = encoded_input[i];
        bool is_negative_run = false;

        if (first_byte_of_seq == '-') {
            is_negative_run = true;
            i++;
            if (i >= n || !std::isdigit(static_cast<unsigned char>(encoded_input[i]))
                ) {
                 throw std::runtime_error("RLE Decode: '-' not followed by a digit or
                     EOF.");
            }
        } else if (!std::isdigit(static_cast<unsigned char>(first_byte_of_seq))) {
            throw std::runtime_error("RLE Decode: Sequence does not start with '-' or
                 digit. Found: '" + std::string(1,first_byte_of_seq) + "' at position
                 " + std::to_string(i));
        }

        std::string num_str;
        while (i < n && std::isdigit(static_cast<unsigned char>(encoded_input[i]))) {
            num_str += encoded_input[i];
            i++;
        }

        if (i >= n || encoded_input[i] != SEPARATOR) {
            throw std::runtime_error("RLE Decode: Missing separator '" + std::string
                (1, SEPARATOR) + "' after number at pos ~" + std::to_string(i) + " (
```

```
                num_str was " + num_str + ")");
        }
        i++;

        if (num_str.empty()) {
            throw std::runtime_error("RLE Decode: Failed to read number for length/
                count.");
        }

        int count_or_length = 0;
        try {
            count_or_length = std::stoi(num_str);
        } catch (const std::out_of_range& oor) {
            throw std::runtime_error("RLE Decode: Number '" + num_str + "' out of
                range for int.");
        } catch (const std::invalid_argument& ia) {
             throw std::runtime_error("RLE Decode: Number '" + num_str + "' is not a
                 valid integer.");
        }

        if (count_or_length <= 0) {
             throw std::runtime_error("RLE Decode: Invalid count/length (<=0): " +
                 std::to_string(count_or_length));
        }

        if (is_negative_run) {
            if (i + count_or_length > n) {
                throw std::runtime_error("RLE Decode: Not enough data for literal
                    sequence. Expected " + std::to_string(count_or_length) + ",
                    available " + std::to_string(n-i));
            }
            decoded_ss << encoded_input.substr(i, count_or_length);
            i += count_or_length;
        } else {
            if (i >= n) {
                 throw std::runtime_error("RLE Decode: Missing char_to_repeat after
                     count and separator.");
            }
            char char_to_repeat = encoded_input[i];
            i++;
            for (int k_rep = 0; k_rep < count_or_length; ++k_rep) {
                decoded_ss << char_to_repeat;
            }
        }
```

```
67      }
68      return decoded_ss.str();
69  }
```

Листинг 33: Функция advancedRleDecode в RLE

5 Реализация кодирования Фано

Для второго этапа сжатия (или как самостоятельный метод) был реализован алгоритм статистического кодирования Фано. Ниже приведено описание ключевых структур и функций.

5.1 Структура `FanoNode`

Прототип структуры:

```
1  namespace Fano {
2  struct FanoNode {
3      char symbol;
4      double probability;
5
6      FanoNode(char sym, double prob) : symbol(sym), probability(prob) {}
7  };
8  }
```

Листинг 34: Структура FanoNode

- **Члены структуры:**
  - `symbol`: Тип `char`. Хранит символ (байт) из обрабатываемого текста.
  - `probability`: Тип `double`. Хранит вычисленную вероятность (или нормализованную частоту) появления данного символа.
- **Конструктор:** `FanoNode(char sym, double prob)` инициализирует символ и его вероятность.
- **Назначение:** Данная структура используется для временного представления символов и их вероятностей, в частности, для их последующей сортировки и обработки в процессе построения кодов Шеннона-Фано.

5.2 Функция `generateFanoCodes`

- **Вход:**

- `nodes`: `std::vector<FanoNode*>&` – ссылка на вектор указателей на узлы `FanoNode`, отсортированные по убыванию вероятностей.
- `start`, `end`: `int` – индексы, определяющие текущий диапазон узлов в векторе `nodes` для обработки.
- `prefix`: `std::string` – текущий двоичный префикс кода, сформированный на предыдущих шагах рекурсии.
- `codes`: `std::map<char, std::string>&` – ссылка на карту, в которую будут записаны результирующие коды (символ → строка кода).

- **Выход:** модифицированная карта `codes`.
- **Описание:** Это основная рекурсивная функция для построения кодов Шеннона-Фано. Для заданного диапазона отсортированных символов она находит точку разделения `splitIndex`, которая делит диапазон на две подгруппы с суммарными вероятностями, максимально близкими друг к другу. Затем функция рекурсивно вызывается для первой подгруппы (добавляя '0' к префиксу) и для второй подгруппы (добавляя '1' к префиксу). Базовый случай рекурсии — когда в диапазоне остается один символ; тогда текущий префикс становится его кодом. Реализована обработка случая, когда все вероятности в подгруппе равны нулю, и защита от пустых диапазонов.

```
void generateFanoCodes(std::vector<FanoNode*>& nodes, int start, int end, std::string
     prefix, std::map<char, std::string>& codes) {
    if (start > end) {
        return;
    }
    if (start == end) {
        codes[nodes[start]->symbol] = (prefix.empty() && nodes.size() == 1) ? "0" :
            prefix;
        return;
    }

    double totalWeight = 0;
    for (int i = start; i <= end; ++i) {
        totalWeight += nodes[i]->probability;
    }

    if (totalWeight == 0) {
        for(int i = start; i <= end; ++i) {
            codes[nodes[i]->symbol] = prefix + ( (i-start) % 2 == 0 ? "0" : "1");
        }
        return;
```

```
    }

    double partialWeight = 0;
    int splitIndex = start;
    for (; splitIndex < end; ++splitIndex) {
        if (partialWeight + nodes[splitIndex]->probability > totalWeight / 2.0 &&
            partialWeight > 0) {
            splitIndex--;
            break;
        }
        partialWeight += nodes[splitIndex]->probability;
        if (partialWeight >= totalWeight / 2.0) {
            break;
        }
    }
    if (splitIndex < start && start < end ) splitIndex = start;
    generateFanoCodes(nodes, start, splitIndex, prefix + "0", codes);
    if (splitIndex + 1 <= end) {
         generateFanoCodes(nodes, splitIndex + 1, end, prefix + "1", codes);
    }
}
```

Листинг 35: Функция generateFanoCodes

5.3 Функция `buildFanoCodes`

- **Вход:** `text`: `const std::string&` – входная строка, для символов которой строятся коды.
- **Выход:** `std::map<char, std::string>` – карта "символ → код Фано".
- **Описание:** Функция оркестрирует процесс создания кодовой таблицы Шеннона-Фано. Она выполняет подсчет частот символов во входном тексте, преобразует их в вероятности, создает для каждого символа узел `FanoNode`, сортирует эти узлы по убыванию вероятностей, а затем вызывает рекурсивную функцию `generateFanoCodes` для построения самих кодов. В конце освобождается память, выделенная для узлов `FanoNode`.

```
std::map<char, std::string> buildFanoCodes(const std::string& text) {
    std::map<char, double> frequencies;
    if (text.empty()) {
        return {};
    }
    for (char c : text) {
        frequencies[c]++;
    }
```

```
9
10      std::vector<FanoNode*> nodes;
11      nodes.reserve(frequencies.size());
12      for (auto& pair : frequencies) {
13          nodes.push_back(new FanoNode(pair.first, pair.second / text.size()));
14      }
15
16      std::sort(nodes.begin(), nodes.end(), [](FanoNode* a, FanoNode* b) {
17          return a->probability > b->probability;
18      });
19
20      std::map<char, std::string> codes;
21      if (!nodes.empty()) {
22          generateFanoCodes(nodes, 0, static_cast<int>(nodes.size()) - 1, "", codes);
23      }
24
25      for (FanoNode* node_ptr : nodes) {
26          delete node_ptr;
27      }
28
29      return codes;
30  }
```

Листинг 36: Функция buildFanoCodes

5.4 Функция `encodeFano`

- **Вход:**
  - `text`: `const std::string&` – исходный текст для кодирования.
  - `codes`: `const std::map<char, std::string>&` – таблица кодов Фано.
- **Выход:** `std::string` – строка, состоящая из символов ’0’ и ’1’, представляющая закодированный текст.
- **Описание:** Кодирует входной текст `text`, заменяя каждый символ его соответствующим кодом Шеннона-Фано из таблицы `codes`. Результатом является строка, представляющая последовательность битов. Данная реализация не выполняет упаковку битов в байты.

```
1  std::string encodeFano(const std::string& text, const std::map<char, std::string>&
       codes) {
2      std::string encoded_bit_string;
3      if (text.empty() || codes.empty()) return encoded_bit_string;
4
```

```
    encoded_bit_string.reserve(text.length() * 4);

    for (char c : text) {
        auto it = codes.find(c);
        if (it != codes.end()) {
            encoded_bit_string += it->second;
        } else {
             std::cerr << "ПРЕДУПРЕЖДЕНИЕ (encodeFano): Код для символа '" << c
                       << "' (ASCII: " << static_cast<int>(static_cast<unsigned char
                          >(c))
                       << ") не найден!" << std::endl;
        }
    }
    return encoded_bit_string;
}
```

Листинг 37: Функция encodeFano

5.5 Функция `decodeFano`

- **Вход:**
  - `encoded_bit_string`: `const std::string&` – строка из '0' и '1', представляющая закодированные данные.
  - `codes`: `const std::map<char, std::string>&` – таблица кодов Фано.
- **Выход:** `std::string` – исходный текст.
- **Описание:** Декодирует строку `encoded_bit_string`. Сначала создает обратную карту кодов ("код"→ "символ") для удобства поиска. Затем итерирует по "битовой"строке, накапливая биты в буфер. Как только содержимое буфера совпадает с одним из кодов в обратной карте, соответствующий символ добавляется к результату, и буфер очищается.

```
std::string decodeFano(const std::string& encoded_bit_string, const std::map<char,
    std::string>& codes) {
    if (encoded_bit_string.empty() || codes.empty()) return "";

    std::map<std::string, char> reversedCodes;
    for (auto& pair : codes) {
        if (!pair.second.empty()) {
            reversedCodes[pair.second] = pair.first;
        }
    }

```

```
    std::string decoded_text;
    std::string current_code_buffer;
    decoded_text.reserve(encoded_bit_string.length());

    for (char bit_char : encoded_bit_string) {
        current_code_buffer += bit_char;
        auto it = reversedCodes.find(current_code_buffer);
        if (it != reversedCodes.end()) {
            decoded_text += it->second;
            current_code_buffer.clear();
        }
    }
    if (!current_code_buffer.empty()) {
         std::cerr << "ПРЕДУПРЕЖДЕНИЕ (decodeFano): Остались необработанные биты в бу
            фере: "
                  << current_code_buffer << std::endl;
    }
    return decoded_text;
}
```

Листинг 38: Функция decodeFano

6 Вспомогательные функции обработки текста

6.1 Глобальная функция `processTextToWords`

- **Вход:** `text_utf8`: `const std::string&` – входной текст в кодировке UTF-8.
- **Выход:** `std::vector<std::string>` – вектор строк, где каждая строка представляет собой выделенное слово.
- **Описание:** Функция выполняет токенизацию (разбиение на слова) входной текстовой строки. Разделителями слов считаются пробельные символы (проверяемые `std::isspace`) и стандартные знаки препинания ASCII (проверяемые `std::ispunct`). Функция итерирует по байтам входной строки, накапливая последовательности байтов, не являющихся разделителями, во временную строку. При встрече разделителя или по достижении конца строки, накопленная непустая строка добавляется в результирующий вектор слов. Функция корректно обрабатывает многобайтовые UTF-8 символы, сохраняя их байтовые последовательности внутри выделенных слов.

```
std::vector<std::string> processTextToWords(const std::string& text_utf8) {
    std::vector<std::string> words;
    std::string current_word;
```

```
    for (char c_byte : text_utf8) {
        unsigned char u_c_byte = static_cast<unsigned char>(c_byte);
        if (std::isspace(u_c_byte) || std::ispunct(u_c_byte)) {
            if (!current_word.empty()) {
                words.push_back(current_word);
                current_word.clear();
            }
        } else {
            current_word += c_byte;
        }
    }
    if (!current_word.empty()) {
        words.push_back(current_word);
    }
    return words;
}
```

Листинг 39: Функция processTextToWords

6.2 Глобальная функция `readFileToString`

- **Вход:** `filepath`: `const std::string&` – путь к файлу.
- **Выход:** `std::string` – содержимое файла в виде строки.
- **Описание:** Считывает все содержимое указанного файла в строку. Файл открывается в бинарном режиме (`std::ios::binary`) для корректной обработки любых данных, включая UTF-8. Если открытие файла не удалось, генерируется исключение.

```
std::string readFileToString(const std::string& filepath) {
    std::ifstream file(filepath, std::ios::in | std::ios::binary);
    if (!file.is_open()) {
        throw std::runtime_error("Не удалось открыть файл: " + filepath);
    }
    std::stringstream buffer;
    buffer << file.rdbuf();
    return buffer.str();
}
```

Листинг 40: Функция readFileToString

7 Результаты работы

```
--- Главное Меню ---
1. Работать со словарем на Хеш-таблице
2. Работать со словарем на Красно-Черном дереве
3. RLE кодирование/декодирование текста
0. Выход
Ваш выбор: 1

--- Меню Словаря (Хеш-таблица) ---
1. Добавить слово
2. Удалить слово
3. Найти слово
4. Загрузить словарь из файла (перезаписать)
5. Дополнить словарь из файла
6. Очистить словарь
7. Показать текущее содержимое словаря (стандартный print)
8. Визуализировать структуру
0. Вернуться в главное меню
Ваш выбор:
```

Рис. 3. Главное меню.

```
--- Меню Словаря (Хеш-таблица) ---
1. Добавить слово
2. Удалить слово
3. Найти слово
4. Загрузить словарь из файла (перезаписать)
5. Дополнить словарь из файла
6. Очистить словарь
7. Показать текущее содержимое словаря (стандартный print)
8. Визуализировать структуру
0. Вернуться в главное меню
Ваш выбор: 3
Введите слово для поиска: мой
Слово 'мой' (ключ: 'мой') найдено, частота: 1
```

Рис. 4. Поиск слова.

```
--- Меню Словаря (Хеш-таблица) ---
1. Добавить слово
2. Удалить слово
3. Найти слово
4. Загрузить словарь из файла (перезаписать)
5. Дополнить словарь из файла
6. Очистить словарь
7. Показать текущее содержимое словаря (стандартный print)
8. Визуализировать структуру
0. Вернуться в главное меню
Ваш выбор: 8
Визуализация Хеш-таблицы (размер: 101, элементы: 56):
Корзина [0]: <пусто>
Корзина [1]: ("Печально": 1)
Корзина [2]: <пусто>
Корзина [3]: ("забавлять": 1)
Корзина [4]: <пусто>
Корзина [5]: <пусто>
Корзина [6]: ("Ему": 1)
Корзина [7]: ("себя": 2)
Корзина [8]: <пусто>
Корзина [9]: ("наука": 1)
Корзина [10]: <пусто>
Корзина [11]: <пусто>
Корзина [12]: <пусто>
Корзина [13]: <пусто>
Корзина [14]: ("Его": 1) -> ("низкое": 1)
Корзина [15]: ("Мой": 1)
Корзина [16]: ("Он": 1)
Корзина [17]: ("какая": 1) -> ("Не": 1) -> ("думать": 1)
Корзина [18]: <пусто>
Корзина [19]: <пусто>
Корзина [20]: ("уважать": 1)
Корзина [21]: <пусто>
Корзина [22]: ("Вздыхать": 1)
Корзина [23]: ("заставил": 1)
Корзина [24]: <пусто>
Корзина [25]: <пусто>
Корзина [26]: <пусто>
Корзина [27]: ("день": 1)
Корзина [28]: <пусто>
Корзина [29]: ("скука": 1)
Корзина [30]: ("отходя": 1)
Корзина [31]: <пусто>
Корзина [32]: ("больным": 1)
Корзина [33]: ("мог": 1) -> ("подушки": 1)
Корзина [34]: <пусто>
Корзина [35]: <пусто>
Корзина [36]: ("самых": 1)
Корзина [37]: ("шагу": 1)
```

Рис. 5. Отображение хэш-таблицы.

```
8. Визуализировать структуру
0. Вернуться в главное меню
Ваш выбор: 8
Визуализация Красно-Черного Дерева:

                                        ("шутку":1 B)
                                                        ("шагу":1 R)
                                ("честных":1 B)
                                                                        ("черт":1 R)
                                                        ("уважать":1 B)
                                                                        ("тебя":1 R)
                                        ("скука":1 R)
                                                                        ("сидеть":1 R)
                                                        ("себя":2 B)
                        ("самых":1 B)
                                                                        ("прочь":1 R)
                                                        ("про":1 B)
                                                                        ("пример":1 R)
                                        ("правил":1 B)
                                                                        ("поправлять":1 B)
                                                        ("подушки":1 R)
                                                                                        ("подносить":1 R)
                                                                        ("отходя":1 B)
                                ("ночь":1 R)
                                                                        ("низкое":1 R)
                                                        ("ни":1 B)
                                        ("не":2 B)
                                                        ("наука":1 B)
                                                                        ("мой":1 R)
("мог":1 B)
                                                                        ("лучше":1 R)
                                                        ("лекарство":1 B)
                                                                        ("коварство":1 R)
                                        ("какая":1 R)
                                                        ("и":3 B)
                                ("заставил":1 B)
                                                        ("занемог":1 R)
                                        ("забавлять":1 B)
                                                        ("же":1 R)
                        ("дядя":1 B)
                                                                                        ("думать":1 R)
                                                                        ("другим":1 B)
                                                        ("день":1 R)
                                                                        ("выдумать":1 B)
                                                                                        ("возьмет":1 R)
                                        ("в":1 B)
                                                                        ("больным":1 B)
                                                        ("боже":1 R)
                                                                                        ("С":1 R)
                                                                        ("Полуживого":1 B)
                                                                                        ("Печально":1 R)
                                ("Он":1 R)
                                                        ("Но":1 B)
                                                                        ("Не":1 R)
                                        ("Мой":1 B)
                                                                        ("Когда":2 B)
                                                                                        ("Какое":1 R)
                                                        ("И":1 R)
                                                                                        ("Ему":1 R)
                                                                        ("Его":1 B)
                                                                                        ("Вздыхать":1 R)

Конец визуализации.
```

Рис. 6. Отображение КЧ-дерева.

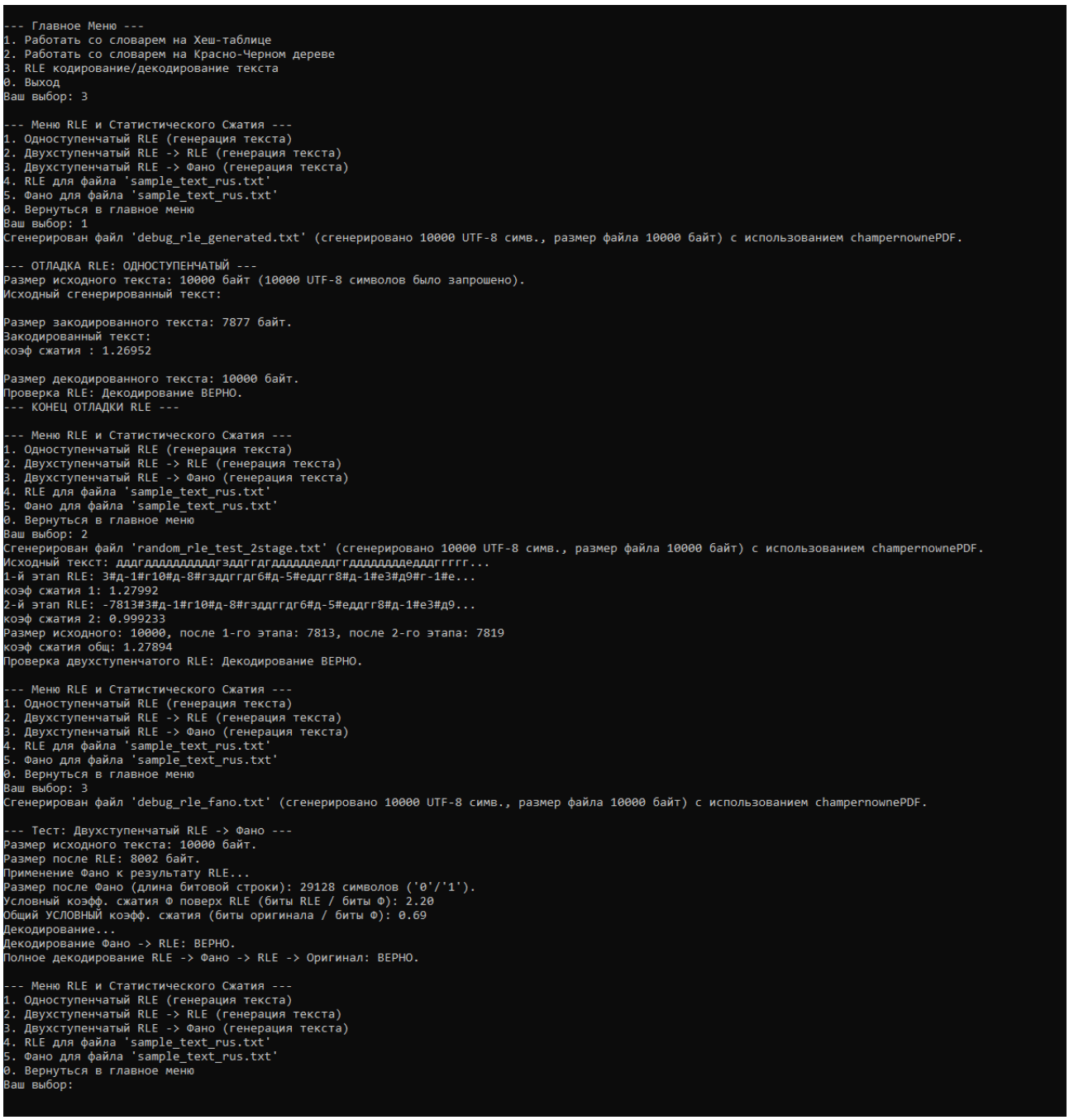

Рис. 7. Операции с функциями сжатия.

Заключение

В ходе выполнения работы были успешно реализованы структуры данных хеш-таблица и красно-черное дерево, а также словари на их основе с поддержкой всех заявленных операций. Реализован продвинутый алгоритм сжатия RLE.

Анализ RLE показал, что его эффективность сильно зависит от характера входных данных: для текстов с большим количеством повторов он может дать сжатие, тогда как для случайных данных или текстов естественного языка с низкой избыточностью повторов он может приводить к увеличению размера.

Достоинства

Главными достоинствами разработанной программы являются:

- **Реализация ключевых структур данных с нуля:** Программа демонстрирует глубокое понимание принципов работы хеш-таблиц и красно-черных деревьев, так как эти структуры были реализованы самостоятельно без использования стандартных контейнеров STL (за исключением базовых, таких как `std::vector` и `std::list` для внутренних нужд хеш-таблицы).
- **Интерактивный консольный интерфейс:** Наличие меню позволяет пользователю выбирать структуру данных для работы, выполнять операции со словарями и алгоритмом RLE, а также визуализировать внутреннее устройство хеш-таблицы и КЧ-дерева, что улучшает наглядность и удобство использования.
- **Обработка русского текста (UTF-8):** Приняты меры для базовой обработки текстов на русском языке в кодировке UTF-8 при чтении файлов и генерации случайного текста.
- **Структурированный и модульный код:** Код организован с использованием пространств имен для различных частей программы, что способствует его читаемости и поддержке.

Недостатки

К недостаткам текущей реализации можно отнести:

- **Ограниченная нормализация текста:** Приведение слов к нижнему регистру в словарях реализовано только для ASCII-символов. Для полноценной регистронезависимой обработки русского текста в UTF-8 потребовались бы более сложные решения или использование специализированных библиотек.
- **Консольная визуализация:** Хотя визуализация структур данных реализована, ее наглядность в консольном режиме ограничена, особенно для больших и сложных деревьев. Создание действительно "графического"представления дерева в консоли является нетривиальной задачей.

- **Производительность файловых операций и генерации текста:** Для очень больших файлов чтение всего содержимого в строку (`readFileToString`) или генерация больших случайных текстов символ за символом может быть не оптимальной с точки зрения производительности и использования памяти по сравнению с потоковыми методами обработки.
- **Двухэтапный алгоритм сжатия RLE:** Малоэффективен так как после первого этапа сжатия остается небольшой процент символов, которве можно сжать.

Масштабируемость:

Генерация текста: Текущий генератор случайного текста может быть расширен путем добавления новых символов, изменения их весов или логики генерации серий без изменения основной структуры функции. Производительность генерации очень больших текстов может потребовать оптимизации.

Возможные улучшения:

- Реализация полноценной поддержки UTF-8 для операций со строками в словарях (например, корректное приведение к нижнему регистру для русского языка).
- Рассмотрение возможности использования более эффективных форматов для хранения RLE-закодированных данных (например, бинарного формата вместо строкового представления чисел).
- Оптимизация файловых операций и генерации текста для работы с очень большими объемами данных (потоковая обработка).
- Улучшение консольной визуализации, например, путем более сложного форматирования или использования псевдографики для отображения ветвей дерева.
- Добавление измерения времени выполнения ключевых операций для сравнительного анализа производительности словарей на хеш-таблице и КЧ-дереве.

Список материалов

[1] Новиков, Ф.А. Дискретная математика для программистов. – СПб.: Питер, 2008. – С. 384. (дата обращения: 03.06.2025)
https://stugum.files.wordpress.com/2014/03/novikov.pdf

[2] Шапорев С.Д. "Дискретная математика"(дата обращения: 03.06.2025)
https://wiki.fenix.help/informatika/sdnf?ysclid=lr6bu660ud240614897

[3] Хаггарти Р. "Дискретная математика для программистов"(дата обращения: 24.05.2025)
https://studizba.com/show/1019108-1-gorbatov-va-fundamentalnye-osnovy.html