

МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО 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
по дисциплине «Теория графов»

Построение словаря на основе хеш-таблицы и В+-дерева

Обучающийся: _____

Гладков И.А.

Руководитель: _____

Востров А.В.

« _____ » _____ 20 ____ г.

Санкт-Петербург, 2024

Содержание

Введение	4
1 Математическое описание	5
1.1 Определение хеш-функции	5
1.2 Хеш-функция	5
1.3 Хеш-таблица	5
1.3.1 Разрешение коллизий методом цепочек	6
1.4 В+-дерево	7
1.5 Операции над В+-деревом	8
1.5.1 Поиск	8
1.5.2 Добавление	8
1.5.3 Удаление	9
1.5.4 Пример Б+-дерева	9
2 Особенности реализации	11
2.1 Класс HashTable	11
2.1.1 Метод hashCode	11
2.1.2 Метод Contains	12
2.1.3 Метод Add	12
2.1.4 Метод Delete	13
2.1.5 Метод toDown	13
2.1.6 Метод is_a_conj	14
2.1.7 Метод Delete_all	14
2.1.8 Метод From_file	15
2.1.9 Метод menu	16
2.2 Класс Node	22
2.3 Класс Root	22
2.3.1 Метод Add_word	22
2.3.2 Метод Add_word (с добавлением ссылок)	24
2.3.3 Метод Splitting	25
2.3.4 Метод Delete_word (удаление узла)	26
2.3.5 Метод Delete_word (удаление слова)	27
2.3.6 Метод Find_word	28
2.3.7 Метод Rebalancing	29
2.3.8 Метод Rebalancing_keys	31
2.4 Класс InnerNode	33
2.4.1 Метод Add_word	33

2.4.2	Метод Add_word (с добавлением ссылок)	33
2.4.3	Метод Splitting	34
2.4.4	Метод Delete_word	35
2.4.5	Метод Rebalancing	35
2.4.6	Метод Rebalancing_keys	37
2.5	Класс Leaf	38
2.5.1	Метод Add_word	38
2.5.2	Метод Add_word_without_key (добавление слова без ключа)	39
2.5.3	Метод Delete_word	40
2.5.4	Метод Splitting	40
2.5.5	Метод Delete_word	41
2.5.6	Метод Is_Sibling	42
2.5.7	Метод Rebalancing	42
2.5.8	Метод Rebalancing_keys	44
2.6	Класс Bplus	45
2.6.1	Метод is_a_conj	45
2.6.2	Метод Add_word	46
2.6.3	Метод Delete_word	47
2.6.4	Метод Find_word	47
2.6.5	Метод From_file	48
2.6.6	Метод Print	49
2.6.7	Метод Delete_all	51
2.6.8	Метод Rebalancing	51
2.6.9	Метод Menu	53
3	Результаты работы программы	59
	Заключение	67
	Список использованной литературы	68

Введение

В данной работе необходимо разработать словарь, в котором будут присутствовать операции добавления, удаления, поиска, очистки словаря и дополнение слов из текстового файла. Для хранения данных необходимо использовать хэш-таблица и B+ дерево.

1 Математическое описание

1.1 Определение хеш-функции

Хеш-функция — это математическая функция, которая преобразует значение ключа (входные данные произвольного размера) в фиксированное значение, называемое **хешем** или **хеш-значением**. Это значение служит для идентификации ключа в структуре данных, чаще всего в хеш-таблицах, или для вычисления адреса хранения данных в памяти или на диске. Основная цель хеш-функции — быстро и эффективно определить место хранения записи по ключу, так как хеш-значение указывает на конкретный адрес (индекс массива, кластер на диске или другой элемент структуры).

Хеш-функция разрабатывается таким образом, чтобы минимизировать вероятность того, что два разных ключа будут иметь одинаковое хеш-значение, но, поскольку мощность множества ключей зачастую значительно больше размера пространства возможных хешей, в большинстве случаев возникает ситуация, когда два или более разных ключа будут иметь одинаковое хеш-значение. Это явление называется *коллизией*.

Важное требование к хеш-функции — равномерное распределение ключей по множеству возможных хеш-значений. При этом множество возможных ключей обычно гораздо больше, чем размер пространства хеш-значений, что делает возникновение коллизий неизбежным.

1.2 Хеш-функция

В данной работе в качестве хеш-функции использовалось выражение $S = \sum_{i=0}^n (7+i)x_i^i$, где n — количество букв в слове; x_i — код i -ого символа слова по таблице ASCII. Выражение хешируется как:

$$H(S) = (7 \cdot x_0^0 + 8 \cdot x_1^1 + 9 \cdot x_2^2 + \dots + (7+n) \cdot x_n^n) \mod m$$

где:

- x_i — код по таблице ASCII i -го символа слова,
- n — количество символов в слове,
- m — размер хеш-таблицы или общее количество бакетов.

1.3 Хеш-таблица

Хеш-таблица — это структура данных, которая представляет собой одну из реализаций ассоциативной памяти. Она используется для хранения пар вида (**ключ**, **значение**) и поддерживает три основные операции: добавление пары, поиск и удаление пары по ключу.

Хеш-таблицы бывают двух основных типов: с открытой адресацией и с использованием метода цепочек (списков). В хеш-таблице с открытой адресацией каждый элемент массива либо содержит

пару (**ключ**, **значение**), либо пуст, тогда как в хеш-таблице с цепочками каждый элемент является списком пар, что позволяет хранить несколько пар в одном месте массива. Массив хеш-таблицы состоит из n ячеек, каждая из которых в зависимости от типа может содержать пару или список пар.

Ключевая часть работы хеш-таблицы — это **хеш-функция**, которая преобразует ключ в индекс массива хеш-таблицы. Важной особенностью является то, что хеш-функция зависит только от ключа и не использует значение. Впрочем, как было указано ранее (см. раздел 1.2), возможны ситуации, при которых различные ключи могут иметь одинаковый хеш — это явление называется **коллизией**. Для разрешения коллизий используются различные стратегии, такие как линейное пробирование при открытой адресации или хранение нескольких элементов в одной ячейке с помощью списка в методе цепочек.

Одним из важнейших параметров хеш-таблицы является **коэффициент загрузки**, который равен отношению числа хранимых элементов к количеству ячеек в массиве хеш-таблицы. Этот параметр оказывает значительное влияние на эффективность операций: чем выше коэффициент загрузки, тем больше вероятность коллизий, что может замедлить работу таблицы.

В идеальных условиях, при правильно подобранной хеш-функции и разумном значении коэффициента загрузки, все три основные операции (добавление, поиск и удаление) могут выполняться за время $O(1)$ в среднем. Однако в худшем случае время выполнения может быть значительно больше, особенно если происходит много коллизий. Когда коэффициент загрузки превышает определённый порог, возникает необходимость в **рехешировании** — процессе, при котором создаётся новый массив большего размера, и все существующие пары из старого массива переносятся в новый, с пересчётом индексов на основе новой хеш-функции.

Структура реализованной хеш-таблицы представлена рисунке 1

```
[0]: таблица
[1]: зеркало
[2]:
[3]: шкаф мультфильм
[4]:
[5]: олень
[6]: мельница
[7]:
[8]: овсянка
[9]: пустыня
[10]:
[11]: темнота
```

Рис. 1. Реализованная структура хеш-таблицы

1.3.1 Разрешение коллизий методом цепочек

Метод цепочек, или списков, является популярным способом разрешения коллизий в хеш-таблицах. В этом методе каждый элемент массива хеш-таблицы представляет собой связанный

список пар (**ключ**, **значение**). Когда несколько ключей хешируются в одно и то же значение (то есть происходит коллизия), все соответствующие пары помещаются в один и тот же список или массив (бакет).

Когда в методе цепочек при добавлении новой пары возникает коллизия, эта пара добавляется в конец массива (списка), связанного с индексом, на который указала хеш-функция. Время добавления элемента в конец массива составляет $O(n)$, где n — длина массива.

Операции удаления и поиска элемента в массиве требуют последовательного прохода по нему. Время поиска или удаления элемента будет $O(n)$, где n — количество элементов в массиве.

Среднее время выполнения операций в хеш-таблице зависит от **коэффициента загрузки** α , который равен отношению количества хранимых элементов к количеству бакетов. Если распределение хешей равномерное, то средняя длина массива при каждом индексе будет небольшой, и время поиска элемента составит $O(1 + \alpha)$. При низком коэффициенте загрузки ($\alpha \ll 1$) время выполнения операций близко к $O(1)$. Однако при высоком коэффициенте загрузки ($\alpha \gg 1$) длина массивов увеличивается, что замедляет операции поиска и удаления, делая их время выполнения ближе к $O(n)$.

Операции с массивом:

- **Добавление элемента:** Элемент добавляется в конец массива. Сложность операции — $O(n)$, где n — количество элементов в массиве.
- **Удаление элемента:** Требуется найти элемент, после чего удалить его. Сложность операции — $O(n)$.
- **Поиск элемента:** Необходимо последовательно пройти по массиву и найти элемент. Сложность операции — $O(n)$.

1.4 В+-дерево

В+-дерево — это сбалансированная и сильно ветвистая структура данных, предназначенная для эффективного хранения и поиска элементов. Основное преимущество В+-дерева — это возможность выполнения операций поиска, добавления и удаления за $O(\log n)$, где n — количество элементов в дереве.

Сбалансированность означает, что длина путей от корня до любого листа одинакова, что предотвращает деградацию производительности. **Ветвистость** дерева подразумевает, что каждый узел содержит ссылки на множество потомков, что уменьшает глубину дерева и, следовательно, время выполнения операций.

В+-дерево степени $t > 2$ обладает следующими основными свойствами:

- Каждый узел содержит хотя бы один ключ, при этом ключи в узлах упорядочены по возрастанию. Корневой узел содержит от 1 до $2t - 1$ ключей, а все остальные узлы содержат от $t - 1$ до $2t - 1$ ключей.

- Листовые узлы не имеют потомков. Внутренние узлы, содержащие n ключей K_1, K_2, \dots, K_n , имеют $n + 1$ потомков. При этом:
 - Первый потомок и все его ключи меньше K_1 .
 - Потомки между K_{i-1} и K_i содержат ключи, принадлежащие интервалу (K_{i-1}, K_i) для $2 \leq i \leq n$.
 - Последний потомок и все его ключи больше K_n .
- Все листовые узлы находятся на одном уровне, что гарантирует равномерную глубину.
- Листовые узлы содержат указатели на своих соседей, что обеспечивает эффективный обход дерева в порядке возрастания ключей.

1.5 Операции над B+-деревом

1.5.1 Поиск

Поиск элемента в B+-дереве начинается с корня и продолжается до листа. Благодаря свойству, что каждый потомок имеет ключи из определённого интервала, можно эффективно направлять поиск. Пусть требуется найти ключ k . В каждом внутреннем узле производится одно из следующих действий:

- Если k меньше наименьшего ключа узла, спускаемся к первому потомку.
- Иначе находим ключи K_i , при которых выполняется $K_i \leq k < K_{i+1}$, и спускаемся к $i + 1$ -му потомку.
- Если $k \geq K_n$, спускаемся к последнему потомку.

Процесс продолжается до тех пор, пока не будет найден соответствующий лист, который либо содержит ключ k , либо указывает на его отсутствие. Поскольку дерево сбалансировано, глубина поиска составляет $O(\log n)$.

1.5.2 Добавление

Чтобы добавить новый элемент в B+-дерево, сначала необходимо найти подходящий листовый узел для вставки ключа. Алгоритм добавления следующий:

- Если узел не заполнен, то ключ просто добавляется, сохраняя порядок.
- Если узел заполнен (содержит $2t - 1$ ключей), происходит расщепление:
 - Узел делится на два, при этом половина ключей переносится в новый узел.
 - Копия наименьшего ключа из нового узла добавляется в родительский узел.
 - Если родительский узел также заполнен, процесс расщепления продолжается вверх по дереву.

- Если расщепляется корневой узел, создаётся новый корень, содержащий один ключ и две ссылки на потомков.

Добавление элемента требует $O(\log n)$ операций, поскольку в худшем случае может потребоваться проход от листа до корня.

1.5.3 Удаление

Алгоритм удаления элемента из B+-дерева также начинается с поиска соответствующего листового узла. После нахождения ключа выполняются следующие шаги:

- Если после удаления узел остаётся наполовину заполненным (содержит не менее $t-1$ ключей), операция завершается.
- Если узел становится менее чем наполовину заполненным, необходимо перераспределить ключи с соседними узлами. Можно взять ключ у левого или правого «брата» (соседа на том же уровне).
- Если перераспределение невозможно, узлы объединяются с соседом, и ключ, указывающий на объединённые узлы, удаляется из родительского узла.

Удаление также выполняется за $O(\log n)$, так как может потребоваться корректировка структуры вплоть до корня.

1.5.4 Пример B+-дерева

На рисунке 2 представлен пример структуры B+-дерева.

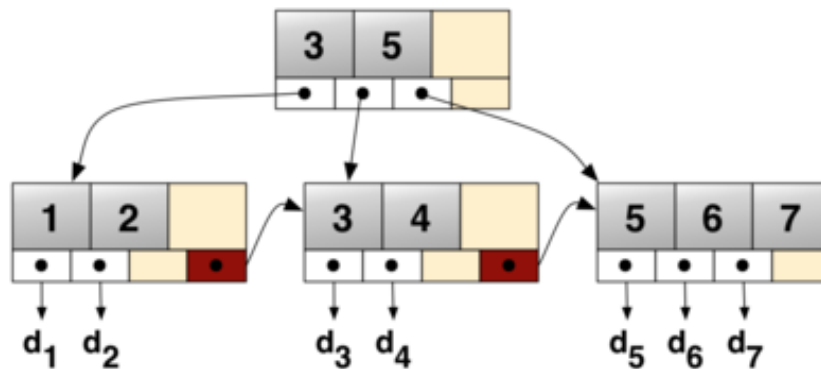


Рис. 2. Пример структуры B+-дерева

На рисунках 3-4 представлена реализованная заполненная структура Б+-деревя.

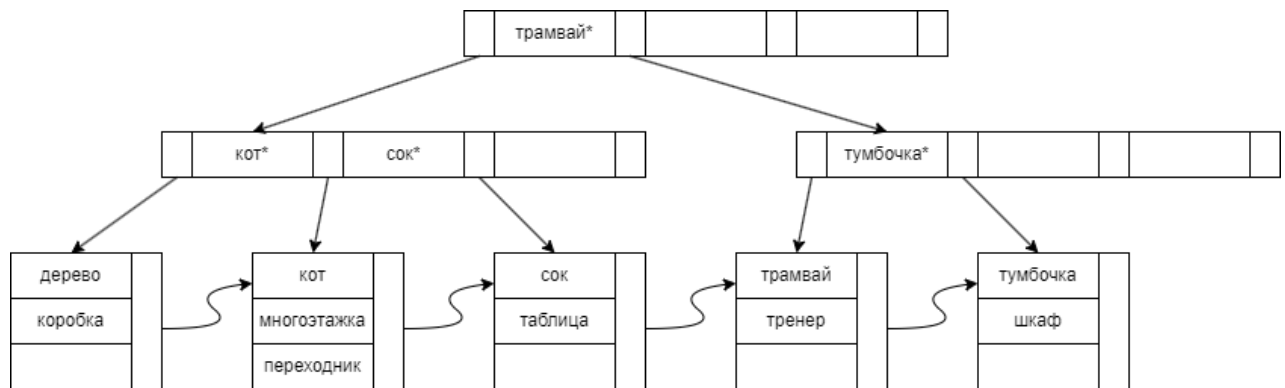


Рис. 3. Пример реализованной заполненной структуры Б+-деревя

```

трамвай*
  кот*  сок*
    дерево  коробка
    кот  многоэтажка  переходник
    сок  таблица
  тумбочка*
    трамвай  тренер
    тумбочка  шкаф
  
```

Рис. 4. Пример реализованной заполненной структуры Б+-деревя

2 Особенности реализации

2.1 Класс HashTable

Класс HashTable представляет из себя реализацию хеш-таблицы. В нем существуют следующие переменные:

1. **int count_baskets** – количество бакетов в хеш-таблице;
2. **int count_words** – количество слов в хеш-таблице;
3. **vector<vector<string>> baskets** – динамический массив бакетов; в каждом бакете динамический массив, хранящий в себе слова;
4. **vector<string> conj** – массив, хранящий в себе слова, которые не добавляются в хеш-таблицу (местоимения, союзы и тп.);
5. **double percent** – коэффициент заполнения таблицы;

2.1.1 Метод HashCode

Вход: **string str** - слово, добавляемое в хеш-таблицу.

Выход: **int** - хеш слова.

Метод **hashFunction** вычисляет хеш-значение для слова **str**. Данный метод служит для вычисления хеша, которое затем используется для определения индекса в хеш-таблице.

Алгоритм работы следующий: метод инициализирует переменную **result** значением **indx**, которое установлено равным 7. Далее для каждого символа строки метод вычисляет числовое значение символа **x** и последовательно умножает его на себя (возводя в степень, равную индексу символа в строке). Итоговое значение для каждого символа добавляется к **result** с учетом текущего индекса символа и базового значения **indx**.

Код метода представлен в листинге 1.

```
1 int HashTable::HashCode(string str) {
2     int result = 0;
3     int indx = 7;
4
5     result = indx;
6
7     for (int i = 1; i < str.size() + 1; i++) {
8         int x = static_cast<int>(str[i - 1]);
9         for (int j = 1; j < i; j++) {
10             x *= x;
11         }
12         result += (indx + i) * x;
13     }
```

```

14     return result;
15 }

```

Листинг 1. Метод `HashCode`

2.1.2 Метод `Contains`

Вход: `string str` — слово для поиска.

Выход: `bool` — `true`, если слово найдено, иначе `false`.

Метод `Contains` проверяет, содержится ли слово `str` в хеш-таблице. Для этого вычисляется хеш-код слова, определяется соответствующий бакет, и производится поиск слова в этом бакете. Если слово найдено, возвращается `true`, в противном случае — `false`.

Код метода представлен в листинге 2.

```

1  bool HashTable::Contains(string str) {
2      int code = HashCode(str);
3      int n = code % count_baskets;
4      n = n < 0 ? -1 * n : n;
5      for (string word : baskets[n]) {
6          if (word == str)
7              return true;
8      }
9      return false;
10 }

```

Листинг 2. Метод `Contains`

2.1.3 Метод `Add`

Вход: `string str` — строка для добавления.

Выход: слово добавлено в словарь.

Метод `Add` добавляет строку `str` в хеш-таблицу. Если строка уже существует в таблице, она не добавляется. При превышении заполненности таблицы на 75

Код метода представлен в листинге 3.

```

1  void HashTable::Add(string str) {
2      str = toDown(str);
3      if (!Contains(str)) {
4          if (percent >= 75) {
5              vector<vector<string>> buf(count_baskets * 2);
6              count_baskets *= 2;
7              for (vector<string> strings : baskets) {
8                  for (string s : strings) {
9                      int code = HashCode(s);
10                     int n = code % count_baskets;
11                     n = n < 0 ? -1 * n : n;

```

```

12         buf[n].push_back(s);
13     }
14
15     }
16     buckets = buf;
17 }
18 int code = GetHashCode(str);
19 int n = code % count_buckets;
20 n = n < 0 ? -1 * n : n;
21 buckets[n].push_back(str);
22 count_words++;
23 percent = static_cast<double>(count_words) / count_buckets * 100;
24 }
25 }

```

Листинг 3. Метод Add

2.1.4 Метод Delete

Вход: string str — слово для удаления.

Выход: слово удалено из словаря.

Метод `Delete` удаляет слово `str` из хеш-таблицы, если оно там содержится. Сначала проверяется наличие слова в таблице, после чего оно удаляется из соответствующего бакета.

Код метода представлен в листинге 4.

```

1 void HashTable::Delete(string str) {
2     str = toDown(str);
3     if (Contains(str)) {
4         int code = GetHashCode(str);
5         int n = code % count_buckets;
6         n = n < 0 ? -1 * n : n;
7         for (int i = 0; i < buckets[n].size(); i++) {
8             if (buckets[n][i] == str) {
9                 buckets[n].erase(buckets[n].begin() + i);
10                count_words--;
11                break;
12            }
13        }
14    }
15 }

```

Листинг 4. Метод Delete

2.1.5 Метод toDown

Вход: string str — слово для преобразования.

Выход: слово, преобразованное в нижний регистр.

Метод `toDown` преобразует все буквы слова `str` в нижний регистр, поддерживая как латинские, так и кириллические символы.

Код метода представлен в листинге 5.

```
1  string HashTable::toDown(string str) {
2      string result = "";
3      for (char c : str) {
4          if (c >= 'A' && c <= 'Z') {
5              c = c + ('a' - 'A');
6          }
7          if (c >= 'А' && c <= 'Я') {
8              c = c + ('а' - 'А');
9          }
10         result += c;
11     }
12     return result;
13 }
```

Листинг 5. Метод `toDown`

2.1.6 Метод `is_a_conj`

Вход: `string str` — слово для проверки.

Выход: `bool` — `true`, если слово является исключением, `false` — если нет.

Метод `is_a_conj` проверяет, является ли слово `str` словом-исключением, сравнивая его с перечнем слов-исключений `conj`.

Код метода представлен в листинге 6.

```
1
2  bool HashTable::is_a_conj(string str) {
3      str = toDown(str);
4      for (string word : conj) {
5          if (str == word) {
6              return true;
7          }
8      }
9      return false;
10 }
```

Листинг 6. Метод `is_a_conj`

2.1.7 Метод `Delete_all`

Вход: хеш-таблица.

Выход: словарь полностью очищен.

Метод `Delete_all` полностью очищает хеш-таблицу, сбрасывая все бакеты, количество слов и восстанавливая исходное количество бакетов.

Код метода представлен в листинге 7.

```
1 void HashTable::Delete_all() {
2     vector<vector<string>> buf(6);
3     buckets = buf;
4     count_buckets = 6;
5     count_words = 0;
6 }
```

Листинг 7. Метод Delete_all

2.1.8 Метод From_file

Вход: string str — путь к файлу.

Выход: слова из файла добавлены в словарь.

Метод From_file считывает слова из файла по указанному пути **str**, очищает их от знаков пунктуации и добавляет в хеш-таблицу. Исключения не добавляются. В случае неудачной попытки открытия файла выводится сообщение об ошибке.

Код метода представлен в листинге 8.

```
1 void HashTable::From_file(string str) {
2     //std::filesystem::path file_path = str;
3     //ifstream file(R"(str)");
4     ifstream file(str);
5     if (!file.is_open()) {
6         printf("\nНе удалось открыть файл\n\n");
7     }
8     else {
9         string line;
10        vector<string> words = { "" };
11        int last_index = 0;
12        char c;
13        int k = 0;
14        while (file.get(c)) {
15            if (c != ',' && c != '.' && c != ':' && c != ';' && c != '!' && c != '?' && c != '-'
16                && c != '?')
17                && c != '(' && c != ')' && c != '{' && c != '}' && c != '[' && c != ']' && c != '\',
18                && c != '"') {
19                if (c != ' ' && c != '\n') {
20                    words[last_index] += c;
21                }
22                else {
23                    if (words[last_index].size() > 0) {
24                        words.push_back("");
25                        last_index++;
26                    }
27                }
28            }
29        }
30    }
```

```

27     }
28
29 }
30
31 if (words[words.size() - 1] == "") {
32     words.pop_back();
33 }
34
35 file.close();
36 for (auto word : words) {
37     if (!this->is_a_conj(word)) {
38         this->Add(word);
39     }
40 }
41
42 printf("\nСлова из файла добавлены в словарь\n\n");
43 }
44 }

```

Листинг 8. Метод From_file

2.1.9 Метод menu

Вход: создание консольного меню, для работы с хеш-таблицей.

Выход: консольное меню для работы с хеш-таблицей.

Метод menu реализует цикл пользовательского меню, предоставляя различные опции для работы со словарём, такие как просмотр содержимого, добавление и удаление слов, очистка словаря, проверка наличия слов, загрузка данных из файла и просмотр списка исключений. Меню завершает свою работу при выборе пункта выхода.

Код метода представлен в листинге 9.

```

1 void HashTable::menu() {
2     while (true) {
3         printf("\n----- Словарь (хэш-таблица) ----- \n");
4         printf("Выберите действие:\n");
5         printf("[1] - просмотр содержимое словаря \n");
6         printf("[2] - добавление нового слова\n");
7         printf("[3] - удаление слова\n");
8         printf("[4] - проверка наличие слов в словаре\n");
9         printf("[5] - полная очистка словаря\n");
10        printf("[6] - добавление слов из файла\n");
11        printf("[7] - просмотр список недобавляющихся слов\n");
12
13        printf("[0] - выход из словаря\n");
14        printf("----- \n");
15
16        int choose;

```



```

17     bool out = false;
18     string word;
19     string file;
20     vector<string> files = { "Мастер и Маргарита.txt", "Преступление и наказание.txt" };
21     int k;
22     bool flag_out_path;
23
24     while (true) {
25
26         cin >> choose;
27
28         // Проверка на корректность ввода
29         if (std::cin.fail()) {
30             std::cin.clear(); // Сброс состояния потока
31             std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Очистка бу
32             fера ввода
33             std::cout << "Некорректный ввод! Пожалуйста, введите число." << std::endl;
34         }
35         else if (choose < 0 || choose > 7) {
36             // Проверка, что число находится в нужном диапазоне
37             std::cout << "Число вне диапазона! Пожалуйста, введите число от 0 до 7."<< std::
38             endl;
39         }
40         else {
41             break; // Выход из цикла, если ввод корректен и число в диапазоне
42         }
43     }
44
45     switch (choose)
46     {
47         case 0:
48             out = true;
49             break;
50         case 1:
51             if (count_words == 0) {
52                 printf("\nСловарь пуст\n\n");
53             }
54             else {
55                 printf("\nСодержимое словаря:\n");
56                 printf("[1] - по бакетам \n");
57                 printf("[2] - списком \n\n");
58                 printf("Как вы хотите? Введите свой выбор: ");
59                 while (true) {
60
61                     cin >> k;
62
63                     // Проверка на корректность ввода

```

```

63         if (std::cin.fail()) {
64             std::cin.clear(); // Сброс состояния потока
65             std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Очистк
а буфера ввода
66             std::cout << "Некорректный ввод! Пожалуйста, введите число." << std::endl;
67         }
68         else if (k < 1 || k > 2) {
69             // Проверка, что число находится в нужном диапазоне
70             std::cout << "Число вне диапазона! Пожалуйста, введите число от 1 до 2." <<
std::endl;
71         }
72         else {
73             break; // Выход из цикла, если ввод корректен и число в диапазоне
74         }
75     }
76
77
78     switch (k) {
79         case 2:
80             k = 1;
81             for (auto words : buckets) {
82                 for (auto word : words) {
83                     printf("%d. %s \n", k, word.c_str());
84                     k++;
85                 }
86             }
87
88             break;
89
90         case 1:
91             k = 0;
92             for (auto words : buckets) {
93                 printf("[%d]: ", k);
94                 for (auto word : words) {
95                     cout << word << ' ';
96                 }
97                 cout << endl;
98                 k++;
99             }
100             printf("\nКоличество слов: %d\n\n", count_words);
101             break;
102         }
103     }
104     break;
105     case 2:
106     while (true) {
107         printf("Введите слово для добавления в словарь (для выхода в меню напишите '0'):"
");

```

```

108     cin >> word;
109     if (word == "0") {
110         break;
111     }
112     else {
113         if (!Contains(word)) {
114             if (!is_a_conj(word)) {
115                 this->Add(word);
116                 printf("\n\tСлово '%s' добавлено в словарь\n\n", word.c_str());
117                 //break;
118             }
119             else {
120                 printf("\n\tСлово '%s' не добавлено в словарь, так как оно входит в список
недобавляющихся слов :(\n\n", word.c_str());
121             }
122         }
123         else {
124             printf("\n\tСлово '%s' не добавлено в словарь, оно уже там присутствует\n\n",
word.c_str());
125         }
126     }
127
128 };
129 break;
130
131 case 3:
132 while (true) {
133     printf("введите слово, которое хотите удалить (для выхода в меню напишите '0'): "
);
134     cin >> word;
135     if (word == "0") {
136         break;
137     }
138     else {
139         if (!this->Contains(word)) {
140             printf("\n\tСлово '%s' отсутствует\n\n", word.c_str());
141         }
142         else {
143             this->Delete(word);
144             printf("\n\tСлово '%s' удалено\n\n", word.c_str());
145             //break;
146         }
147     }
148 }
149 break;
150
151 case 4:
152 while (true) {

```

```

153     printf("Введите слово, наличие которого хотите проверить (для выхода в меню напишит
e '0'): ");
154     cin >> word;
155     if (word == "0") {
156         break;
157     }
158     else {
159         if (this->Contains(word)) {
160             printf("\n\tСлово '%s' присутствует в словаре\n\n", word.c_str());
161         }
162         else {
163             printf("\n\tСлово '%s' отсутствует в словаре\n\n", word.c_str());
164         }
165         //break;
166     }
167 }
168 break;
169
170 case 5:
171     this->Delete_all();
172     printf("\nСловарь полностью очищен\n\n");
173     break;
174
175 case 6:
176     printf("\nВыберите файл из существующих:\n");
177     printf("1. %s\n", files[0].c_str());
178     printf("2. %s\n", files[1].c_str());
179     printf("3. другое...\n\n");
180     printf("0. ВЫХОД\n\n");
181     printf("Ваш выбор: ");
182
183     flag_out_path = true;
184
185     while (true) {
186
187         cin >> k;
188
189         // Проверка на корректность ввода
190         if (std::cin.fail()) {
191             std::cin.clear(); // Сброс состояния потока
192             std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Очистка
буфера ввода
193             std::cout << "Некорректный ввод! Пожалуйста, введите число." << std::endl;
194         }
195         else if (k < 0 || k > 3) {
196             // Проверка, что число находится в нужном диапазоне
197             std::cout << "Число вне диапазона! Пожалуйста, введите число от 0 до 3." << std
::endl;

```

```

198     }
199     else {
200         break; // Выход из цикла, если ввод корректен и число в диапазоне
201     }
202 }
203
204
205 switch (k) {
206     case 1:
207         //file = "D:/Another/CLion Projects/txt files/" + files[0];
208         file = files[0];
209         break;
210     case 2:
211         //file = "D:/Another/CLion Projects/txt files/" + files[1];
212         file = files[1];
213         break;
214     case 3:
215         printf("напишите название своего файла с '.txt': ");
216         cin >> file;
217         break;
218     case 0:
219         flag_out_path = false;
220         break;
221 }
222 if (flag_out_path)
223     this->From_file(file);
224 break;
225
226 case 7:
227     printf("Список недобавляющихся слов\n");
228
229     for (int i = 0; i < this->conj.size(); i++) {
230         printf("%d. %s\n", i + 1, this->conj[i].c_str());
231     }
232     cout << "\n";
233 }
234
235 if (out) {
236     break;
237 }
238 system("pause");
239 }
240 }

```

Листинг 9. Метод menu

2.2 Класс Node

Класс Node является базовой реализацией узла B+-дерева. Этот класс обладает следующими полями:

1. `int DEGREE` — степень дерева;
2. `int count_words` — количество слов (ключей) в узле;
3. `vector<string> words` — массив слов (ключей) в узле;
4. `vector<Node*> links` — массив ссылок на детей;
5. `Node* parent_link` — ссылка на родителя;

2.3 Класс Root

Класс Root является реализацией корневого узла B+-дерева. Он наследует класс Node. Дополнительные переменных не имеет.

2.3.1 Метод Add_word

Вход: `string str` — слово для добавления.

Выход: слово добавлено в дерево.

Метод `Add_word` добавляет слово `str` в дерево, следуя правилам вставки для B+ деревьев. Если узел (лист или внутренний) переполняется, происходит разделение узлов и продвижение ключа вверх по дереву.

Код метода представлен в листинге 10.

```
1 void Root::Add_word(string str) {
2     str = toDown(str);
3     if (!Find_word(str))
4         if (links.size() == 0) { //первый случай
5             if (words.size() == 0) {
6                 words.push_back(str);
7                 count_words++;
8             }
9             else {
10                bool flag = true;
11                for (int i = 0; i < words.size(); i++) {
12                    if (str < words[i]) {
13                        words.insert(words.begin() + i, str);
14                        flag = false;
15                        count_words++;
16                        break;
17                    }
18                }
19                if (flag) {
```

```

20         words.push_back(str);
21         count_words++;
22     }
23 }
24 }
25 else { //остальные случаи
26     Node* current_link = this;
27     Node* previous_link = nullptr;
28
29     //проходит по дереву до листа
30     while (true) {
31         if (Leaf* buf_leaf = dynamic_cast<Leaf*>(current_link))
32             break;
33         vector<string> keys = current_link->Get_Words();
34         int index = 0;
35         for (int i = 0; i < keys.size(); i++) {
36             if (str >= keys[i])
37                 index = i + 1;
38         }
39         previous_link = current_link;
40         current_link = current_link->Get_Links()[index];
41     }
42
43     if (current_link->Get_Count_words() < 2 * DEGREE - 1) {
44         current_link->Add_word(str);
45     }
46     //сейчас будет подниматься если происходит переполнение
47     else {
48
49         while (true) {
50             if (current_link == nullptr)
51                 break;
52             if (Leaf* buf_leaf = dynamic_cast<Leaf*>(current_link)) {
53                 if (current_link->Get_Count_words() < 2 * DEGREE - 1)
54                     break;
55                 vector<Node*> splited_leaf = current_link->Splitting(str);
56                 previous_link->Add_word(current_link->Get_Words()[2], splited_leaf);
57                 splited_leaf[0]->Set_parent_link(previous_link);
58                 splited_leaf[1]->Set_parent_link(previous_link);
59
60                 if (buf_leaf->Get_brother() != nullptr) {
61                     buf_leaf->Get_brother()->Clear_Links();
62                     buf_leaf->Get_brother()->Set_Links(splited_leaf[0]);
63                     if (Leaf* leaf = dynamic_cast<Leaf*>(splited_leaf[0])) {
64                         leaf->Set_brother(buf_leaf);
65                     }
66                 }
67                 if (buf_leaf->Get_Links().size() > 0) {

```

```

68         splitted_leaf[1]->Set_Links(buf_leaf->Get_Links()[0]);
69         if (Leaf* leaf = dynamic_cast<Leaf*>(buf_leaf->Get_Links()[0])) {
70             leaf->Set_brother(splited_leaf[1]);
71         }
72     }
73 }
74 if (InnerNode* buf_node = dynamic_cast<InnerNode*>(current_link)) {
75     if (current_link->Get_Count_words() < 2 * DEGREE)
76         break;
77     vector<Node*> splitted_node = current_link->Splitting(str);
78     previous_link->Add_word(current_link->Get_Words()[2], splitted_node);
79     splitted_node[0]->Set_parent_link(previous_link);
80     splitted_node[1]->Set_parent_link(previous_link);
81 }
82 if (Root* buf_root = dynamic_cast<Root*>(current_link)) {
83     if (current_link->Get_Count_words() < 2 * DEGREE)
84         break;
85     vector<Node*> splitted_root = current_link->Splitting(str);
86     auto actual_key = current_link->Get_Words()[2];
87     buf_root->Root_Cleaning();
88     current_link->Add_word(actual_key, splitted_root);
89     splitted_root[0]->Set_parent_link(current_link);
90     splitted_root[1]->Set_parent_link(current_link);
91 }
92 if (previous_link != nullptr) {
93     //delete current_link;
94     current_link = previous_link;
95     previous_link = current_link->Get_Parent_link();
96 }
97
98 }
99 }
100 }
101 }

```

Листинг 10. Метод Add_word

2.3.2 Метод Add_word (с добавлением ссылок)

Вход: string str — слово для добавления, vector<Node*> link — ссылки на два узла.

Выход: слово добавлено в дерево с обновлением ссылок на узлы.

Метод Add_word добавляет слово str и обновляет ссылки на дочерние узлы в B+ дереве. Если вставляемое слово меньше текущих, оно вставляется на нужную позицию вместе с ссылками на узлы. При необходимости ссылки корректируются для поддержания связности узлов.

Код метода представлен в листинге 11.

```

1 void Root::Add_word(string str, vector<Node*> link) {

```



```

2   bool flag_link = false;
3   if (Leaf* buf_leaf = dynamic_cast<Leaf*>(link[0]))
4   flag_link = true;
5   if (words.size() == 0) {
6       words.push_back(str);
7       links.push_back(link[0]);
8       links.push_back(link[1]);
9       count_words++;
10  }
11  else {
12      bool flag = true;
13      for (int i = 0; i < words.size(); i++) {
14          if (str < words[i]) {
15              words.insert(words.begin() + i, str);
16              links.erase(links.begin() + i);
17              links.insert(links.begin() + i, link[0]);
18              links.insert(links.begin() + i + 1, link[1]);
19              flag = false;
20              count_words++;
21              break;
22          }
23      }
24      if (flag) {
25          words.push_back(str);
26          links.pop_back();
27          links.push_back(link[0]);
28          links.push_back(link[1]);
29          count_words++;
30      }
31  }
32  if (flag_link) {
33      for (int i = 0; i < links.size() - 1; i++) {
34          links[i]->Clear_Links();
35          links[i]->Set_Links(links[i + 1]);
36      }
37  }
38  }

```

Листинг 11. Метод Add_word (с добавлением ссылок)

2.3.3 Метод Splitting

Вход: string str — слово для добавления и разделения узла.

Выход: два новых узла.

Метод **Splitting** добавляет слово **str** и делит текущий корень на два новых внутренних узла, если в корне произошло переполнение. Первые два слова остаются в одном узле, оставшиеся — во втором. Если узел имеет ссылки на другие узлы, они также распределяются между новыми узлами.

Код метода представлен в листинге 12.

```
1 vector<Node*> Root::Splitting(string str) {
2     bool flag = true;
3     for (int i = 0; i < words.size(); i++) {
4         if (str < words[i]) {
5             words.insert(words.begin() + i, str);
6             flag = false;
7             count_words++;
8             break;
9         }
10    }
11    if (flag) {
12        words.push_back(str);
13        count_words++;
14    }
15
16    InnerNode* root1 = new InnerNode;
17    root1->Add_word(words[0]);
18    root1->Add_word(words[1]);
19
20    InnerNode* root2 = new InnerNode;
21    root2->Add_word(words[3]);
22
23    if (links.size() != 0) {
24        root1->Set_Links(vector<Node*>{links[0], links[1], links[2]});
25        root2->Set_Links(vector<Node*>{links[3], links[4]});
26    }
27
28    return vector<Node*>{root1, root2};
29 }
```

Листинг 12. Метод Splitting

2.3.4 Метод Delete_word (удаление узла)

Вход: Node* node — указатель на узел для удаления.

Выход: узел удалён из списка ссылок, слово удалено из дерева.

Метод Delete_word удаляет указанный узел node из списка ссылок и удаляет последнее слово из текущего узла. После удаления количество слов уменьшается.

Код метода представлен в листинге 13.

```
1 void Root::Delete_word(Node* node) {
2     for (int i = 0; i < links.size(); i++) {
3         if (links[i] == node) {
4             links.erase(links.begin() + i);
5             words.pop_back();
6             count_words--;
```

```

7         break;
8     }
9 }
10 }

```

Листинг 13. Метод Delete_word

2.3.5 Метод Delete_word (удаление слова)

Вход: string str — слово для удаления.

Выход: удалено слово из дерева, возвращён указатель на текущий узел.

Метод Delete_word удаляет слово str из дерева. Если текущий узел — лист, слово удаляется непосредственно из него. Если нет, метод проходит по дереву до листа, где хранится слово, и удаляет его. По завершении возвращается указатель на текущий узел, где было найдено и удалено слово.

Код метода представлен в листинге 14.

```

1 Node* Root::Delete_word(string str) {
2     Node* current_link = this;
3     Node* previous_link = nullptr;
4
5     if (links.size() == 0) {
6         for (int i = 0; i < words.size(); i++) {
7             if (str == words[i]) {
8                 words.erase(words.begin() + i);
9                 count_words--;
10                return current_link;
11            }
12        }
13    }
14    //проходит по дереву до листа
15    while (true) {
16        if (Leaf* buf_leaf = dynamic_cast<Leaf*>(current_link))
17            break;
18        vector<string> keys = current_link->Get_Words();
19        int index = 0;
20        for (int i = 0; i < keys.size(); i++) {
21            if (str >= keys[i])
22                index = i + 1;
23        }
24        previous_link = current_link;
25        current_link = current_link->Get_Links()[index];
26    }
27    ///еще разобрать ситуацию когда удаляется в корне, а не в листе
28    if (Leaf* leaf = dynamic_cast<Leaf*>(current_link)) {
29        leaf->Delete_word(str);
30    }
31    return current_link;

```

Листинг 14. Метод Delete_word

2.3.6 Метод Find_word

Вход: string str — слово для поиска.

Выход: найдено ли слово в дереве (true/false).

Метод Find_word проверяет, присутствует ли слово **str** в дереве. Если узел не имеет ссылок, поиск происходит по списку слов в текущем узле. Если узел имеет ссылки, метод проходит по дереву до листа, чтобы найти слово. Если слово не найдено в текущем листе, дополнительно проверяется соседний узел.

Код метода представлен в листинге 15.

```

1  bool Root::Find_word(string str) {
2      if (words.size() == 0) {
3          return false;
4      }
5      if (links.size() == 0) {
6          for (auto word : words) {
7              if (word == str)
8                  return true;
9          }
10     }
11     else {
12         if (links.size() == 0) {
13             for (auto word : words) {
14                 if (word == str)
15                     return true;
16             }
17         }
18         else {
19             Node* current_link = this;
20             while (true) {
21                 if (Leaf* buf_leaf = dynamic_cast<Leaf*>(current_link))
22                     break;
23                 vector<string> keys = current_link->Get_Words();
24                 int index = 0;
25                 for (int i = 0; i < keys.size(); i++) {
26                     if (str >= keys[i])
27                         index = i + 1;
28                 }
29                 current_link = current_link->Get_Links()[index];
30             }
31             for (auto word : current_link->Get_Words()) {
32                 if (str == word)
33                     return true;

```

```

34     }
35     //если не нашлось на всякий проверим брата справа, вдруг он там
36     if (current_link->Get_Links().size() > 0) {
37         current_link = current_link->Get_Links()[0];
38         for (auto word : current_link->Get_Words()) {
39             if (str == word)
40                 return true;
41         }
42     }
43
44 }
45 }
46 return false;
47 }

```

Листинг 15. Метод Find_word

2.3.7 Метод Rebalancing

Вход: Node* node — указатель на узел для ребалансировки.

Выход: дерево сбалансировано.

Метод **Rebalancing** выполняет балансировку поддеревьев для поддержания структуры дерева. Для листьев метод завершает работу, а для внутренних узлов проверяет количество потомков, объединяя их при необходимости и корректируя ссылки на родителей. Если узлы могут быть разделены равномерно, метод распределяет потомков по узлам, иначе удаляет лишние ссылки.

Код метода представлен в листинге 16.

```

1 void Root::Rebalancing(Node* node) {
2     if (!node) return;
3     if (Leaf* leaf = dynamic_cast<Leaf*>(node)) {
4         return;
5     }
6     if (InnerNode* inner_node = dynamic_cast<InnerNode*>(node->Get_Links()[0])) {
7         auto current_links = node->Get_Links();
8         int count_links = node->Get_Links().size();
9         vector<Node*> new_links;
10
11         for (auto link : current_links) {
12             auto children_links = link->Get_Links();
13             for (auto child_link : children_links) {
14                 new_links.push_back(child_link);
15             }
16         }
17
18         if (new_links.size() < 4 && current_links.size()==2) { //когда остается два узла (1 р
            ебенок и 2 ребенка) то нужно их соединить с узлом-родителем ( в родителе будет 3 ссыл
            ки на детей)

```

```

19     node->Clear_Links();
20     node->Set_Links(new_links);
21     for (auto link : new_links) {
22         link->Set_parent_link(node);
23     }
24     for (auto child : node->Get_Links()) {
25         Rebalancing(child);
26     }
27 }
28 else {
29
30     int count = new_links.size() / count_links;
31     if (count > 1) {
32         for (int i = 0; i < current_links.size(); i++) {
33             current_links[i]->Clear_Links();
34             if (i != current_links.size() - 1) {
35                 for (int ii = 0; ii < count; ii++) {
36                     current_links[i]->Set_Links(new_links[0]);
37                     new_links[0]->Set_parent_link(current_links[i]); // поправляем ссылку на род
38 ителя
39                     new_links.erase(new_links.begin());
40                 }
41             }
42             else {
43                 current_links[i]->Set_Links(new_links);
44                 for (auto link : new_links) {
45                     link->Set_parent_link(current_links[i]);
46                 }
47             }
48         }
49         else { //если остается одна ссылка в каком то из узлов -> нужно убрать одну ссылку и
50 з родителя
51             count++;
52             for (int i = 0; i < current_links.size() - 1; i++) {
53                 current_links[i]->Clear_Links();
54                 if (i != current_links.size() - 2) {
55                     for (int ii = 0; ii < count; ii++) {
56                         current_links[i]->Set_Links(new_links[0]);
57                         new_links[0]->Set_parent_link(current_links[i]); // поправляем ссылку на род
58 ителя
59                         new_links.erase(new_links.begin());
60                     }
61                 }
62             }
63             else {
64                 current_links[i]->Set_Links(new_links);
65                 for (auto link : new_links) {
66                     link->Set_parent_link(current_links[i]);

```

```

64         }
65     }
66 }
67     links.pop_back();
68 }
69
70     for (auto child : current_links) {
71         Rebalancing(child);
72     }
73 }
74 }
75
76 else if (Root* root = dynamic_cast<Root*>(node)) {
77     if (node->Get_Links().size() == 1) {
78         auto current_links = node->Get_Links();
79         vector<string> new_words;
80         for (auto link : current_links) {
81             for (auto word : link->Get_Words()) {
82                 new_words.push_back(word);
83             }
84         }
85         node->Clear_word();
86         node->Set_Words(new_words);
87         node->Clear_Links();
88     }
89     for (auto child : root->Get_Links()) {
90         Rebalancing(child);
91     }
92 }
93 }

```

Листинг 16. Метод Rebalancing

2.3.8 Метод Rebalancing_keys

Вход: Node* node — указатель на узел для ребалансировки ключей.

Выход: ключи узлов пересчитаны.

Метод `Rebalancing_keys` пересчитывает ключи в корневом и внутренних узлах дерева, основываясь на первом ключе каждого из потомков. Для каждого узла извлекается первый ключ из его дочернего листа, который затем обновляет ключи текущего узла. В конце метод рекурсивно вызывает себя для всех дочерних узлов.

Код метода представлен в листинге 17.

```

1 void Root::Rebalancing_keys(Node* node) {
2     if (!node) return;
3
4     if (Root* root= dynamic_cast<Root*>(node)) {

```

```

5     if (root->Get_Links().size() > 1) {
6         auto links = node->Get_Links();
7         vector<string> new_words;
8
9         for (int i = 1; i < links.size(); i++) {
10             Node* current_link = links[i];
11             while (true) {
12                 if (Leaf* leaf = dynamic_cast<Leaf*>(current_link))
13                     break;
14                 current_link = current_link->Get_Links()[0];
15             }
16             new_words.push_back(current_link->Get_Words()[0]);
17         }
18         root->Clear_word();
19         root->Set_Words(new_words);
20
21
22         for (auto child : root->Get_Links()) {
23             Rebalancing_keys(child);
24         }
25     }
26 }
27
28 if (InnerNode* inner_node= dynamic_cast<InnerNode*>(node)) {
29     auto links = node->Get_Links();
30     vector<string> new_words;
31
32     for (int i = 1; i < links.size(); i++) {
33         Node* current_link = links[i];
34         while (true) {
35             if (Leaf* leaf = dynamic_cast<Leaf*>(current_link))
36                 break;
37             current_link = current_link->Get_Links()[0];
38         }
39         new_words.push_back(current_link->Get_Words()[0]);
40     }
41     node->Clear_word();
42     node->Set_Words(new_words);
43
44     for (auto child : inner_node->Get_Links()) {
45         Rebalancing_keys(child);
46     }
47 }
48 }

```

Листинг 17. Метод Rebalancing_keys

2.4 Класс InnerNode

Класс InnerNode является реализацией внутреннего узла B+-дерева. Он наследует класс Node. Дополнительных переменных не имеет.

2.4.1 Метод Add_word

Вход: string str — слово для добавления.

Выход: слово добавлено в узел.

Метод Add_word добавляет новое слово в узел. Если узел пуст, слово добавляется в список. Если узел не пуст, слово вставляется в соответствующую позицию, обеспечивая сохранение порядка. После добавления увеличивается счётчик слов в узле.

Код метода представлен в листинге 18.

```
1 void InnerNode::Add_word(string str) {
2     if (words.size() == 0) {
3         words.push_back(str);
4         count_words++;
5     }
6     else {
7         bool flag = true;
8         for (int i = 0; i < words.size(); i++) {
9             if (str < words[i]) {
10                words.insert(words.begin() + i, str);
11                count_words++;
12                flag = false;
13                break;
14            }
15        }
16        if (flag) {
17            words.push_back(str);
18            count_words++;
19        }
20    }
21 }
```

Листинг 18. Метод Add_word для класса InnerNode

2.4.2 Метод Add_word (с добавлением ссылок)

Вход: string str — слово для добавления; vector<Node*> link — ссылки на дочерние узлы.

Выход: слово и ссылки добавлены в узел.

Метод Add_word добавляет новое слово и соответствующие ссылки в узел. Если узел пуст, слово и ссылки добавляются в списки. Если узел не пуст, слово и ссылки вставляются в соответствующие позиции, обеспечивая сохранение порядка. После добавления увеличивается счётчик слов в узле.

Код метода представлен в листинге 19.

```

1 void InnerNode::Add_word(string str, vector<Node*> link) {
2     bool flag_link = false;
3     if (Leaf* buf_leaf = dynamic_cast<Leaf*>(link[0]))
4         flag_link = true;
5     if (words.size() == 0) {
6         words.push_back(str);
7         links.push_back(link[0]);
8         links.push_back(link[1]);
9         count_words++;
10    }
11    else {
12        bool flag = true;
13        for (int i = 0; i < words.size(); i++) {
14            if (str < words[i]) {
15                words.insert(words.begin() + i, str);
16                count_words++;
17                links.erase(links.begin() + i);
18                links.insert(links.begin() + i, link[0]);
19                links.insert(links.begin() + i + 1, link[1]);
20                flag = false;
21                break;
22            }
23        }
24        if (flag) {
25            words.push_back(str);
26            links.pop_back();
27            links.push_back(link[0]);
28            links.push_back(link[1]);
29            count_words++;
30        }
31    }
32    if (flag_link) {
33        for (int i = 0; i < links.size()-1; i++) {
34            links[i]->Clear_Links();
35            links[i]-> Set_Links(links[i + 1]);
36        }
37    }
38 }

```

Листинг 19. Метод Add_word (с добавлением ссылок) для класса InnerNode

2.4.3 Метод Splitting

Вход: string str — слово для обработки.

Выход: vector<Node*> — два новых узла.

Метод Splitting разделяет текущий узел на два новых узла. Первые два слова из текущего

узла добавляются в первый новый узел, а третье слово добавляется во второй новый узел. Если узел содержит ссылки, они распределяются между двумя новыми узлами. Возвращается вектор, содержащий указатели на оба новых узла.

Код метода представлен в листинге 20.

```
1 vector<Node*> InnerNode::Splitting(string str) {
2     InnerNode* node1 = new InnerNode;
3     node1->Add_word(words[0]);
4     node1->Add_word(words[1]);
5
6     InnerNode* node2 = new InnerNode;
7     node2->Add_word(words[3]);
8
9     if (links.size() != 0) {
10         node1->Set_Links(vector<Node*>{links[0], links[1], links[2]});
11         node2->Set_Links(vector<Node*>{links[3], links[4]});
12     }
13     return vector<Node*>{node1, node2};
14 }
```

Листинг 20. Метод Splitting для класса InnerNode

2.4.4 Метод Delete_word

Вход: Node* node — указатель на узел, который нужно удалить.

Выход: — удаляется ссылка на необходимый узел из массива ссылок.

Метод Delete_word удаляет ссылку на указанный узел из текущего узла. Если указанный узел найден в списке ссылок, он удаляется, а остальные ссылки остаются неизменными.

Код метода представлен в листинге 21.

```
1 void InnerNode::Delete_word(Node* node) {
2     for (int i = 0; i < links.size(); i++) {
3         if (links[i] == node) {
4             links.erase(links.begin() + i);
5             break;
6         }
7     }
8 }
```

Листинг 21. Метод Delete_word для класса InnerNode

2.4.5 Метод Rebalancing

Вход: Node* node — указатель на узел, который необходимо перебалансировать.

Выход: дерево сбалансировано.

Метод `Rebalancing` выполняет перебалансировку текущего узла и его дочерних узлов. Если текущий узел является внутренним узлом, он собирает все ссылки на дочерние узлы и перераспределяет их, чтобы обеспечить корректную структуру дерева. В зависимости от количества ссылок и дочерних узлов, метод может объединять узлы или убирать лишние ссылки.

Код метода представлен в листинге 22.

```
1 void InnerNode::Rebalancing(Node* node) {
2     if (!node) return;
3
4     if (InnerNode* inner_node = dynamic_cast<InnerNode*>(node->Get_Links()[0])) {
5         auto current_links = node->Get_Links();
6         int count_links = node->Get_Links().size();
7         vector<Node*> new_links;
8
9         for (auto link : current_links) {
10             auto children_links = link->Get_Links();
11             for (auto child_link : children_links) {
12                 new_links.push_back(child_link);
13             }
14         }
15
16         int count = new_links.size() / count_links;
17
18         if (count > 1) {
19             for (int i = 0; i < current_links.size(); i++) {
20                 current_links[i]->Clear_Links();
21                 if (i != current_links.size() - 1) {
22                     for (int ii = 0; ii < count; ii++) {
23                         current_links[i]->Set_Links(new_links[0]);
24                         new_links[0]->Set_parent_link(current_links[i]); // поправляем ссылку на родит
25
26                         new_links.erase(new_links.begin());
27                     }
28                 }
29                 else {
30                     current_links[i]->Set_Links(new_links);
31                     for (auto link : new_links) {
32                         link->Set_parent_link(current_links[i]);
33                     }
34                 }
35             }
36         }
37         else { //если остается одна ссылка в каком то из узлов -> нужно убрать одну ссылку из р
38             count++;
39             for (int i = 0; i < current_links.size() - 1; i++) {
```

```

40     current_links[i]->Clear_Links();
41     if (i != current_links.size() - 2) {
42         for (int ii = 0; ii < count; ii++) {
43             current_links[i]->Set_Links(new_links[0]);
44             new_links[0]->Set_parent_link(current_links[i]); // поправляем ссылку на родит
45             еля
46             new_links.erase(new_links.begin());
47         }
48     }
49     else {
50         current_links[i]->Set_Links(new_links);
51         for (auto link : new_links) {
52             link->Set_parent_link(current_links[i]);
53         }
54     }
55     links.pop_back();
56 }
57 for (auto child : inner_node->Get_Links()) {
58     Rebalancing(child);
59 }
60 }
61 else if (Root* root = dynamic_cast<Root*>(node)) {
62
63     for (auto child : root->Get_Links()) {
64         Rebalancing(child);
65     }
66 }
67 }

```

Листинг 22. Метод Rebalancing для класса InnerNode

2.4.6 Метод Rebalancing_keys

Вход: Node* node — указатель на узел, для которого нужно перебалансировать ключи.

Выход: — ключи пересчитаны.

Метод **Rebalancing_keys** отвечает за перебалансировку ключей в узле и его дочерних узлах. Если текущий узел является корнем, он собирает ключи из дочерних узлов, используя первый ключ каждого дочернего узла, чтобы обновить свой набор ключей. Если узел является внутренним, метод выполняет аналогичную операцию, обрабатывая ключи от всех дочерних узлов.

Код метода представлен в листинге 23.

```

1 void InnerNode::Rebalancing_keys(Node* node) {
2     if (!node) return;
3
4     if (Root* root = dynamic_cast<Root*>(node)) {
5         auto links = node->Get_Links();

```

```

6     vector<string> new_words;
7
8     for (int i = 1; i < links.size(); i++) {
9         Node* current_link = links[i];
10        while (true) {
11            if (Leaf* leaf = dynamic_cast<Leaf*>(current_link))
12                break;
13            current_link = current_link->Get_Links()[0];
14        }
15        new_words.push_back(current_link->Get_Words()[0]);
16    }
17 }
18
19 if (InnerNode* root = dynamic_cast<InnerNode*>(node)) {
20     auto links = node->Get_Links();
21     vector<string> new_words;
22
23     for (int i = 1; i < links.size(); i++) {
24         Node* current_link = links[i];
25         while (true) {
26             if (Leaf* leaf = dynamic_cast<Leaf*>(current_link))
27                 break;
28             current_link = current_link->Get_Links()[0];
29         }
30         new_words.push_back(current_link->Get_Words()[0]);
31     }
32 }
33 }

```

Листинг 23. Метод Rebalancing_keys для класса InnerNode

2.5 Класс Leaf

Класс Inter является реализацией листового узла B+-дерева. Он наследует класс Node. Этот класс обладает дополнительным полем Node* left_Brother - указатель на лист слева.

2.5.1 Метод Add_word

Вход: string str — слово для добавления.

Выход: слово добавлено в узел, узел теперь содержит это слово.

Метод Add_word добавляет указанное слово **str** в текущий узел. Если узел пустой, слово добавляется как единственное. В противном случае, новое слово вставляется в соответствии с алфавитным порядком. После добавления количество слов в узле увеличивается. Код метода представлен в листинге 24.

```

1 void Leaf::Add_word(string str) {

```

```

2
3  if (words.size() == 0) {
4      words.push_back(str);
5      count_words++;
6  }
7  else {
8      bool flag = true;
9      for (int i = 0; i < words.size(); i++) {
10         if (str < words[i]) {
11             words.insert(words.begin() + i, str);
12             count_words++;
13             flag = false;
14             break;
15         }
16     }
17     if (flag) {
18         words.push_back(str);
19         count_words++;
20     }
21 }
22 }

```

Листинг 24. Метод Add_word (в классе Leaf)

2.5.2 Метод Add_word_without_key (добавление слова без ключа)

Вход: string str — слово для добавления.

Выход: слово добавлено в дерево.

Метод Add_word_without_key добавляет указанное слово **str** в текущий узел. Если узел пустой, слово добавляется в начало. В противном случае слово вставляется в правильное положение, сохраняя порядок.

Код метода представлен в листинге 25.

```

1 void Leaf::Add_word_without_key(string str) {
2     if (words.size() == 0) {
3         words.push_back(str);
4         count_words++;
5     }
6     else {
7         bool flag = true;
8         for (int i = 0; i < words.size(); i++) {
9             if (str < words[i]) {
10                 words.insert(words.begin() + i, str);
11                 count_words++;
12                 flag = false;
13                 break;
14             }

```

```

15     }
16     if (flag) {
17         words.push_back(str);
18         count_words++;
19     }
20 }
21 }

```

Листинг 25. Метод Add_word_without_key (добавление слова без ключа)

2.5.3 Метод Delete_word

Вход: string str — слово для удаления.

Выход: слово удалено из дерева.

Метод Delete_word ищет указанное слово **str** в текущем узле и удаляет его, если оно найдено.

После удаления количество слов в узле уменьшается.

Код метода представлен в листинге 26.

```

1 void Leaf::Delete_word(string str) {
2     for (int i = 0; i < words.size(); i++) {
3         if (str == words[i]) {
4             words.erase(words.begin() + i);
5             count_words--;
6             break;
7         }
8     }
9 }

```

Листинг 26. Метод Delete_word

2.5.4 Метод Splitting

Вход: string str — слово для обработки.

Выход: vector<Node*> — два новых узла.

Метод Splitting добавляет слово **str** в текущий узел, после чего делит его на два новых узла. Первые два слова из текущего узла добавляются в первый новый узел, а третье и четвёртое слова добавляются во второй новый узел. Также устанавливаются ссылки между новыми узлами. Возвращается вектор, содержащий указатели на оба новых узла.

Код метода представлен в листинге 27.

```

1 vector<Node*> Leaf::Splitting(string str) {
2     bool flag = true;
3     for (int i = 0; i < words.size(); i++) {
4         if (str < words[i]) {
5             words.insert(words.begin() + i, str);
6             flag = false;

```



```

7         count_words++;
8         break;
9     }
10 }
11 if (flag) {
12     words.push_back(str);
13     count_words++;
14 }
15
16 Leaf* leaf1 = new Leaf;
17 leaf1->Add_word(words[0]);
18 leaf1->Add_word(words[1]);
19
20 Leaf* leaf2 = new Leaf;
21 leaf2->Add_word(words[2]);
22 leaf2->Add_word(words[3]);
23
24 leaf1->Set_Links(leaf2);
25 leaf2->Set_brother(leaf1);
26
27 return vector<Node*>{leaf1, leaf2};
28 }

```

Листинг 27. Метод Splitting для класса Leaf

2.5.5 Метод Delete_word

Вход: Node* node — указатель на узел для удаления.

Выход: узел удалён из списка ссылок, связи с братом обновлены.

Метод Delete_word удаляет указанный узел node из дерева. Если удаляемый узел является левым братом, обновляется указатель на брата для следующего листа. Если узел является правым, то у него нет ссылок, и в этом случае обновляется указатель на левого брата. Устанавливаются новые связи между узлами в зависимости от ситуации.

Код метода представлен в листинге 24.

```

1 void Leaf::Delete_word(Node* node) {
2     if (this == node) {
3
4         if (left_Brother == nullptr) { //если он левый в дерево, то у след листа указатель на
5             брата будет ноль
6             if (Leaf* leaf = dynamic_cast<Leaf*>(links[0])) {
7                 leaf->Set_brother(nullptr);
8             }
9         }
10        else if (links.size() == 0) { // если он правый в дереве, то у него в ссылках будет но
11            ль ссылок
12            left_Brother->Clear_Links();
13        }
14    }
15 }

```

```

11     }
12     else {
13
14         left_Brother->Clear_Links();
15         left_Brother->Set_Links(this->links[0]);
16
17         if (Leaf* leaf = dynamic_cast<Leaf*>(links[0])) {
18             leaf->Set_brother(this->left_Brother);
19         }
20     }
21 }
22 }

```

Листинг 28. Метод Delete_word для класса Leaf

2.5.6 Метод Is_Sibling

Вход: Node* node — указатель на узел для проверки.

Выход: true, если узел является братом текущего, иначе false.

Метод Is_Sibling проверяет, является ли указанный узел node братом текущего узла. Метод просматривает ссылки родительского узла и возвращает true, если находит совпадение, и false в противном случае.

Код метода представлен в листинге 25.

```

1 bool Leaf::Is_Sibling(Node* node) {
2     auto links = parent_link->Get_Links();
3     for (auto link : links) {
4         if (node == link) {
5             return true;
6         }
7     }
8     return false;
9 }

```

Листинг 29. Метод Is_Sibling для класса Leaf

2.5.7 Метод Rebalancing

Вход: Node* node — указатель на узел для ребалансировки.

Выход: узлы и ссылки ребалансированы в дереве.

Метод Rebalancing осуществляет ребалансировку узлов в дереве, начиная с указанного узла node. Если узел является внутренним узлом, метод распределяет ссылки между дочерними узлами. Если остаётся одна ссылка, она удаляется из родителя. Для корневого узла метод рекурсивно вызывает себя для всех дочерних узлов.

Код метода представлен в листинге 26.

```

1 void Leaf::Rebalancing(Node* node) {
2     if (!node) return;
3
4     if (InnerNode* inner_node = dynamic_cast<InnerNode*>(node->Get_Links()[0])) {
5         auto current_links = node->Get_Links();
6         int count_links = node->Get_Links().size();
7         vector<Node*> new_links;
8
9         for (auto link : current_links) {
10             auto children_links = link->Get_Links();
11             for (auto child_link : children_links) {
12                 new_links.push_back(child_link);
13             }
14         }
15
16         int count = new_links.size() / count_links;
17
18         if (count > 1) {
19             for (int i = 0; i < current_links.size(); i++) {
20                 current_links[i]->Clear_Links();
21                 if (i != current_links.size() - 1) {
22                     for (int ii = 0; ii < count; ii++) {
23                         current_links[i]->Set_Links(new_links[0]);
24                         new_links[0]->Set_parent_link(current_links[i]); // поправляем ссылку на родит
25 еля
26                         new_links.erase(new_links.begin());
27                     }
28                 }
29                 else {
30                     current_links[i]->Set_Links(new_links);
31                     for (auto link : new_links) {
32                         link->Set_parent_link(current_links[i]);
33                     }
34                 }
35             }
36         }
37         else { //если остается одна ссылка в каком то из узлов -> нужно убрать одну ссылку из р
38 одителя
39             count++;
40             for (int i = 0; i < current_links.size() - 1; i++) {
41                 current_links[i]->Clear_Links();
42                 if (i != current_links.size() - 2) {
43                     for (int ii = 0; ii < count; ii++) {
44                         current_links[i]->Set_Links(new_links[0]);
45                         new_links[0]->Set_parent_link(current_links[i]); // поправляем ссылку на родит
46 еля

```

```

45         new_links.erase(new_links.begin());
46     }
47 }
48 else {
49     current_links[i]->Set_Links(new_links);
50     for (auto link : new_links) {
51         link->Set_parent_link(current_links[i]);
52     }
53 }
54 }
55 links.pop_back();
56 }
57 for (auto child : inner_node->Get_Links()) {
58     Rebalancing(child);
59 }
60 }
61 else if (Root* root = dynamic_cast<Root*>(node)) {
62
63     for (auto child : root->Get_Links()) {
64         Rebalancing(child);
65     }
66 }
67 }

```

Листинг 30. Метод Rebalancing для класса Leaf

2.5.8 Метод Rebalancing_keys

Вход: Node* node — указатель на узел для перебалансировки ключей.

Выход: ключи обновлены в узлах дерева.

Метод **Rebalancing_keys** обновляет ключи в узлах дерева, начиная с указанного узла **node**. Если узел является корнем, метод собирает ключи из дочерних узлов и обновляет список ключей корня. Если узел является внутренним, процесс аналогичен: ключи собираются из дочерних узлов и обновляются соответственно.

Код метода представлен в листинге 27.

```

1 void Leaf::Rebalancing_keys(Node* node) {
2     if (!node) return;
3
4     if (Root* root = dynamic_cast<Root*>(node)) {
5         auto links = node->Get_Links();
6         vector<string> new_words;
7
8         for (int i = 1; i < links.size(); i++) {
9             Node* current_link = links[i];
10            while (true) {
11                if (Leaf* leaf = dynamic_cast<Leaf*>(current_link))

```

```

12         break;
13         current_link = current_link->Get_Links()[0];
14     }
15     new_words.push_back(current_link->Get_Words()[0]);
16 }
17 }
18
19 if (InnerNode* root = dynamic_cast<InnerNode*>(node)) {
20     auto links = node->Get_Links();
21     vector<string> new_words;
22
23     for (int i = 1; i < links.size(); i++) {
24         Node* current_link = links[i];
25         while (true) {
26             if (Leaf* leaf = dynamic_cast<Leaf*>(current_link))
27                 break;
28             current_link = current_link->Get_Links()[0];
29         }
30         new_words.push_back(current_link->Get_Words()[0]);
31     }
32 }
33 }

```

Листинг 31. Метод `Rebalancing_keys` для класса `Leaf`

2.6 Класс `Bplus`

Класс `Bplus` является реализацией `B+`-дерева. Он обладает следующими полями:

1. `Root* root` — ссылка на корневой узел;
2. `int DEGREE` — степень дерева;
3. `int levels` — количество уровней;
4. `vector<string> conj` — массив слов-исключений;

2.6.1 Метод `is_a_conj`

Вход: `string str` — слово для проверки.

Выход: `bool` — возвращает `true`, если слово является исключением, иначе `false`.

Метод `is_a_conj` проверяет, является ли переданное слово исключением. Слово приводится к нижнему регистру, затем ищется в списке исключений `conj`. Если слово найдено, возвращается `true`, в противном случае — `false`.

Код метода представлен в листинге 28.

```

1 bool Tree::is_a_conj(string str) {
2     str = toDown(str);

```

```

3   for (string word : conj) {
4       if (str == word) {
5           return true;
6       }
7   }
8   return false;
9 }

```

Листинг 32. Метод `is_a_conj` для класса `Tree`

2.6.2 Метод `Add_word`

Вход: `string str` — слово для добавления в дерево.

Выход: слово добавлено в дерево.

Метод `Add_word` добавляет новое слово в дерево. Если в корневом узле нет ссылок (первый случай), и количество слов в корне меньше, чем $2 \cdot DEGREE - 1$, слово добавляется в корень. Если корень переполнен, он разделяется на новый корневой узел, а количество уровней увеличивается. В остальных случаях слово добавляется в дерево с последующей ребалансировкой ключей.

Код метода представлен в листинге 29.

```

1 void Tree::Add_word(string str) {
2
3     if (root->Get_Links().size() == 0) { //первый случай
4
5         if (root->Get_Count_words() < (2 * DEGREE - 1)) {
6             root->Add_word(str);
7         }
8     } else {
9         auto actual_words = root->Get_Words();
10        bool flag = true;
11        for (int i = 0; i < actual_words.size(); i++) {
12            if (str < actual_words[i]) {
13                actual_words.insert(actual_words.begin() + i, str);
14                flag = false;
15                break;
16            }
17        }
18        if (flag) {
19            actual_words.push_back(str);
20        }
21        Root* new_root = new Root;
22        delete root;
23        root = new_root;
24        root->Set_New_Root(actual_words);
25        levels++;
26    }
27 }

```

```

28 //остальные случаи
29 else {
30     root->Add_word(str);
31     root->Rebalancing_keys(root);
32 }
33 }

```

Листинг 33. Метод Add_word для класса Tree

2.6.3 Метод Delete_word

Вход: string str — слово для удаления из дерева.

Выход: слово удалено из дерева, дерево ребалансировано.

Метод Delete_word удаляет указанное слово из дерева. Если слово найдено с помощью метода Find_word, оно удаляется из соответствующего узла. После удаления вызывается метод Rebalancing для поддержания сбалансированности дерева.

Код метода представлен в листинге 30.

```

1 void Tree::Delete_word(string str) {
2     if (this->Find_word(str)) {
3         auto edited_node = root->Delete_word(str);
4         Rebalancing(edited_node);
5     }
6 }

```

Листинг 34. Метод Delete_word для класса Tree

2.6.4 Метод Find_word

Вход: string str — слово для поиска в дереве.

Выход: bool — true, если слово найдено; иначе false.

Метод Find_word выполняет поиск указанного слова в дереве. Если корневой узел не содержит ссылок, поиск осуществляется непосредственно в его словах. В противном случае происходит обход по ссылкам в зависимости от значений ключей, и поиск выполняется в листьях. При необходимости проверяется правый брат узла, если он существует.

Код метода представлен в листинге 31.

```

1 bool Tree::Find_word(string str) {
2     if (root->Get_Links().size() == 0) {
3         for (auto word : root->Get_Words()) {
4             if (word == str)
5                 return true;
6         }
7     }
8     else {
9         Node* current_link = root;

```

```

10 while (true) {
11     if (Leaf* buf_leaf = dynamic_cast<Leaf*>(current_link))
12         break;
13     vector<string> keys = current_link->Get_Words();
14     int index = 0;
15     for (int i = 0; i < keys.size(); i++) {
16         if (str >= keys[i])
17             index = i + 1;
18     }
19     current_link = current_link->Get_Links()[index];
20 }
21 for (auto word : current_link->Get_Words()) {
22     if (str == word)
23         return true;
24 }
25 //если не нашлось на всякий проверим брата справа, вдруг он там
26 if (current_link->Get_Links().size() > 0) {
27     current_link = current_link->Get_Links()[0];
28     for (auto word : current_link->Get_Words()) {
29         if (str == word)
30             return true;
31     }
32 }
33
34 }
35 return false;
36 }

```

Листинг 35. Метод Find_word для класса Tree

2.6.5 Метод From_file

Вход: string str — путь к файлу для обработки.

Выход: слова из файла добавлены в дерево, если они не являются исключениями.

Метод From_file загружает содержимое текстового файла, путь к которому передаётся в параметре str. Метод обрабатывает символы текста, игнорируя знаки препинания, и извлекает слова. Если слово не является исключением, оно добавляется в дерево с помощью метода Add_word.

Код метода представлен в листинге 32.

```

1 void Tree::From_file(string str) {
2     //std::filesystem::path file_path = str;
3     //ifstream file(R"(str)");
4     ifstream file(str);
5     if (!file.is_open()) {
6         printf("\nНе удалось открыть файл\n\n");
7     }
8     else {

```



```

9     string line;
10    vector<string> words = { "" };
11    int last_index = 0;
12    char c;
13    int k = 0;
14    while (file.get(c)) {
15        if (c != ',' && c != '.' && c != ':' && c != ';' && c != '!' && c != '?' && c != '-'
16            && c != '?')
17            && c != '(' && c != ')' && c != '{' && c != '}' && c != '[' && c != ']' && c != '\',
18
19            && c != '"') {
20            if (c != ' ' && c != '\n') {
21                words[last_index] += c;
22            }
23            else {
24                if (words[last_index].size() > 0) {
25                    words.push_back("");
26                    last_index++;
27                }
28            }
29        }
30
31        if (words[words.size() - 1] == "") {
32            words.pop_back();
33        }
34
35        file.close();
36        for (auto word : words) {
37            if (!this->is_a_conj(word)) {
38                this->Add_word(word);
39            }
40        }
41
42        printf("\nСлова из файла добавлены в словарь\n\n");
43    }
44 }

```

Листинг 36. Метод From_file для класса Tree

2.6.6 Метод Print

Вход: Node* node — указатель на узел для печати; int level — текущий уровень узла.

Выход: структура дерева напечатана в консоль с отступами по уровням.

Метод Print выводит дерево на консоль, начиная с узла, переданного в параметре node, и продолжает печать рекурсивно для всех дочерних узлов. Уровень узла определяется параметром

level, который используется для форматирования отступов. Листовые узлы выводят свои слова, внутренние узлы выводят слова со звёздочкой (*), а корневой узел также учитывается при выводе.

Код метода представлен в листинге 33.

```
1 void Tree::Print(Node* node, int level) {
2     if (!node) return;
3
4     if (Leaf* leaf = dynamic_cast<Leaf*>(node)) {
5         cout << setw(level * 4) << " ";
6         for (const auto& words : leaf->Get_Words()) {
7             cout << words << " ";
8         }
9         cout << endl;
10    }
11    else if (InnerNode* inner_node = dynamic_cast<InnerNode*>(node)) {
12        cout << setw(level * 4) << " ";
13        for (const auto& words : inner_node->Get_Words()) {
14            cout << words << "* ";
15        }
16        cout << endl;
17        for (auto child : inner_node->Get_Links()) {
18            Print(child, level + 1);
19        }
20    }
21    else if (Root* root = dynamic_cast<Root*>(node)) {
22        if (root->Get_Links().size() == 0 && root->Get_Count_words() == 0) {
23            cout << "Словарь пуст" << " ";
24        }
25        else {
26            cout << setw(level * 4) << " ";
27
28            for (const auto& words : root->Get_Words()) {
29                if (root->Get_Links().size() > 1) {
30                    cout << words << "* ";
31                }
32                else {
33                    cout << words << " ";
34                }
35            }
36        }
37        cout << endl;
38        for (auto child : root->Get_Links()) {
39            Print(child, level + 1);
40        }
41    }
42 }
```

Листинг 37. Метод Print для класса Tree

2.6.7 Метод Delete_all

Вход: дерево с данными.

Выход: все узлы дерева удалены, корневой узел сброшен до нового пустого состояния.

Метод `Delete_all` очищает всё дерево, удаляя все узлы и устанавливая корневой узел (`root`) в новое пустое состояние.

Код метода представлен в листинге 34.

```
1 void Tree::Delete_all() {
2     Root* buf = new Root;
3     root->Clear(root);
4     root = buf;
5 }
```

Листинг 38. Метод `Delete_all` для класса `Tree`

2.6.8 Метод Rebalancing

Вход: `Node* edited_leaf` — указатель на редактируемый лист.

Выход: сбалансированы узлы дерева после удаления слова, изменены связи и распределены слова между узлами.

Метод `Rebalancing` проверяет количество слов в узле после удаления и при необходимости перераспределяет слова и ссылки между узлами для поддержания сбалансированного состояния дерева. Если количество слов в узле равно нулю, узел удаляется, а родительский узел корректируется. Далее происходит корректировка структуры дерева и перебалансировка ключей.

Код метода представлен в листинге 35.

```
1 void Tree::Rebalancing(Node* edited_leaf) {
2     if (Root* edited_root = dynamic_cast<Root*>(edited_leaf)) {
3         return;
4     }
5     else {
6         if (edited_leaf->Get_Count_words() == 0) {
7             edited_leaf->Delete_word(edited_leaf);
8             edited_leaf->Get_Parent_link()->Delete_word(edited_leaf);
9         }
10
11         Node * current_link = root;
12         Node* previous_link = nullptr;
13         Node* first_leaf = nullptr;
14         vector<Node*> links;
15         vector<string> new_words;
16
17         //проходит по дереву до листа
18         while (true) {
19             if (Leaf* buf_leaf = dynamic_cast<Leaf*>(current_link))
```

```

20     break;
21     previous_link = current_link;
22     current_link = current_link->Get_Links()[0];
23 }
24 while (true) {
25     links.push_back(current_link);
26     auto words = current_link->Get_Words();
27     for (auto word : words) {
28         new_words.push_back(word);
29     }
30
31     if (current_link->Get_Links().size() == 0)
32         break;
33     current_link = current_link->Get_Links()[0];
34 }
35
36 int count = new_words.size() / links.size();
37
38
39 if (count > 1) {
40     for (int i = 0; i < links.size(); i++) {
41         links[i]->Clear_word();
42         if (i != links.size() - 1) {
43             for (int ii = 0; ii < count; ii++) {
44                 links[i]->Add_word(new_words[0]);
45                 new_words.erase(new_words.begin());
46             }
47         }
48         else {
49             links[i]->Set_Words(new_words);
50         }
51     }
52 }
53
54 else { //если остается одна ссылка в каком то из узлов -> нужно убрать одну ссылку из
родителя
55     count++;
56
57     if (new_words.size() < 4) {
58
59     }
60     for (int i = 0; i < links.size() - 1; i++) {
61         links[i]->Clear_word();
62         if (i != links.size() - 2) {
63             for (int ii = 0; ii < count; ii++) {
64                 links[i]->Add_word(new_words[0]);
65                 new_words.erase(new_words.begin());
66             }

```

```

67     }
68     else {
69         links[i]->Clear_word();
70         links[i]->Set_Words(new_words);
71     }
72 }
73 Node* link_for_deleteing = links[links.size() - 1];
74 link_for_deleteing->Get_Parent_link()->Delete_word(link_for_deleteing);
75 links[links.size()-2]->Clear_Links();
76 links.pop_back();
77
78 }
79 root->Rebalancing(root);
80 root->Rebalancing_keys(root);
81 }
82 }

```

Листинг 39. Метод Rebalancing для класса Tree

2.6.9 Метод Menu

Вход: создание консольного меню, для работы с B+-деревом.

Выход: консольное меню для работы с B+-деревом.

Метод **Menu** отвечает за взаимодействие пользователя с приложением. В меню можно выбрать различные действия по работе со словарём, такие как просмотр содержимого, добавление или удаление слов, проверка их наличия, загрузка из файла и очистка словаря.

Код метода представлен в листинге 36.

```

1 void Tree::Menu() {
2     while (true) {
3         printf("\n----- Словарь (B+ дерево)-----\n");
4         printf("Выберите действие:\n");
5         printf("[1] - просмотр содержимое словаря \n");
6         printf("[2] - добавление нового слова\n");
7         printf("[3] - удаление слова\n");
8         printf("[4] - проверка наличие слов в словаре\n");
9         printf("[5] - полная очистка словаря\n");
10        printf("[6] - добавление слов из файла\n");
11        printf("[7] - просмотр список недобавляющихся слов\n");
12
13        printf("[0] - выход из словаря\n");
14        printf("-----\n");
15
16        int choose;
17        bool out = false;
18        string word;
19        string file;

```

```

20 //vector<string> files = { "Master and Margarita.txt", "Crime and punishment.txt" };
21 vector<string> files = { "Мастер и Маргарита.txt", "Преступление и наказание.txt" };
22 int k;
23 bool flag_out_path;
24
25 while (true) {
26
27     cin >> choose;
28
29     // Проверка на корректность ввода
30     if (std::cin.fail()) {
31         std::cin.clear(); // Сброс состояния потока
32         std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Очистка бу
33    фера ввода
34     std::cout << "Некорректный ввод! Пожалуйста, введите число." << std::endl;
35 }
36 else if (choose < 0 || choose > 7) {
37     // Проверка, что число находится в нужном диапазоне
38     std::cout << "Число вне диапазона! Пожалуйста, введите число от 0 до 7." << std::
39 endl;
40 }
41 else {
42     break; // Выход из цикла, если ввод корректен и число в диапазоне
43 }
44 }
45
46 switch (choose)
47 {
48     case 0:
49         out = true;
50         break;
51     case 1:
52         if (root->Get_Count_words() == 0) {
53             printf("\nСловарь пуст\n\n");
54         }
55         else {
56             printf("\nСодержимое словаря:\n\n");
57             Print(root);
58             cout << endl;
59         }
60     }
61     break;
62     case 2:
63         while (true) {
64             printf("Введите слово для добавления в словарь (для выхода в меню напишите '0')
65 ");
66             cin >> word;
67             if (word == "0") {
68                 break;

```

```

65     }
66     else {
67         if (root->Get_Count_words() != 0) {
68             if (!Find_word(word)) {
69                 if (!is_a_conj(word)) {
70                     this->Add_word(word);
71                     printf("\n\tСлово '%s' добавлено в словарь\n\n", word.c_str());
72                     //break;
73                 }
74                 else {
75                     printf("\n\tСлово '%s' не добавлено в словарь, так как оно входит в списо
к недобавляющихся слов :(\n\n", word.c_str());
76                 }
77             }
78             else {
79                 printf("\n\tСлово '%s' не добавлено в словарь, оно уже там присутствует\n\n"
, word.c_str());
80             }
81         }
82         else {
83             if (!is_a_conj(word)) {
84                 this->Add_word(word);
85                 printf("\n\tСлово '%s' добавлено в словарь\n\n", word.c_str());
86             }
87         }
88     }
89 };
90 break;
91
92 case 3:
93
94 while (true) {
95     printf("введите слово, которое хотите удалить (для выхода в меню напишите '0'): "
);
96     cin >> word;
97     if (word == "0") {
98         break;
99     }
100    else {
101        if (root->Get_Count_words() != 0) {
102
103            if (!this->Find_word(word)) {
104                printf("\n\tСлово '%s' отсутствует\n\n", word.c_str());
105            }
106            else {
107                this->Delete_word(word);
108                printf("\n\tСлово '%s' удалено\n\n", word.c_str());
109            }

```

```

110     }
111     else {
112         printf("\nСловарь пуст\n\n");
113     }
114 }
115 }
116 break;
117
118 case 4:
119 while (true) {
120     printf("Введите слово, наличие корого хотите проверить (для выхода в меню напишит
e '0'): ");
121     cin >> word;
122     if (word == "0") {
123         break;
124     }
125     else {
126         if (root->Get_Count_words() != 0) {
127             if (this->Find_word(word)) {
128                 printf("\n\tСлово '%s' присутствует в словаре\n\n", word.c_str());
129             }
130             else {
131                 printf("\n\tСлово '%s' отсутствует в словаре\n\n", word.c_str());
132             }
133             //break;
134         }
135         else {
136             printf("\nСловарь пуст\n\n");
137         }
138     }
139 }
140 break;
141
142 case 5:
143 this->Delete_all();
144 printf("\nСловарь полностью очищен\n\n");
145 break;
146
147 case 6:
148 printf("\nВыберите файл из существующих:\n");
149 printf("1. %s\n", files[0].c_str());
150 printf("2. %s\n", files[1].c_str());
151 printf("3. другое...\n\n");
152 printf("0. ВЫХОД\n\n");
153 printf("Ваш выбор: ");
154
155 flag_out_path = true;
156

```



```

157     while (true) {
158         cin >> k;
159         // Проверка на корректность ввода
160         if (std::cin.fail()) {
161             std::cin.clear(); // Сброс состояния потока
162             std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Очистка
буфера ввода
163             std::cout << "Некорректный ввод! Пожалуйста, введите число." << std::endl;
164         }
165         else if (k < 0 || k > 3) {
166             // Проверка, что число находится в нужном диапазоне
167             std::cout << "Число вне диапазона! Пожалуйста, введите число от 0 до 3." << std
::endl;
168         }
169         else {
170             break; // Выход из цикла, если ввод корректен и число в диапазоне
171         }
172     }
173     switch (k) {
174         case 1:
175             //file = "D:/Another/CLion Projects/txt files/" + files[0];
176             file = files[0];
177             break;
178         case 2:
179             //file = "D:/Another/CLion Projects/txt files/" + files[1];
180             file = files[1];
181             break;
182         case 3:
183             printf("напишите название своего файла с '.txt': ");
184             cin >> file;
185             break;
186         case 0:
187             flag_out_path = false;
188             break;
189     }
190     if (flag_out_path)
191         this->From_file(file);
192     break;
193
194     case 7:
195         printf("Список недобавляющихся слов\n");
196
197         for (int i = 0; i < this->conj.size(); i++) {
198             printf("%d. %s\n", i + 1, this->conj[i].c_str());
199         }
200         cout << "\n";
201     }
202

```

```
203     if (out) {  
204         break;  
205     }  
206     system("pause");  
207 }  
208  
209 }
```

Листинг 40. Метод Menu для класса Tree

3 Результаты работы программы

При запуске программу пользователю выводится меню с выбором структуры данных (рис.5).

```
----- Структуры данных -----  
Сделайте свой выбор:  
[1] - Хэш-таблицв  
[2] - В+ дерево  
[0] - выход из программы  
-----
```

Рис. 5. Меню выбора структуры данных

При выборе хеш-таблицы, то есть нажав на клавишу «1», выводится меню для работы с хеш-таблицей (рис.6).

```
----- Словарь (хэш-таблица)-----  
Выберите действие:  
[1] - просмотр содержимое словаря  
[2] - добавление нового слова  
[3] - удаление слова  
[4] - проверка наличие слов в словаре  
[5] - полная очистка словаря  
[6] - добавление слов из файла  
[7] - просмотр список недобавляющихся слов  
[0] - выход из словаря  
-----
```

Рис. 6. Меню для работы с хеш-таблицей

При нажатии клавиши «1», в самом начале, когда словарь пуст, пользователю будет выведено сообщение о пустом словаре (рис.7).

```
1  
  
Словарь пуст
```

Рис. 7. Сообщение о пустом словаре

При нажатии клавиши «2», пользователю предлагается вводить слова, которые он хочет добавлять в словарь (рис.8).

```
2
Введите слово для добавления в словарь (для выхода в меню напишите '0'): капуста
    Слово 'капуста' добавлено в словарь
Введите слово для добавления в словарь (для выхода в меню напишите '0'):
```

Рис. 8. Добавление слов в словарь

При нажатии клавиши «1» и наличии в словаре слов, пользователю предлагается вывод в разных форматах: по бакетам и списком (рис.9)

```
Содержимое словаря:
[1] - по бакетам
[2] - списком
```

```
Как вы хотите? Введите свой выбор: |
```

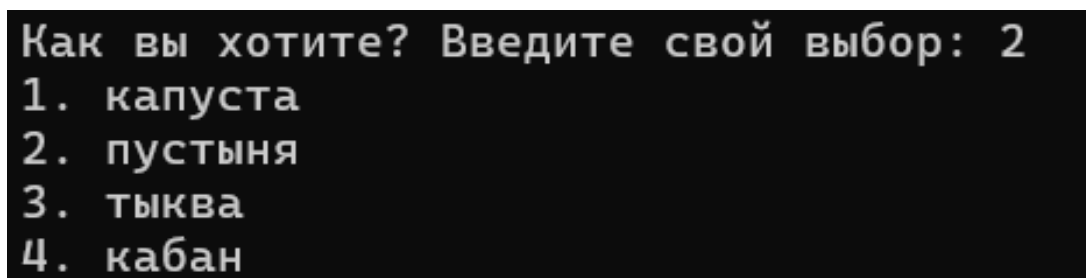
Рис. 9. Предложение выбора вывода словаря

При нажатии на «1», слова выводятся по бакетам (рис.10)

```
Как вы хотите? Введите свой выбор: 1
[0]: капуста
[1]:
[2]:
[3]: пустыня
[4]: тыква
[5]: кабан
```

Рис. 10. Вывод хеш-таблицы по бакетам

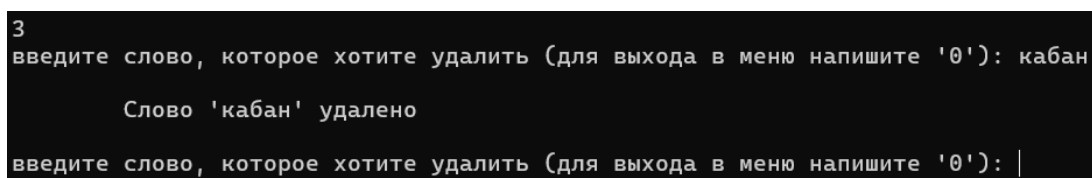
Иначе, если нажать «2», слова выведутся одним списком (рис.11)



```
Как вы хотите? Введите свой выбор: 2
1. капуста
2. пустыня
3. тыква
4. кабан
```

Рис. 11. Вывод словаря списком

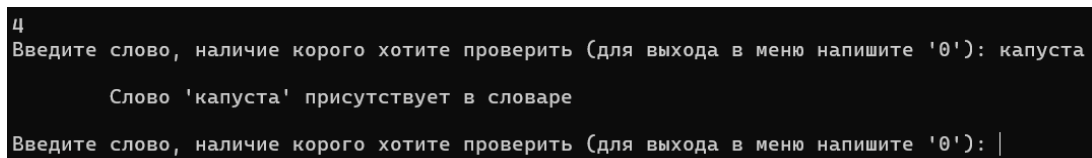
При нажатии клавиши «3», пользователю предлагается вводить слова, которые он хочет удалить из словаря (рис.12).



```
3
введите слово, которое хотите удалить (для выхода в меню напишите '0'): кабан
Слово 'кабан' удалено
введите слово, которое хотите удалить (для выхода в меню напишите '0'): |
```

Рис. 12. Удаление слова из словаря

При нажатии клавиши «4», пользователю предлагается вводить слова по одному, наличие которых он хочет проверить в словаре (рис.13).



```
4
Введите слово, наличие которого хотите проверить (для выхода в меню напишите '0'): капуста
Слово 'капуста' присутствует в словаре
Введите слово, наличие которого хотите проверить (для выхода в меню напишите '0'): |
```

Рис. 13. Проверка наличия слова в словаре

При нажатии клавиши «5» происходит полная очистка словаря (рис.14)

```
5  
Словарь полностью очищен
```

Рис. 14. Полная очистка словаря

При нажатии клавиши «6» предлагается выбрать файл, из которого будут добавляться слова в хеш-таблицу. Пользователь может выбрать файл из существующих либо выбрать свой файл (рис.15)

```
6  
  
Выберите файл из существующих:  
1. Мастер и Маргарита.txt  
2. Преступление и наказание.txt  
3. другое...  
  
0. ВЫХОД
```

Рис. 15. Выбор файла, слова из которого будут добавлены в словарь

Допустим пользователь выбирает файл «Мастер и Маргарита.txt», после выводится сообщение о том, что слова из файла добавлены в словарь (рис.16)

```
Ваш выбор: 1  
  
Слова из файла добавлены в словарь
```

Рис. 16. Сообщение о том, что слова из файла добавлены

При нажатии клавиши «7» пользователю выводится список слов-исключений, которые не будут добавляться в словарь (рис.17)

```
7
Список недобавляющихся слов
1. и
2. да
3. ни
4. также
5. тоже
6. а
7. но
8. однако
```

Рис. 17. Список слов-исключений

В самом начале при выборе структуры данных, если пользователь нажмет «2», то есть выберет В+-дерево, то выведется меню для работы с В+-деревом (рис.18)

```
----- Словарь (В+ дерево)-----
Выберите действие:
[1] - просмотр содержимое словаря
[2] - добавление нового слова
[3] - удаление слова
[4] - проверка наличие слов в словаре
[5] - полная очистка словаря
[6] - добавление слов из файла
[7] - просмотр список недобавляющихся слов
[0] - выход из словаря
-----
```

Рис. 18. Меню для работы с В+-деревом

При нажатии клавиши «1», в самом начале, когда словарь пуст, пользователю будет выведено сообщение о пустом словаре (рис.19).

```
1
Словарь пуст
```

Рис. 19. Сообщение о пустом словаре

При нажатии клавиши «2», пользователю предлагается вводить слова, которые он хочет добавлять в словарь (рис.20).

```
2
Введите слово для добавления в словарь (для выхода в меню напишите '0'): капуста
    Слово 'капуста' добавлено в словарь
Введите слово для добавления в словарь (для выхода в меню напишите '0'):
```

Рис. 20. Добавление слов в словарь

При нажатии клавиши «1» и наличии в словаре слов, пользователю предлагается вывод в разных форматах: по бакетам и списком (рис.21)

```
1
Содержимое словаря:
монитор* путешествие*
    капуста* лестница*
        алфавит будущее
        капуста колонка
        лестница ложка магазин
    программа*
        монитор оценка
        программа пульт
    стакан*
        путешествие словарь
        стакан трамвай
```

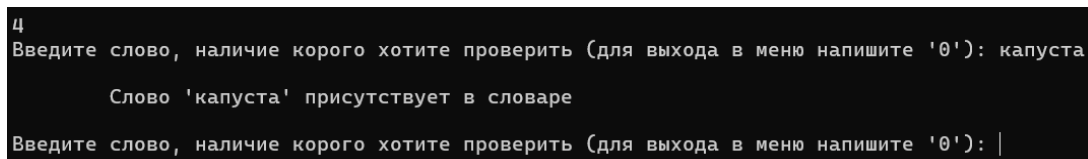
Рис. 21. Вывод словаря в виде дерева

При нажатии клавиши «3», пользователю предлагается вводить слова, которые он хочет удалить из словаря (рис.22).

```
3
введите слово, которое хотите удалить (для выхода в меню напишите '0'): стакан
    Слово 'стакан' удалено
введите слово, которое хотите удалить (для выхода в меню напишите '0'): |
```

Рис. 22. Удаление слова из словаря

При нажатии клавиши «4», пользователю предлагается вводить слова по одному, наличие которых он хочет проверить в словаре (рис.23).



```
4
Введите слово, наличие которого хотите проверить (для выхода в меню напишите '0'): капуста
      Слово 'капуста' присутствует в словаре
Введите слово, наличие которого хотите проверить (для выхода в меню напишите '0'): |
```

Рис. 23. Проверка наличия слова в словаре

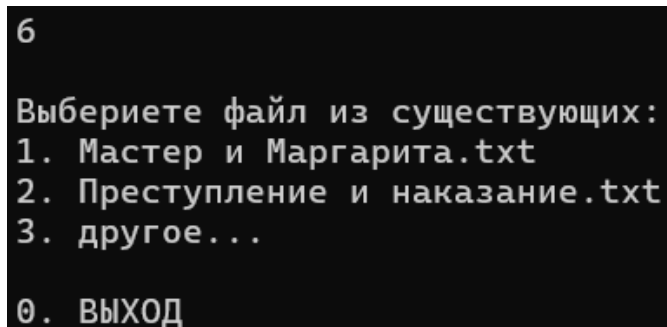
При нажатии клавиши «5» происходит полная очистка словаря (рис.24)



```
5
Словарь полностью очищен
```

Рис. 24. Полная очистка словаря

При нажатии клавиши «6» предлагается выбрать файл, из которого будут добавляться слова в В+-дерево. Пользователь может выбрать файл из существующих либо выбрать свой файл (рис.25)



```
6
Выберите файл из существующих:
1. Мастер и Маргарита.txt
2. Преступление и наказание.txt
3. другое...
0. ВЫХОД
```

Рис. 25. Выбор файла, слова из которого будут добавлены в словарь

Допустим пользователь выбирает файл «Мастер и Маргарита.txt», после выводится сообщение о том, что слова из файла добавлены в словарь (рис.26)

```
Ваш выбор: 1
Слова из файла добавлены в словарь
```

Рис. 26. Сообщение о том, что слова из файла добавлены

При нажатии клавиши «7» пользователю выводится список слов-исключений, которые не будут добавляться в словарь (рис.27)

```
7
Список недобавляющихся слов
1. и
2. да
3. ни
4. также
5. тоже
6. а
7. но
8. однако
```

Рис. 27. Список слов-исключений

При попытке пользователя ввести некорректное значение, программа выдает об этом сообщение (рис.28)

```
----- Словарь (В+ дерево)-----
Выберите действие:
[1] - просмотр содержимое словаря
[2] - добавление нового слова
[3] - удаление слова
[4] - проверка наличие слов в словаре
[5] - полная очистка словаря
[6] - добавление слов из файла
[7] - просмотр список недобавляющихся слов
[0] - выход из словаря
-----
апвыв
Некорректный ввод! Пожалуйста, введите число.
```

Рис. 28. Сообщение о неправильном вводе

Заключение

В ходе выполнения лабораторной работы был реализован словарь в двух вариантах: хеш-таблицей и В+-деревом. Для каждого из вариантов словаря были реализованы операции: добавления, удаления и поиска слова, полной очистки словаря и добавления слов в словарь из текстового файла. Для разрешения коллизий в хеш-таблице был использован метод цепочек. В хеш-функции хеш слова вычислялся при помощи полинома.

Хеш-функция, используемая в хеш-таблице генерирует значения на всем диапазоне переменной `int`. Чтобы уменьшить количество коллизий была введена переменная - **коэффициент заполнения**, которая представляет из себя отношение количества элементов в хеш-таблице к общему количеству бакетов. Когда коэффициент заполнения превышает определённый порог (0.75), происходит **рехеширование** — процесс, при котором создаётся новый массив большего размера с удвоенным количеством бакетов, и все существующие пары из старого массива переносятся в новый, с пересчётом индексов на основе новой хеш-функции.

Из достоинств реализованной программы можно назвать наличие функции рехеширования, которая предотвращает появление большого числа коллизий при увеличении количества добавляемых слов, и рассмотрение всех ситуаций при реализации операций добавления и удаления слова из словаря, основанного на В+-дереве.

Недостатками реализованной программы можно выделить повторения кода в программе, однотипность хранимых данных, а также то что, В+-дерево в данной программе степени два

В планы на масштабирование программы можно записать следующее:

- Реализация поиска однокоренных слов в хеш-таблице.
- Возможность создания В+-дерева любой, допустимой его свойствам, степени.
- Возможность хранения любого типа данных данных структурах.

Список литературы

- [1] Секция «Телематика». - Текст: электронный // tema.spbstu.ru : [сайт]. - URL: <https://tema.spbstu.ru/tgraph/> (дата обращения 12.05.2024).
- [2] Новиков, Ф.А. ДИСКРЕТНАЯ МАТЕМАТИКА ДЛЯ ПРОГРАММИСТОВ Ф.А. Новиков. - 3-е издание. - Питер : Питер Пресс, 2009. - 384 с.