

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: \_\_\_\_\_

Шихалев А.О.

Руководитель: \_\_\_\_\_

Востров А.В.

«\_\_\_\_\_» \_\_\_\_\_ 20\_\_\_\_ г.

Санкт-Петербург, 2024

# Содержание

<b>Введение</b>	<b>4</b>
<b>1 Математическое описание</b>	<b>5</b>
1.1 Определение хеш-функции	5
1.2 Полиномиальное хеширование	5
1.3 Хеш-таблица	6
1.3.1 Разрешение коллизий методом цепочек	7
1.4 В+-дерево	8
1.5 Операции над В+-деревом	9
1.5.1 Поиск	9
1.5.2 Добавление	10
1.5.3 Удаление	10
1.5.4 Пример Б+-дерева	10
<b>2 Особенности реализации</b>	<b>12</b>
2.1 Структура Node	12
2.2 Класс LinkedList	12
2.2.1 Метод insert	12
2.2.2 Метод remove	12
2.2.3 Метод clear	13
2.2.4 Метод search	14
2.3 Класс HashTable	15
2.3.1 Метод hashFunction	15
2.3.2 Метод insert	15
2.3.3 Метод remove	16
2.3.4 Метод clear	17
2.3.5 Метод search	17
2.3.6 Метод insertFromFile	18
2.3.7 Метод rehash	19
2.4 Класс Node	20
2.4.1 Метод GetBrother	20
2.5 Класс LeafNode	21
2.5.1 Метод Split	21
2.5.2 Метод Insert	21
2.5.3 Метод Delete	22
2.5.4 Метод Merge	23
2.6 Класс InterNode	23
2.6.1 Метод Merge	23

2.6.2	Метод Insert . . . . .	24
2.6.3	Метод Split . . . . .	24
2.6.4	Метод Delete . . . . .	25
2.6.5	Метод Slib . . . . .	26
2.7	Класс Bplus . . . . .	27
2.7.1	Метод Find . . . . .	27
2.7.2	Метод Add_Node . . . . .	27
2.7.3	Метод Search . . . . .	28
2.7.4	Метод Insert . . . . .	29
2.7.5	Метод Delete . . . . .	31
2.7.6	Метод Remove_Node . . . . .	33
2.7.7	Метод loadFromFile . . . . .	35
2.7.8	Метод clear . . . . .	36
<b>3</b>	<b>Результаты работы программы</b>	<b>38</b>
	<b>Заключение</b>	<b>42</b>
	<b>Список использованной литературы</b>	<b>43</b>

## Введение

Отчет посвящен разработке приложения-словаря, в котором были реализованы следующие операции: добавление, удаление, поиск, очистка словаря, а также дополнение словаря из текстового файла. Для хранения данных были взяты хэш-таблица и В<sup>+</sup>-дерево.

Работа была выполнена в среде Visual Studio 2022 на языке программирования C++.

# 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. Различные методы хеширования отличаются способами разрешения коллизий (например, метод цепочек, открытая адресация) и способами построения самой хеш-функции (например, метод деления, умножения, криптографические хеш-функции и др.).

## 1.2 Полиномиальное хеширование

**Полиномиальное хеширование** — это метод вычисления хеш-значения строки, где каждый символ строки рассматривается как коэффициент одночлена в полиноме, а итоговое значение полинома используется для вычисления хеша. Формально, строка  $S = s_0 s_1 s_2 \dots s_{n-1}$  хешируется как:

$$H(S) = (s_0 \cdot a^{n-1} + s_1 \cdot a^{n-2} + \dots + s_{n-1} \cdot a^0) \mod m$$

где:

- $s_i$  — код  $i$ -го символа строки (например, его числовое значение),
- $a$  — некоторое заранее выбранное основание (обычно небольшое простое число, например 33),
- $m$  — размер хеш-таблицы или общее количество бакетов.

В программе используется упрощенная формула полиномиального хеша:

$$H(S) = (H(S) \cdot a + s_i) \mod m$$

### 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]: джунгли
[1]:
[2]: вечным
[3]: тихим
[4]: светом
[5]:
[6]:
[7]:
[8]:
[9]: природа
[10]: спокойно
[11]:
[12]: качал->где
[13]: здесь
[14]: покрытом->гладь->ей
[15]: превращаются
[16]:
[17]:

```

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. Как правило, для реализации таких списков используется структура данных, называемая **односвязным списком**.

Односвязный список — это структура данных, состоящая из узлов, где каждый узел содержит элемент данных и ссылку на следующий узел. Первый элемент списка называют *головой* (head), а последний элемент — *хвостом*. (см. [рис.2](#))



Рис. 2. Структура односвязного списка

Основное преимущество односвязного списка заключается в том, что его структура позволяет динамически добавлять элементы, при этом элементы в памяти могут располагаться не последовательно, а в произвольном порядке. За счёт этого односвязный список

предоставляет гибкость в размере и легко масштабируется. Однако, в отличие от массивов, где доступ к элементам можно получить за постоянное время, в односвязном списке доступ к элементам осуществляется только последовательно.

Когда в методе цепочек при добавлении новой пары возникает коллизия, эта пара добавляется в конец списка, связанного с индексом, на который указала хеш-функция. Время добавления элемента в конец односвязного списка составляет  $O(1)$ , если есть ссылка на хвост. В противном случае, если список не имеет явного указания на хвост, может потребоваться полный проход по списку, что увеличивает время добавления до  $O(n)$ , где  $n$  — длина списка.

Операции удаления и поиска элемента в односвязном списке требуют последовательного прохода по списку. В худшем случае время поиска или удаления элемента будет  $O(n)$ , где  $n$  — количество элементов в списке. При удалении элемента важно перенаправить ссылки таким образом, чтобы исключить удалённый элемент из цепочки. Это делается путём изменения ссылки предыдущего элемента на следующий после удаляемого узел.

Среднее время выполнения операций в хеш-таблице с цепочками зависит от **коэффициента загрузки**  $\alpha$ , который равен отношению количества хранимых элементов к размеру массива. Если распределение хешей равномерное, то средняя длина списка 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 списков увеличивается, что замедляет операции поиска и удаления, делая их время выполнения ближе к  $O(n)$ .

#### Операции с односвязным списком:

- **Добавление элемента** (add): Элемент можно добавить в начало или конец списка. Сложность операции —  $O(1)$  при добавлении в начало,  $O(1)$  при наличии ссылки на хвост или  $O(n)$  без такой ссылки при добавлении в конец.
- **Удаление элемента** (remove): Требуется найти элемент, после чего перенаправить ссылки для исключения узла. Сложность операции —  $O(n)$ .
- **Поиск элемента** (find): Необходимо последовательно пройти по списку и найти элемент. Сложность операции —  $O(n)$ .
- **Подсчёт элементов по условию** (count): Проход по списку с проверкой каждого элемента на заданное условие. 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 В+-деревом

### 1.5.1 Поиск

Поиск элемента в В+-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 листовой 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 3](#) представлен пример структуры B+-дерева.

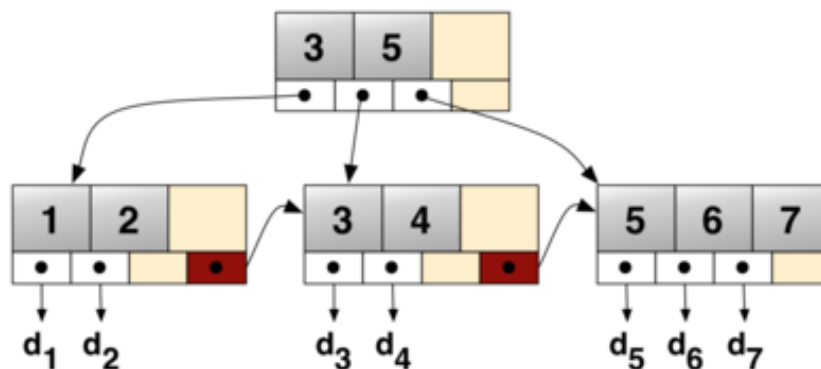


Рис. 3. Пример структуры Б+-дерева

На [рисунках 4-5](#) представлена реализованная заполненная структура Б+-дерева.

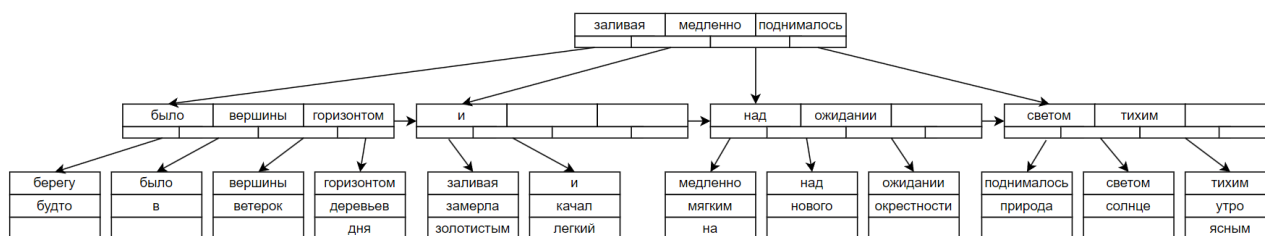


Рис. 4. Пример реализованной заполненной структуры Б+-дерева

```

Уровень: 0
заливая медленно поднималось |
Уровень: 1
было вершины горизонтом | и | над ожидании | светом тихим |
Уровень: 2
берегу будто | было в | вершины ветерок | горизонтом деревьев дня | заливая замерла золотистым | и качал легкий
медленно мягким на | над нового | ожидании окрестности | поднималось природа | светом солнце | тихим утро ясным

Б+- дерево дополнено словами из файла

```

Рис. 5. Пример реализованной заполненной структуры Б+-дерева

## 2 Особенности реализации

### 2.1 Структура Node

Структура Node представляет из себя реализацию одного узла односвязного списка LinkedList. Структура Node Обладает следующими полями:

1. Node\* next - указатель на следующий элемент списка.
2. string data - ключ, слово словаря.

### 2.2 Класс LinkedList

Класс LinkedList представляет из себя реализацию односвязного списка. Класс LinkedList обладает одним полем — head типа Node\*. Это поле является указателем на голову односвязного списка.

#### 2.2.1 Метод insert

Вход: объект типа LinkedList, string value - строковое значение.

Выход: вставленный новый Node в LinkedList.

Метод **insert** добавляет новый элемент в конец связного списка. В качестве параметра он принимает строковое значение **value**, которое должно быть вставлено в список.

Этот метод работает за время  $O(n)$ , где  $n$  — количество элементов в списке, так как в худшем случае требуется пройти по всему списку до конца, чтобы вставить новый элемент.

Код метода представлен в листинге 1.

Листинг 1. Метод insert

```
1 void LinkedList::insert(const string& value) {
2     ListNode* newNode = new ListNode(value);
3     if (head == nullptr) {
4         head = newNode;
5     }
6     else {
7         ListNode* temp = head;
8         while (temp->next != nullptr) {
9             temp = temp->next;
10        }
11        temp->next = newNode;
12    }
13 }
```

#### 2.2.2 Метод remove

Вход: объект типа LinkedList, string key - строковое значение.

Выход: bool - true, если элемент был удалён, иначе false.

Метод **remove** удаляет элемент из связанного списка по его значению **value**. В случае успешного удаления возвращается **true**, если элемент не найден — **false**.

Время работы метода в худшем случае —  $O(n)$ , где  $n$  — количество элементов в списке, так как в худшем случае может потребоваться пройти весь список.

Код метода представлен в листинге 2.

Листинг 2. Метод remove

```
1 bool LinkedList::remove(const string& value) {
2     if (head == nullptr) {
3         // cout << "Элемент не найден!" << endl;
4         return false;
5     }
6
7     if (head->data == value) {
8         ListNode* temp = head;
9         head = head->next;
10        delete temp;
11        cout << "Слово успешно удалено! " << endl;
12        return true;
13    }
14
15
16    ListNode* current = head;
17    ListNode* previous = nullptr;
18
19    while (current != nullptr && current->data != value) {
20        previous = current;
21        current = current->next;
22    }
23
24    if (current == nullptr) {
25        // cout << "Элемент " << value << "не найден!!!" << endl;
26        return false;
27    }
28
29    previous->next = current->next;
30    delete current;
31    cout << "Слово успешно удалено! " << endl;
32    return true;
33
34 }
```

### 2.2.3 Метод clear

Вход: Объект типа LinkedList.

Выход: очищенный объект типа `LinkedList`.

Метод производит проход от головы списка до его хвоста, постепенно удаляя на каждом шаге очередной узел.

Код метода представлен в листинге 3.

Листинг 3. Метод `clear`

```
1 void LinkedList::clear() {
2
3     ListNode* current = head;
4     while (current != nullptr) {
5         ListNode* toDelete = current;
6         current = current->next;
7         delete toDelete;
8     }
9     head = nullptr;
10
11 }
```

#### 2.2.4 Метод `search`

Вход: объект типа `LinkedList`, `string key` - строковое значение.

Выход: `true`, если объект найден, `false` - если нет.

Метод **`search`** осуществляет поиск элемента по его значению **`value`** в связанном списке.

Если элемент найден, метод возвращает **`true`**, в противном случае — **`false`**.

Время работы метода в худшем случае —  $O(n)$ , где  $n$  — количество элементов в списке, так как метод может потребовать пройти весь список.

Код метода представлен в листинге 4.

Листинг 4. Метод `find`

```
1 bool LinkedList::search(const string& value) {
2     ListNode* temp = head;
3
4     while (temp != nullptr) {
5         if (temp->data == value) {
6             return true;
7         }
8         temp = temp->next;
9     }
10    return false;
11
12 }
```

## 2.3 Класс HashTable

Класс HashTable представляет из себя реализацию хеш-таблицы. Структура Node Обладает следующими полями:

1. int total\_buckets - размер массива хеш-таблицы.
2. int total\_elements - количество элементов в хеш-таблице.
3. double fillability - коэффициент заполняемости.
4. LinkedList\* table - динамический массив хеш-таблицы.

### 2.3.1 Метод hashFunction

Вход: string key - строковое значение.

Выход: unsigned int - хеш слова.

Метод `hashFunction` вычисляет хеш-значение для строки `key` с помощью полиномиальной хеш-функции. Эта функция служит для преобразования ключа в числовое значение, которое затем используется для определения индекса в хеш-таблице.

Алгоритм работы следующий: метод инициализирует переменную `hash_value` равной 0 и использует основание полинома `a`, равное 33. Для каждого символа строки `key` хеш-значение обновляется по формуле: `hash_value = (hash_value * a + c) % total_buckets`, где `c` — это числовое значение символа. Операция `% total_buckets` гарантирует, что итоговое значение находится в диапазоне от 0 до `total_buckets - 1`, что предотвращает выход за пределы массива.

Код метода представлен в листинге 5.

Листинг 5. Метод hashFunction

```
1 unsigned int HashTable::hashFunction(const string& key) {  
2  
3     unsigned int hash_value = 0;  
4     int a = 33; // основание полинома  
5  
6     for (char c : key) {  
7         hash_value = (hash_value * a + static_cast<unsigned int>(c)) %  
total_buckets;  
8     }  
9  
10    return hash_value;  
11 }
```

### 2.3.2 Метод insert

Вход: объект типа HashTable, string key - строковое значение.

Выход: вставленный объект в хеш-таблицу.

Метод **insert** добавляет новый элемент с ключом **key** в хеш-таблицу. Процесс начинается с вычисления индекса для ключа с помощью хеш-функции **hashFunction(key)**. Если ключ уже существует в таблице (проверка выполняется с помощью метода **search**), добавление не выполняется.

Если элемент отсутствует, вызывается метод вставки в связанный список, хранящийся в соответствующем индексе хеш-таблицы. После этого обновляется количество элементов и вычисляется коэффициент заполненности таблицы (**fillability**), как отношение общего числа элементов к числу бакетов.

Если коэффициент заполненности превышает 0.9, выполняется процедура **rehash**, которая пересчитывает хеш-таблицу с большим количеством бакетов для предотвращения переполнения и повышения эффективности.

Код метода представлен в листинге 6.

Листинг 6. Метод insert

```
1 void HashTable::insert(const string& key) {
2     int index = hashFunction(key);
3     if (search(key) == -1) {
4         table[index].insert(key);
5         total_elements++;
6         fillability = static_cast<double>(total_elements) / static_cast<double>(
total_buckets);
7     }
8     if (fillability >= 0.9) {
9         rehash();
10    }
11 }
```

### 2.3.3 Метод remove

Вход: объект типа HashTable, string key\_r - строковое значение.

Выход: удаленный объект.

Метод **remove** удаляет элемент с указанным ключом **key** из хеш-таблицы. Сначала вычисляется индекс с помощью хеш-функции **hashFunction(key)**. Затем вызывается метод удаления **remove** для связанного списка, который хранится в этом индексе.

Если удаление прошло успешно, количество элементов уменьшается, и коэффициент заполненности таблицы **fillability** пересчитывается. Если элемент с таким ключом не был найден, выводится сообщение об ошибке.

Время работы метода удаления в среднем —  $O(1)$ , если коллизий мало. В худшем случае, если в таблице много коллизий, время работы увеличивается до  $O(n)$ , где  $n$  — длина связанного списка.

Код метода представлен в листинге 7.



#### Листинг 7. Метод remove

```
1 void HashTable::remove(const string& key) {
2
3     int index = hashFunction(key);
4     if (table[index].remove(key)) {
5         total_elements--;
6         fillability = static_cast<double>(total_elements) / static_cast<double>(
total_buckets);
7     }
8
9     else cout << "\n Элемент не найден! " << endl;
10 }
```

### 2.3.4 Метод clear

Вход: объект типа HashTable.

Выход: очищенный массив.

Метод **clear** очищает все элементы хеш-таблицы. Он проходит по всем корзинам **total\_buckets** и вызывает метод **clear** для каждого связанного списка, тем самым удаляя все элементы, хранящиеся в этих списках.

После этого количество элементов в таблице **total\_elements** устанавливается в 0, а коэффициент заполненности **fillability** обнуляется.

Код метода представлен в листинге 8.

#### Листинг 8. Метод clear

```
1 void HashTable::clear() {
2     for (int i = 0; i < total_buckets; i++) {
3         table[i].clear();
4     }
5     total_elements = 0;
6     fillability = 0.0;
7     cout << "Хэш-таблица полностью очищена!" << endl;
8
9 }
```

### 2.3.5 Метод search

Вход: Объект типа HashTable, string key - строковое значение.

Выход: индекс бакета в случае, если элемент найден или -1, если не найден.

Метод **search** выполняет поиск ключа в хеш-таблице и возвращает индекс корзины, в которой был найден ключ. Если ключ найден, метод возвращает индекс корзины, соответствующий этому ключу. Если ключ отсутствует в таблице, метод возвращает -1.

Для поиска ключа сначала вычисляется индекс корзины с помощью хеш-функции `hashFunction`. Затем метод проверяет наличие ключа в корзине по этому индексу с помощью метода `search` класса `LinkedList`. Если ключ найден, метод возвращает индекс корзины, в противном случае возвращается `-1`.

Время выполнения метода зависит от количества элементов в корзине и составляет  $O(1)$  в среднем случае при равномерном распределении ключей.

Код метода представлен в листинге 9.

Листинг 9. Метод `find`

```
1 int HashTable::search(const string& key) {
2
3     int index = hashFunction(key);
4     if (table[index].search(key)) return index;
5     else return -1;
6
7 }
```

### 2.3.6 Метод `insertFromFile`

Вход: объект типа `HashTable`, `string filename` - название файла в строковом формате.

Выход: дополненная хеш-таблица.

Метод `insertFromFile` отвечает за добавление элементов в хеш-таблицу из файла. Вначале открывается файл для чтения. Если файл не удалось открыть, выводится сообщение об ошибке.

Далее, метод построчно считывает файл, разбивая каждую строку на слова. Каждое слово преобразуется в нижний регистр и очищается от неалфавитных и нечисловых символов. После этого, если слово не пустое, оно добавляется в хеш-таблицу с помощью метода `insert`.

Код метода представлен в листинге 10.

Листинг 10. Метод `insertFromFile`

```
1 void HashTable::insertFromFile(const string& filename) {
2     ifstream file(filename);
3     if (!file.is_open()) {
4         cout << "Ошибка! Не удалось открыть файл!" << filename << endl;
5         return;
6     }
7     string line, word;
8     while (getline(file, line)) {
9         stringstream ss(line);
10        while (ss >> word) {
11            transform(word.begin(), word.end(), word.begin(), ::tolower);
12            word.erase(remove_if(word.begin(), word.end(), [](char c) {
```

```

13         return !isAlphaNum(c);
14     } , word.end());
15
16     if (!word.empty()) {
17         insert(word);
18         // cout << "Добавили " << word << endl;
19     }
20 }
21 }
22 // cout << "Количество элементов: " << total_elements << endl;
23 // cout << "Коэф. заполняемости хэш-таблицы: " << fillability << endl;
24 }

```

### 2.3.7 Метод rehash

Вход: объект типа `HashTable`, `total_buckets` - количество бакетов в хеш-таблице

Выход: дополненная хеш-таблица.

Метод **rehash** выполняет процесс пересоздания хеш-таблицы с увеличением числа корзин. Сначала старое количество корзин сохраняется в переменной `old_buckets`, и количество корзин удваивается. Затем создается новый массив корзин с обновленным размером.

Все элементы из старой таблицы переносятся в новый массив. Для этого метод перебирает все корзины старой таблицы и вставляет все содержащиеся в них элементы в новую таблицу, используя метод **insert**.

После переноса всех элементов старая таблица удаляется, освобождая память.

Время выполнения метода зависит от количества элементов в таблице и составляет  $O(n)$ , где  $n$  — количество элементов в таблице до выполнения операции **rehash**.

Код метода представлен в листинге 11.

Листинг 11. Метод rehash

```

1 void HashTable::rehash() {
2
3     // cout << "Пересоздаем таблицу..." << endl;
4
5     int old_buckets = total_buckets;
6
7     total_buckets *= 2;
8
9     LinkedList* old_table = table;
10    table = new LinkedList[total_buckets];
11    total_elements = 0;
12
13    for (int i = 0; i < old_buckets; i++) {
14
15        ListNode* temp = old_table[i].getHead();

```

```

16
17         while (temp != nullptr) {
18             insert(temp->data);
19             temp = temp->next;
20         }
21
22     }
23
24     delete[] old_table;
25
26 }

```

## 2.4 Класс Node

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

1. `InterNode* Parent` — указатель на родителя.
2. `vector<string> key` — массив ключей.
3. `int count` — количество ключей в узле.
4. `bool isLeaf` — `true`, если является листом, иначе `false`.

### 2.4.1 Метод GetBrother

Вход: объект типа Node, `int& flag` - ссылка на целочисленное значение.

Выход: указатель на объект типа Node.

Первым делом метод проверяет, является ли текущий узел корнем. Если им является, то возвращается нулевой указатель. Иначе производится проход по всем детям родительского узла, пока не будет найден текущий узел. Далее проводится проверка: если текущий узел является самым правым дочерним узлом, то из метода возвращается указатель на левый узел и в переменную `flag` устанавливается значение 1, иначе возвращается указатель на правый узел и в переменную `flag` устанавливается значение 2.

Код метода представлен в листинге 12.

Листинг 12. Метод GetBrother

```

1 Node* Node::GetBrother(int& flag) {
2     if (Parent == NULL)
3         return NULL;
4
5     Node* p = NULL;
6     for (int i = 0; i <= Parent->count; i++) {
7         if (Parent->Child[i] == this) {

```

```

8         if (i == Parent->count) {
9             p = Parent->Child[i - 1];
10            flag = 1;
11        }
12        else {
13            p = Parent->Child[i + 1];
14            flag = 2;
15        }
16    }
17 }
18 return p;
19 }

```

## 2.5 Класс LeafNode

Класс InterNode является реализацией листового узла B+-дерева. Он наследует класс Node. Этот класс обладает дополнительными полями LeafNode\* Pre\_Node - указатель на лист слева, Next\_Node - указатель на лист справа.

### 2.5.1 Метод Split

Вход: Объект типа LeafNode, p - указатель на узел типа LeafNode.

Выход: значение наименьшего ключа узла p

Первым делом метод производит перенос M последних ключей в лист p, после чего обновляет счётчик количества элементов в текущем узле и в правом брате. Далее возвращает значение наименьшего ключа узла p.

Код метода представлен в листинге 13.

Листинг 13. Метод Split

```

1 string LeafNode::Split(LeafNode* p) {
2     int j = 0;
3     for (int i = M - 1; i < M * 2 - 1; i++, j++)
4         p->key[j] = this->key[i];
5
6     this->count = this->count - j;
7     p->count = j;
8     return p->key[0];
9 }

```

### 2.5.2 Метод Insert

Вход: объект типа LeafNode, string value - строковое значение.

Выход: true, если ключ записан, false, если нет.

Первым делом метод производит поиск позиции, куда будет записан ключ. После чего перемещает ключи справа от этой позиции вправо на 1 ячейку. Далее записывает на позицию ключ value. В конце возвращает true.

Код метода представлен в листинге 14.

Листинг 14. Метод Insert

```
1  bool LeafNode::Insert(string value) {
2      int i = 0;
3      for (; (value > key[i]) && (i < count); i++);
4
5      for (int j = count; j > i; j--)
6          key[j] = key[j - 1];
7
8      key[i] = value;
9      count++;
10     return true;
11 }
```

### 2.5.3 Метод Delete

Вход: объект типа LeafNode, string value - строковое значение.

Выход: true, если элементы сместились, false, если ключ не найден.

Первым делом метод производит поиск ключа в листе, если ключа не было найдено, то возвращает false. Иначе все элементы справа от позиции найденного ключа сдвигаются влево на 1. После чего метод возвращает true.

Код метода представлен в листинге 15.

Листинг 15. Метод Delete

```
1  bool LeafNode::Delete(string value)
2  {
3      bool found = false;
4      int i = 0;
5      for (; i < count; i++){
6          if (value == key[i]) {
7              found = true;
8              break;
9          }
10     }
11     if (found == false)
12         return false;
13     int j = i;
14     for (; j < count - 1; j++)
15         key[j] = key[j + 1];
16     key[j] = "";
17     count--;
```

```

18     return true;
19 }

```

## 2.5.4 Метод Merge

Вход: объект типа LeafNode, указатель на узел типа LeafNode.

Выход: true, если вставка ключей произошла успешно, иначе false.

Изначально метод проверяет, чтобы после слияния количество элементов было больше  $M*2$ , если это так, то производится вставка всех ключей брата в текущий узел методом Insert и возвращается true, иначе возвращается false.

Код метода представлен в листинге 16.

Листинг 16. Метод Merge

```

1  bool LeafNode::Merge(LeafNode* p) {
2      if (this->count + p->count > M * 2 - 1)
3          return false;
4      for (int i = 0; i < p->count; i++)
5          this->Insert(p->key[i]);
6      return true;
7  }

```

## 2.6 Класс InterNode

Класс InterNode является реализацией внутреннего узла B+-дерева. Он наследует класс Node. Этот класс обладает дополнительным полем `vector<Node*> Child` - это массив указателей на детей этого узла.

### 2.6.1 Метод Merge

Вход: объект типа InterNode, указатель на узел типа InterNode.

Выход: bool значение - результат выполнения операции слияния.

Первым делом метод размещает наибольший ключ крайнего левого ребёнка правого брата на последнюю позицию в списке ключей текущего узла. После этого он метод перемещает указатель на крайнего левого ребёнка правого брата в конец списка детей текущего узла. После этого в цикле перемещает оставшиеся ключи и оставшихся детей правого брата в правую часть списка ключей и списка детей текущего узла. После этого возвращается true.

Код метода представлен в листинге 17.

Листинг 17. Метод Merge

```

1  bool InterNode::Merge(InterNode* p) {
2      key[count] = p->Child[0]->key[0];

```

```

3      count++;
4      Child[count] = p->Child[0];
5      Child[count]->Parent = this;
6      for (int i = 0; i < p->count; i++) {
7          key[count] = p->key[i];
8          count++;
9          Child[count] = p->Child[i + 1];
10         Child[count]->Parent = this;
11     }
12     return true;
13 }

```

### 2.6.2 Метод Insert

Вход: объект типа InterNode, string value - строковое значение; указатель на узел типа Node.

Выход: bool значение - результат выполнения операции вставки.

Первым циклом производится поиск позиции, куда будет добавлен ключ и ребёнок. Следующим циклом производится перемещение ключей на единицу вправо от найденной позиции. Аналогично делается и для детей. После этого в найденную позицию записывается ключ и правым ребёнком этого ключа записывается узел New.

Код метода представлен в листинге 18.

Листинг 18. Метод Insert

```

1      bool InterNode::Insert(string value, Node* New) {
2          int i = 0;
3          for (; (i < count) && (value > key[i]); i++);
4
5          for (int j = count; j > i; j--)
6              key[j] = key[j - 1];
7
8          for (int j = count + 1; j > i + 1; j--)
9              Child[j] = Child[j - 1];
10
11         key[i] = value;
12         Child[i + 1] = New;
13         New->Parent = this;
14         count++;
15         return true;
16     }

```

### 2.6.3 Метод Split

Вход: объект типа InterNode, указатель на узел типа InterNode, string k - строковое значение.



Выход: строковое значение.

Изначально производится вычисление индекса, по которому будет производиться деление узла на две части. Этот индекс равен  $M-1$ , то есть  $M-1$  элементов останутся в текущем узле, а последние  $M - 1$  ключей переместятся в новый узел. Следует заметить, что оставшийся ключ записывается в переменную  $k$ . Также происходит и с детьми текущего узла. Далее возвращаем ключ  $k$ .

Код метода представлен в листинге 19.

Листинг 19. Метод Split

```
1  string InterNode::Split(InterNode* p, string k) {
2      int i = 0, j = 0;
3
4      int pos = M-1;
5      k = this->key[pos];
6      j = 0;
7      for (i = pos + 1; i < M * 2 - 1; i++, j++)
8          p->key[j] = this->key[i];
9      j = 0;
10     for (i = pos + 1; i <= M * 2 - 1; i++, j++) {
11         this->Child[i]->Parent = p;
12         p->Child[j] = this->Child[i];
13     }
14     this->count = pos;
15     p->count = M * 2 - pos - 2;
16     return k;
17 }
```

#### 2.6.4 Метод Delete

Вход: объект типа InterNode, string  $k$  - строковое значение.

Выход: bool значение - результат выполнения операции удаления.

Производим поиск позиции, где расположен ключ, далее удаляем ключ. после чего удаляем ребёнка справа от этого ключа.

Код метода представлен в листинге 20.

Листинг 20. Метод Delete

```
1  bool InterNode::Delete(string k) {
2      int i = 0;
3      for (; (k >= key[i]) && (i < count); i++);
4
5      for (int j = i - 1; j < count - 1; j++)
6          key[j] = key[j + 1];
7
8      int d = i;
9      for (; d < count; d++) {
```

```

10         Child[d] = Child[d + 1];
11     }
12     Child[d] = NULL;
13     count--;
14     return true;
15 }

```

### 2.6.5 Метод Slib

Вход: объект типа InterNode, указатель на узел типа InterNode.

Выход: bool значение - результат выполнения операции переноса.

Первым делом проверяем на какого брата указывает p, если левый, то переносим из левого брата наибольшего ребёнка в текущий узел на позицию наименьшего ребёнка. Иначе из правого брата переносим наименьшего ребёнка на позицию старшего ребёнка текущего узла, после чего у старшего брата сдвигаем ключи и детей на один влево.

Код метода представлен в листинге 21.

Листинг 21. Метод Slib

```

1  bool InterNode::Slib(InterNode* p) {
2      int i, j;
3      if (p->key[0] < this->key[0]) {
4          for (i = count; i > 0; i--)
5              key[i] = key[i - 1];
6          for (j = count + 1; j > 0; j--)
7              Child[j] = Child[j - 1];
8          key[0] = Child[0]->key[0];
9          Child[0] = p->Child[p->count];
10         Child[0]->Parent = this;
11     }
12     else {
13         key[count] = p->Child[0]->key[0];
14         Child[count + 1] = p->Child[0];
15         Child[count + 1]->Parent = this;
16         for (i = 1; i < p->count; i++)
17             p->key[i - 1] = p->key[i];
18         for (j = 0; j < p->count; j++)
19             p->Child[j] = p->Child[j + 1];
20     }
21     this->count++;
22     p->count--;
23     return true;
24 }

```

## 2.7 Класс Bplus

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

1. int M - ранг дерева.
2. Node\* Root - указатель на корень дерева.

### 2.7.1 Метод Find

Вход: объект типа BPlus, string data - строковое значение.

Выход: указатель на узел типа LeafNode.

Метод начинает свою работу с корня дерева p. Пробегается по всем ключам корня, пока не находит ключ в узле, больший ключа data. Как только такой ключ будет найден, указатель на ребёнка справа от этого узла будет записан в переменную p. После чего всё повторяется. Так будет продолжаться, пока не будет достигнуть лист.

Код метода представлен в листинге 22.

Листинг 22. Метод Find

```
1 LeafNode* Bplus::Find(string data) {
2     int i = 0;
3     Node* p = Root;
4     InterNode* q;
5
6     while (NULL != p) {
7         if (p->isLeaf)
8             break;
9         for (i = 0; i < p->count; i++) {
10             if (data < p->key[i])
11                 break;
12         }
13         q = (InterNode*)p;
14         p = q->Child[i];
15     }
16     return (LeafNode*)p;
17 }
```

### 2.7.2 Метод Add\_Node

Вход: объект типа BPlus, p - указатель на узел типа InterNode, k - строковое значение, New\_Node - указатель на узел типа Node.

Выход: bool значение - результат выполнения операции добавления узла

Первоначально производится проверка на существование узла p, а также на то, чтобы он не был листовым узлом. Далее проверяем, если количество копий ключей в узле меньше  $M * 2 - 1$ , то можем добавить в этот узел ключ k. Иначе производим расщепление текущего

узла на два. Далее определяем, в какой узел будет помещён ключ с ребёнком, помещаем в выбранный узел. Далее, если текущий узел уже является корневым, то создаём расширяем дерево вверх, иначе снова вызываем метод `Add_Node` для родителя.

Код метода представлен в листинге 23.

Листинг 23. Метод `Add_Node`

```

1  bool Bplus::Add_Node(InterNode* p, string k, Node* New_Node) {
2      if (p == NULL || p->isLeaf)
3          return false;
4      if (p->count < M * 2 - 1)
5          return p->Insert(k, New_Node);
6
7      InterNode* Brother = new InterNode(M);
8      string NewKey = p->Split(Brother, k);
9
10     if (Brother->Child[0]->key[0] < k) {
11         Brother->Insert(k, New_Node);
12     }
13     else {
14         p->Insert(k, New_Node);
15     }
16
17     InterNode* parent = (InterNode*)(p->Parent);
18     if (parent == NULL) {
19         parent = new InterNode(M);
20         parent->Child[0] = p;
21         parent->key[0] = NewKey;
22         parent->Child[1] = Brother;
23         p->Parent = parent;
24         Brother->Parent = parent;
25         parent->count = 1;
26         Root = parent;
27         return true;
28     }
29     return Add_Node(parent, NewKey, Brother);
30 }

```

### 2.7.3 Метод `Search`

Вход: объект типа `Bplus`, `string data` - строковое значение.

Выход: `bool` значение; модификация строки `sPath`.

Метод производит действия, аналогичные методу `Find`, однако не останавливается на нахождении листового узла. Он проходится по листовому узлу в поисках ключа, если находит, то строка из метода будет возвращено значение `true`, иначе `false`. Также по мере спуска по дереву, метод записывает переход с одного яруса дерева на другой в переменную по

ссылке sPath.

Код метода представлен в листинге 24.

Листинг 24. Метод Search

```
1  bool Bplus::Search(string data) {
2      int i = 0;
3      Node* p = Root;
4      if (p == NULL)
5          return false;
6      InterNode* q;
7      while (p != NULL) {
8          if (p->isLeaf)
9              break;
10         for (i = 0; (i < p->count) && (data >= p->key[i]); i++);
11         int k = i > 0 ? i - 1 : i;
12         q = (InterNode*)p;
13         p = q->Child[i];
14     }
15     if (p == NULL)
16         return false;
17     bool found = false;
18     for (i = 0; i < p->count; i++) {
19         if (data == p->key[i])
20             found = true;
21     }
22     return found;
23 }
```

#### 2.7.4 Метод Insert

Вход: объект типа Bplus, string data - строковое значение.

Выход: bool значение - результат выполнения операции вставки.

Метод изначально производит проверку на наличие ключа в дереве, если он есть, то метод возвращает false. Далее производится поиск листового узла, где ключ можно было бы разместить. Если такой не был найден, это означает, что дерево ещё пустое, поэтому создаётся корневой узел дерева. Далее, если размер текущего узла меньше  $M*2 - 1$ , то вставляет данные в этот узел. Иначе требуется расщепить его, используя метод Split класса LeafNode, который возвращает ключ, который в дальнейшем потребуется для изменения данных в родителе. После расщепления происходит переприсваивание ссылок на левых/правых братьев. Далее выбирается, в какой листовой узел будет размещён конкретный ключ. Далее требуется проверить, что если произошло расщепление корневого узла, то требуется создать новый узел. Иначе производится обновление ключа в узлах-родителях.

Код метода представлен в листинге 25.

```

1  bool Bplus::Insert(string data) {
2      string a;
3      if (Search(data, a) == true)
4          return false;
5
6      LeafNode* Old_Node = Find(data);
7
8      if (NULL == Old_Node) {
9          Old_Node = new LeafNode(M);
10         Root = Old_Node;
11     }
12
13     if (Old_Node->count < M * 2 - 1)
14         return Old_Node->Insert(data);
15
16     LeafNode* New_Node = new LeafNode(M);
17
18     string k = Old_Node->Split(New_Node);
19
20     LeafNode* OldNext = Old_Node->Next_Node;
21     Old_Node->Next_Node = New_Node;
22     New_Node->Next_Node = OldNext;
23     New_Node->Pre_Node = Old_Node;
24
25     if (OldNext != NULL)
26         OldNext->Pre_Node = New_Node;
27
28     if (data < k) {
29         Old_Node->Insert(data);
30     }
31     else {
32         New_Node->Insert(data);
33     }
34     InterNode* parent = (InterNode*)(Old_Node->Parent);
35
36     if (parent == NULL) {
37         InterNode* New_Root = new InterNode(M);
38         New_Root->Child[0] = Old_Node;
39         New_Root->key[0] = k;
40         New_Root->Child[1] = New_Node;
41         Old_Node->Parent = New_Root;
42         New_Node->Parent = New_Root;
43         New_Root->count = 1;
44         Root = New_Root;
45         return true;
46     }

```

```

47
48     return Add_Node(parent , k , New_Node);
49 }

```

### 2.7.5 Метод Delete

Вход: объект типа Bplus, string data - строковое значение.

Выход: bool значение - результат выполнения операции удаления.

Первым делом метод производит поиск листового узла, в котором может быть расположен ключ. Далее производится удаление ключа из узла. После этого метод получает ссылку на родителя этого листового узла. Далее, проверяем ситуацию, если родитель у текущего узла нет, то требуется проверить, не был ли элемент последним в листовом узле, если был, то удаляем этот листовой узел. Далее, если количество ключей до сих пор больше  $M-1$ , то пробуем обновить ключи в родителе и возвращаем true. Если же узел меньше  $M-1$ , то пробуем переместить ключ от левого брата или ключ от правого брата. Если же и это не получилось сделать, то это значит, что можно произвести слияние текущего узла с одним из братьев. Далее алгоритм производит слияние узлов и вызывает метод Remove\_Node, который будет удалять ключ data по направлению к корню дерева.

Код метода представлен в листинге 26.

Листинг 26. Метод Delete

```

1  bool Bplus::Delete(string data) {
2      LeafNode* Old_Node = Find(data);
3      string NewKey = Old_Node->key[0];
4      if (Old_Node == NULL)
5          return false;
6      if (false == Old_Node->Delete(data))
7          return false;
8
9      InterNode* parent = (InterNode*)(Old_Node->Parent);
10     if (NULL == parent || parent->Child.size() == 0) {
11         if (0 == Old_Node->count) {
12             delete Old_Node;
13             Root = NULL;
14         }
15         return true;
16     }
17     if (Old_Node->count >= M - 1) {
18         for (int i = 0; (i < parent->count) && (data >= parent->key[i]); i++)
19         {
20             if (parent->Child[i + 1]->isLeaf) {
21                 parent->key[i] = parent->Child[i + 1]->key[0];
22             }
23             else {

```

```

23         parent->key[i] = static_cast<InterNode*>(parent->Child[i + 1])
->getMin();
24     }
25 }
26     return true;
27 }
28
29     int flag = 1;
30     LeafNode* Brother = (LeafNode*)(Old_Node->GetBrother(flag));
31     string NewData = "";
32     if (Brother->count > M - 1) {
33         if (1 == flag) {
34             NewData = Brother->key[Brother->count - 1];
35         }
36         else {
37             NewData = Brother->key[0];
38         }
39         Old_Node->Insert(NewData);
40         Brother->Delete(NewData);
41
42         if (1 == flag) {
43             for (int i = 0; i <= parent->count; i++) {
44                 if (parent->Child[i] == Old_Node && i > 0)
45                     parent->key[i - 1] = Old_Node->key[0];
46             }
47         }
48         else {
49             for (int i = 0; i <= parent->count; i++) {
50                 if (parent->Child[i] == Old_Node && i > 0)
51                     parent->key[i - 1] = Old_Node->key[0];
52                 if (parent->Child[i] == Brother && i > 0)
53                     parent->key[i - 1] = Brother->key[0];
54             }
55         }
56         return true;
57     }
58
59     if (1 == flag) {
60         Brother->Merge(Old_Node);
61         NewKey = Old_Node->key[0];
62         LeafNode* OldNext = Old_Node->Next_Node;
63         Brother->Next_Node = OldNext;
64         if (NULL != OldNext)
65             OldNext->Pre_Node = Brother;
66         delete Old_Node;
67     }
68     else {

```



```

69         Old_Node->Merge( Brother );
70         NewKey = Brother->key [ 0 ];
71         LeafNode* OldNext = Brother->Next_Node;
72         Old_Node->Next_Node = OldNext;
73         if (NULL != OldNext)
74             OldNext->Pre_Node = Old_Node;
75
76         delete Brother;
77     }
78     return Remove_Node( parent , NewKey );
79 }

```

### 2.7.6 Метод Remove\_Node

Вход: объект типа Bplus, указатель на узел типа InterNode\*, string k - строковое значение.

Выход: bool значение - результат выполнения операции удаления узла.

Изначально метод попытается удалить ключ из текущего узла, если не получилось, значит такого ключа больше нет по направлению к корню дерева и метод можно завершать со значением false. Далее производятся те же действия, что и в методе Delete, за исключением последней части, где при слиянии с братьями не производится установка ссылок друг на друга, т.к. во внутренних узлах их нет.

Код метода представлен в листинге 27.

Листинг 27. Метод Remove\_Node

```

1  bool Bplus::Remove_Node(InterNode* p, string k) {
2      if (k == "")
3          return false;
4
5      if (false == p->Delete(k))
6          return false;
7
8      InterNode* parent = (InterNode*)(p->Parent);
9      if (NULL == parent || parent->Child.size() == 0) {
10         if (0 == p->count) {
11             Root = p->Child[0];
12             delete p;
13         }
14         return true;
15     }
16
17     if (p->count >= M - 1) {
18         for (int i = 0; (i < parent->count) && (k >= parent->key[i]); i++) {
19             // обновляем ключи в родителе если надо
20             if (parent->key[i] == k)

```

```

20         parent->key[i] = p->getMin();
21     }
22     for (int i = 0; i < p->count; i++) { // обновляем ключи в узле относит
ельно правого ребенка
23         if (p->Child[i + 1]->isLeaf) {
24             p->key[i] = p->Child[i + 1]->key[0];
25         }
26         else {
27             p->key[i] = static_cast<InterNode*>(p->Child[i + 1])->getMin()
;
28         }
29     }
30
31     return true;
32 }
33 int flag = 1;
34 InterNode* Brother = (InterNode*)(p->GetBrother(flag));
35 if (Brother->count > M - 1) {
36     p->Slib(Brother);
37
38     for (int i = 0; i < Brother->count; i++) { // обновляем ключи у узла,
откуда перенесли
39         if (Brother->Child[i + 1]->isLeaf) {
40             Brother->key[i] = Brother->Child[i + 1]->key[0];
41         }
42         else {
43             Brother->key[i] = static_cast<InterNode*>(Brother->Child[i +
1])->getMin();
44         }
45     }
46     for (int i = 0; i < p->count; i++) { // обновляем ключи у узла, куда п
еренесли
47         if (p->Child[i + 1]->isLeaf) {
48             p->key[i] = p->Child[i + 1]->key[0];
49         }
50         else {
51             p->key[i] = static_cast<InterNode*>(p->Child[i + 1])->getMin()
;
52         }
53     }
54     for (int i = 0; i <= parent->count; i++) { // обновляем ключи у родит
еля
55         if (parent->Child[i] == p && i > 0)
56             parent->key[i - 1] = p->getMin();
57         if (parent->Child[i] == Brother && i > 0)
58             parent->key[i - 1] = Brother->getMin();
59     }

```

```

60         return true;
61     }
62     string NewKey = "";
63     if (1 == flag) {
64         Brother->Merge(p);
65         for (int i = 0; i < Brother->count; i++) { // обновляем ключи в узле
66             if (Brother->Child[i + 1]->isLeaf) {
67                 Brother->key[i] = Brother->Child[i + 1]->key[0];
68             }
69             else {
70                 Brother->key[i] = static_cast<InterNode*>(Brother->Child[i +
1])->getMin();
71             }
72         }
73         NewKey = p->key[0];
74         delete p;
75     }
76     else {
77         p->Merge(Brother);
78         for (int i = 0; i < p->count; i++) { // обновляем ключи в узле
79             if (p->Child[i + 1]->isLeaf) {
80                 p->key[i] = p->Child[i + 1]->key[0];
81             }
82             else {
83                 p->key[i] = static_cast<InterNode*>(p->Child[i + 1])->getMin()
;
84             }
85         }
86         NewKey = Brother->key[0];
87         delete Brother;
88     }
89     return Remove_Node(parent, NewKey);
90 }

```

### 2.7.7 Метод loadFromFile

Вход: объект типа Bplus, string path - путь к файлу.

Выход: дополненное B+-дерево.

Метод производит считывание всего содержимого файла в строку. После этого регулярным выражением из списка убираются все символы, не входящие в русский алфавит. Далее все символы приводятся к нижнему регистру. После чего в цикле while из строки поочерёдно отбираются 30 слов и записываются в B+-дерево.

Код метода представлен в листинге 28.

Листинг 28. Метод loadFromFile

```

1  void Bplus::loadFromFile(string path) {
2      ifstream file(path, ios::in);
3      stringstream ss;
4      ss << file.rdbuf();
5      string editedFileString;
6      editedFileString = regex_replace(ss.str(), regex("[а-яА-Я]"), " "); // уб
ираем небуквенные символы
7      istringstream input{ editedFileString };
8
9      int counter = 0;
10     while (!input.eof()) {
11         string substring;
12         input >> substring;
13         transform(substring.begin(), substring.end(), substring.begin(), ::
tolower); // приводим слово к нижнему регистру
14         Insert(substring);
15         counter++;
16         if (counter == 30) return;
17     }
18 }

```

### 2.7.8 Метод clear

Вход: объект типа Bplus.

Выход: очищенный объект типа Bplus.

Метод рекурсивно производит левый обход дерева. Как только ему встречается лист, он его удаляет. После того как все листья некоторого узла будут удалены, удаляется также и этот узел. Так продолжается, пока не будет очищено всё дерево. После выхода из рекурсии и удаления корня дерева, на его место устанавливается новый корень.

Код метода представлен в листинге 29.

Листинг 29. Метод clear

```

1  void Bplus::clear(Node* root) {
2      for (int i = 0; i < root->count + 1; i++) {
3          if (root->isLeaf) {
4              Root = NULL;
5              delete root;
6              root = NULL;
7              return;
8          }
9          else {
10             clear(((InterNode*)root)->Child[i]);
11         }
12     }
13     if (root == Root) {
14         Root = NULL;

```

```
15         delete root;
16         root = NULL;
17     }
18     else {
19         delete root;
20         root = NULL;
21     }
22 }
```

---

### 3 Результаты работы программы

При запуске программы в консоли выводится меню программы и пользователю предлагается выбрать действие (рис.6).

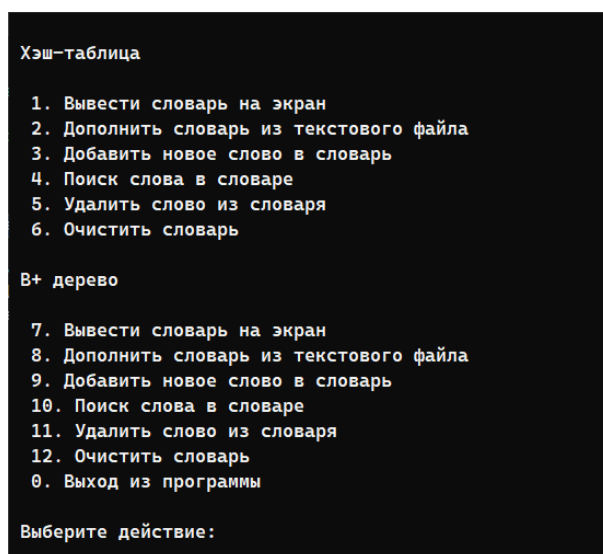


Рис. 6. Меню программы

При нажатии на клавишу один в самом начале программы, пользователю будет выведено сообщение о том, что хеш-таблица пуста. (рис.7).

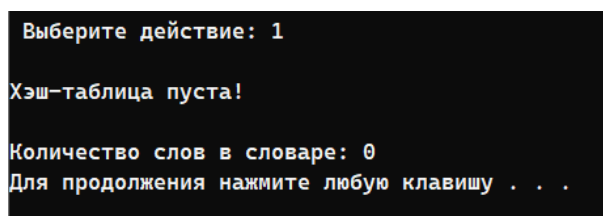


Рис. 7. Хеш-таблица

При нажатии на клавишу "2" в словарь будут добавлены слова из файла "story.txt" расположенного в корне проекта. После этого будет выведено информационное письмо пользователю о проведённой операции. (рис.8)

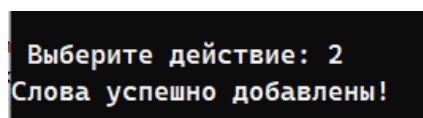


Рис. 8. Дополнение словаря из текстового файла

При нажатии на клавишу "3" пользователю будет предложено ввести слово, которое он хотел бы добавить в словарь. После ввода слова, выводится информация об успешности добавления слова в словарь. (рис.9)

```
Выберите действие: 3
Введите слово (без пробелов), которое хотите добавить:
привет
Слово успешно добавлено!
```

Рис. 9. Добавление нового слова в словарь

При нажатии на клавишу "4" пользователю будет предложено ввести слово, которое он хотел бы найти в словаре. После ввода слова, выводится информация об успешности поиска слова в словаре, а также в случае успеха будет выведен номер индекса массива хеш-таблицы, где расположено слово. (рис.10)

```
Выберите действие: 4
Введите слово, которое хотите найти:
ветерок
Слово ветерок присутствует в словаре!
Индекс бакета: 70
```

Рис. 10. Поиск слова в словаре

При нажатии на клавишу "5" пользователю будет предложено ввести слово, которое он хотел бы удалить из словаря. После ввода слова, выводится информация об успешности удаления слова из словаря. (рис.11)

```
Выберите действие: 5
Введите слово, которое хотите удалить:
ветерок
Слово успешно удалено!
```

Рис. 11. Удаление слова из словаря

При нажатии на клавишу "6" словарь будет очищен. Также будет выведено соответствующее сообщение. (рис.12)

```
Выберите действие: 6
Хэш-таблица полностью очищена!
```

Рис. 12. Очистка хеш-таблицы

Помимо хеш-таблицы, в программе также присутствует возможность работы с B+-деревом. При нажатии на клавишу "7" отвечающую за отображение B+-дерева, при отсутствии в нём каких-либо элементов, будет выведено сообщение о том, что оно пустое. (рис.13)

```
Выберите действие: 7
B+-дерево пустое
```

Рис. 13. Вывод пустого B+-дерева на экран

Работа клавиши "8" (дополнение словаря из текстового файла) аналогична работе клавиши "2" . (рис.14)

```
Уровень: 0
медленно |
Уровень: 1
горизонтом заливая и | над поднималось тихим |
Уровень: 2
было вершины ветерок | горизонтом деревьев | заливая золотистым | и качал легкий | медленно мягким | над окрестности
| поднималось светом солнце | тихим утро ясным |
В+- дерево дополнено словами из файла
```

Рис. 14. Дополнение словаря из текстового файла

Работа клавиши "9" (вставки) аналогична работе клавиши "3" . (рис.15)

```
Выберите действие: 9
Введите слово (без пробелов), которое хотите добавить:
привет
Вставка прошла успешно
```

Рис. 15. Вставка нового слова в В+-дерево

Работа клавиши "10" (поиска) аналогична работе клавиши "4" . (рис.16)

```
Выберите действие: 10
Введите слово, наличие которого хотите проверить: привет
Слово было найдено!
```

Рис. 16. Поиск слова в словаре

Работа клавиши "11" (удаления) аналогична работе клавиши "5" . (рис.17)

```
Выберите действие: 11
Введите слово, которое хотите удалить:
привет
Удаление слова прошло успешно
```

Рис. 17. Удаление слова из словаря

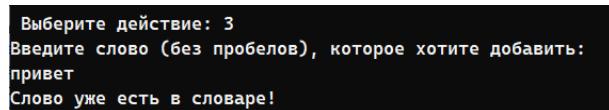
Работа клавиши "12" (очистка словаря) аналогична работе клавиши "6" . (рис.18)

```
Выберите действие: 12
В+-дерево очищено
```

Рис. 18. Очистка словаря

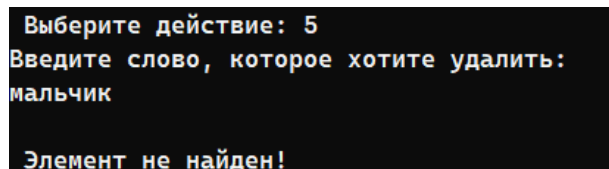


Также на рисунках 19-21 представлены некоторые исключительные ситуации, которые обрабатываются программой.



```
Выберите действие: 3
Введите слово (без пробелов), которое хотите добавить:
привет
Слово уже есть в словаре!
```

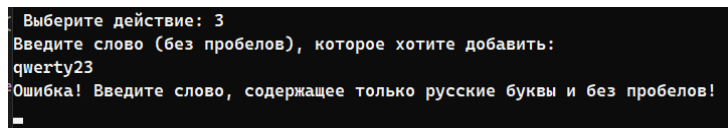
Рис. 19. Слово уже присутствует в словаре



```
Выберите действие: 5
Введите слово, которое хотите удалить:
мальчик

Элемент не найден!
```

Рис. 20. Удаление несуществующего слова



```
Выберите действие: 3
Введите слово (без пробелов), которое хотите добавить:
qwerty23
Ошибка! Введите слово, содержащее только русские буквы и без пробелов!
```

Рис. 21. Ввод слова, используя буквы не русского алфавита

## 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, а также функции очистки словаря и загрузки/дополнения словаря из текстового файла. Для разрешения коллизий в хеш-таблице был использован метод цепочек. В качестве хеш-функции использовался полиномиальный хеш.

Хеш-функция, используемая в хеш-таблице генерирует значения от 0 до 10, за счёт такого небольшого диапазона она имеет значительно меньшую устойчивость к коллизиям, по сравнению с реализациями, в которых ширина диапазона значений достигает  $2^{30}$  и более. Однако, 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, а также рассмотрение всех ситуаций при реализации операций добавления и удаления слова из словаря, основанного на В+-дереве. Также за счёт объектно-ориентированного подхода и разбиения кода В+-дерева на функции достигается высокий уровень его читаемости.

Недостатком реализованной программы является недоработка с подменой ссылок в корневом узле при удалении оттуда элемента.

В планы на масштабирование программы можно записать следующее:

1. Реализация поиска однокоренных слов в хеш-таблице.
2. Реализация бинарного поиска в узлах В+-дерева.

## 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.