

Оглавление

Введение	Error! Bookmark not defined.
Глава 1. Структуры данных для хранения словаря вида слово - частота появления в тексте.....	4
1.1. Хэш таблица.....	4
1.2. AVL-дерево.....	5
1.3. TST.....	5
1.4. Trie.....	6
1.5. Тестирование	6
1.6. Вывод	7
1.7. Дерево Бургкхардта-Келлера.....	7
Глава 2. Практическая реализация алгоритма.....	9
2.1. Класс Trie	9
2.2. Класс BKtree	11
2.3. Алгоритм работы программы (функция main)	14
2.4. Пример работы программы	15
Заключение	17
Список использованной литературы.....	18
Приложение	19

Введение

В современном мире пользовательские интерфейсы требуют удобства и скорости ввода данных. Одной из технологий, упрощающих ввод текста, является автокоррекция и автодополнение, где ключевую роль играют алгоритмы прогнозирования слова по его частичному вводу. Задача создания подобного помощника тесно связана с технологиями T9 (Text on 9 keys), который предлагает пользователю наиболее вероятные слова, основываясь на введенных символах и частоте их использования в языке. Эта технология находит применение в мобильных устройствах, мессенджерах, поисковых системах и других платформах, где важен быстрый и точный ввод текста.

Задача данной курсовой работы заключается в моделировании помощника T9, который сможет подсказывать пользователю слова на основе их частоты появления в тексте. Решение включает построение эффективной структуры для хранения слов и их частот, обработку новых слов и обновление словаря в режиме реального времени. В ходе работы реализован режим обучения, где словарь пополняется новыми словами, и рабочий режим, в котором подсказки автоматически обновляются по мере использования.

Глава 1. Структуры данных для хранения словаря вида слово - частота появления в тексте

Для решения задачи моделирования помощника Т9 ключевым аспектом является выбор структуры данных, которая позволит эффективно хранить и искать слова по их частям. Одним из основных требований является поддержка быстрого поиска по префиксам слов, а также возможность динамического обновления частот слов и добавления новых элементов в словарь. В данном разделе рассматриваются возможные алгоритмы и структуры данных, которые могут быть применены для решения данной задачи.

Для решения задачи поиска слова по префиксу наилучшим образом подходят следующие структуры данных: хэш-таблица, AVL-дерево, тернарное дерево поиска (Ternary Search Tree, TST), префиксное дерево (Trie). Trie кажется самой подходящей из них, но это следует проверить. Пропишем эти структуры данных и сравним их на тестах, чтобы вычислить наиболее эффективную.

1.1. Хэш таблица

- Принцип работы: хранение пар «ключ-значение» (слово-частота) в массиве с использованием хеш-функции и разрешения коллизий (цепочки или открытая адресация).
- Время вставки: в среднем $O(1)$. В худшем случае (при коллизиях) может достигать до $O(n)$, но при хорошем выборе хеша и достаточном размере таблицы средний случай доминирует.
- Время поиска: в среднем $O(1)$. Очень быстрая операция, так как поиск сводится к вычислению хеш-функции.
- Автодополнение: сложно и неэффективно. Нет естественной структуры для обхода по префиксам. Чтобы найти подходящие слова по префиксу, приходится итерировать по всей таблице или иметь дополнительные индексы.
- Память: хранит ключи полностью, а также структуры хеш-таблицы. Память, как правило, плотнее, чем у Trie, но может быть больше, чем у деревьев, поскольку дублируются строки и поддерживающие структуры.
- Дополнительные особенности: очень подходит, если нужен быстрый доступ к частоте слова по точному ключу.
- Код реализации хэш-таблицы представлен в Приложении 1.

1.2. AVL-дерево

- Принцип работы: двоичное дерево поиска, которое автоматически балансируется после каждой вставки или удаления, поддерживая разницу высот поддеревьев максимум 1.
- Время вставки: $O(\log n)$, где n – количество слов. Дерево поддерживает логарифмическую высоту, поэтому время вставки и поиска логарифмическое.
- Время поиска: $O(\log n)$. Поиск упорядочен по лексикографическому порядку слов.
- Автодополнение: не так эффективно, как в Trie или TST. Для нахождения всех слов с заданным префиксом нужно сначала найти диапазон, затем выполнить обход участка дерева. Это операции, связанные с бинарным поиском и обходом, что сложнее и дольше, чем в Trie.
- Память: более эффективна, чем у Trie, так как хранится только один узел на слово, плюс ссылки на детей и балансирующий фактор. Нет больших массивов указателей. Однако ключи (слова) хранятся полностью в каждом узле.
- Дополнительные особенности: хороший универсальный вариант, если важен средний случай для поиска, и нужны операции на множестве слов (поиск, вставка, удаление) с гарантированной $O(\log n)$ сложностью. □ Код реализации AVL-дерева представлен в приложении 2.

1.3. TST

- Принцип работы: является промежуточным решением между Trie и двоичным деревом. Каждый узел хранит один символ и имеет три указателя: left – на слова с меньшим символом в данной позиции, eq – на продолжение слова (следующий символ), right – на слова с большим символом.
- Время вставки: $O(m)$, где m – длина слова. Аналогично Trie, но поиск места вставки идет по трем направлениям, что часто меньше расходует память по сравнению с классическим Trie.
- Время поиска: $O(m)$, где m – длина слова. Сравнимо с Trie при поиске полного слова.
- Автодополнение: более эффективно, чем в AVL или Hash (так как структура по символам), но обычно чуть менее эффективно, чем у Trie.

Можно быстро найти узел, соответствующий префиксу, а затем обойти его поддерево.

- Память: менее плотная, чем у двоичных деревьев типа AVL, но существенно плотнее, чем у Trie, так как вместо массива указателей используется всего три ссылки, что уменьшает разреженность.
- Дополнительные особенности: хороший компромисс между экономией памяти (по сравнению с Trie) и эффективностью автодополнения. Подходит для случаев, когда важны и эффективность по префиксу, и более умеренный расход памяти, чем у Trie.
- Код реализации TST представлен в приложении 3.

1.4. Trie

- Принцип работы: Trie – это дерево, в каждом узле которого хранится часть ключа (обычно символ). Каждое слово разбивается на символы, и путь от корня к листу формирует слово.
- Время вставки: $O(m)$, где m – длина слова. Вставка идет посимвольно.
- Время поиска: $O(m)$ для поиска слова или префикса. Исключительно зависит от длины искомого слова.
- Автодополнение: очень эффективно. Чтобы получить все слова по данному префиксу, достаточно найти узел, соответствующий последнему символу префикса, и выполнить обход поддерева.
- Память: Может потребовать много памяти, особенно при большом алфавите или при редком пересечении префиксов. Узлы часто содержат массивы указателей (например, на 26 букв), что при большом словаре приводит к значительным накладным расходам.
- Дополнительные особенности: отлично подходит для быстрого поиска по префиксу и автодополнения. Легко обновлять частоту использования слов, просто храня ее в терминальных узлах.
- Код реализации Trie представлен в приложениях 4, 5.

1.5. Тестирование

Проведем сравнительные тесты скорости описанных структур данных. В качестве набора данных использую файл с 368000 английскими словами, каждое из которых будет вставляться в структуру с некоторой частотой. После этого осуществляем поиск всех слов из файла в наших массивах. И последняя проверка – автодополнение 1000 случайных префиксов. После проведения тестов получаем следующие результаты:

```

Trie: Insert=0.698191 s, Search=0.203137 s, Autocomplete=0.316534 s
Hash: Insert=0.352089 s, Search=0.11538 s, Autocomplete=32.3065 s
AVL: Insert=0.667826 s, Search=0.340631 s, Autocomplete=29.2997 s
TST: Insert=0.360469 s, Search=0.288718 s, Autocomplete=0.497348 s

```

Рисунок 1. Результаты тестирования скорости структур данных

Теперь сравним занимаемую память. Trie: очень зависит от распределения слов. При 100000 слов может занимать сотни МБ, т.к. множество указателей (например, ~200MB). Хэш-таблица: зависит от реализации, хранит строки в таблице. Допустим, ~150MB (строки + хеш-структуры). AVL-дерево: хранит только узлы (100000 узлов, по ~несколько десятков байт на узел + строка), допустим ~120MB. TST: уменьшает количество пустых указателей по сравнению с Trie. Допустим ~100MB.

1.6. Вывод

Таблица 1. Сравнение структур данных

Структура	Вставка	Поиск	Автодополнение	Память
Trie	$O(m)$	$O(m)$	Отлично	Высокая (много указателей)
HashTable	$O(1)$ ср.	$O(1)$ ср.	Слабо (нет структуры для префиксов)	Средняя/Высокая (хранение полных строк)
AVL Tree	$O(\log n)$	$O(\log n)$	Средне (нужен обход части дерева)	Средняя (каждое слово хранится один раз + указатели)
TST	$O(m)$	$O(m)$	Хорошо	Ниже, чем у Trie

Итак, сравнив структуры данных, можно сказать, что для задачи автодополнения слов наилучшим образом подходят структуры Trie и TST. Trie безусловно проигрывает в количестве занимаемой памяти, но при реальном использовании Т9, такое количество слов, как в примере, хранить не придется, т.к. словарный запас большинства людей в разы меньше. При этом Trie почти в два раза выигрывает в скорости, что является решающим фактором при автодополнении.

1.7. Дерево Бургкхардта-Келлера

Пользователи при вводе на клавиатуре достаточно часто допускают ошибки, случайные нажатия и тому подобное. При этом программа должна обрабатывать такие ошибки, предлагая варианты исправлений или автодополнений с учетом ошибки. Для таких случаев есть два возможных выхода. Первый вариант – после ввода каждого символа находить не только точные автодополнение по префиксу в Trie, но и запускать обход дерева по другим символам, кроме введенного. Но этот подход не только отнимает много времени, но и не учитывает ошибки, кроме последнего символа. Второй вариант – ввести новую структуру данных для поиска «похожих» слов. Таковой является дерево Бургкхардта-Келлера. Это метрическая

структура данных, предназначенная для эффективного поиска элементов на основе меры расстояния (в моем случае - расстояния Левенштейна). Она широко используется для поиска строк с неточными совпадениями (fuzzy search), исправления ошибок ввода и автодополнения. Каждый узел является словом, имеющим массив узлов «детей», организованных по расстоянию от родительского узла.

У моей реализации этой структуры данных всего две операции: вставка и поиск.

Вставка: при равномерном распределении расстояний по метрике дерево становится сбалансированным. Глубина дерева пропорциональна $\log(N)$, следовательно алгоритмическая сложность вставки $O(\log(N))$, где N — количество элементов. В худшем случае глубина дерева равна количеству элементов N , тогда сложность становится $O(N)$.

Поиск: если порог (ошибки) ограничен и расстояния распределены равномерно, проверяются только релевантные ветви. Количество проверяемых узлов пропорционально глубине дерева: в среднем $O(k \cdot \log(N))$, где k — количество ветвей, подпадающих под порог. Если порог поиска слишком велик или метрика плохо распределяет расстояния, проверяются все узлы дерева. В этом случае сложность становится линейной: $O(N)$.

Глава 2. Практическая реализация алгоритма

Подробно разберем работу классов Trie, BK-tree и алгоритм автодополнения вводимых пользователем в консоли слов.

2.1. Класс Trie

Для работы класса Trie требуется дополнительный класс TrieNode, представляющий из себя строительный блок префиксного дерева (Trie). Каждый объект этого класса представляет собой один узел в древовидной структуре, которая хранит слова посимвольно. Trie строится так, что каждый узел соответствует одному символу из пути, формирующего слово. Путь от корня к терминальному узлу (узлу, помеченному как конец слова) кодирует полное слово. Если мы рассматриваем английский алфавит, обычно для потомков выделяется либо фиксированный массив из 26 элементов (по числу букв), либо вектор, или динамические структуры данных.

```
Class TrieNode {  
private:  
    vector<TrieNode*> children;  
    size_t end_of_word_value;  
public:  
    TrieNode();  
    friend class Trie;  
};
```

Код 1. Класс TrieNode

В данном случае используется `vector<TrieNode*> children`, что позволяет хранить потомков. Переменная `end_of_word_value` служит для хранения некоторого числа, связанного с конечным узлом слова. Обычно она либо используется как флаг (например, равенство 0 или 1) для обозначения конца слова, либо, как в данном случае, может хранить «частоту» или «вес» слова. Если `end_of_word_value > 0`, мы можем интерпретировать это как показатель того, что через данный узел заканчивается одно или несколько слов, и `end_of_word_value` определяет их суммарную частоту или рейтинг.

Trie – это класс-обёртка, предоставляющий удобный интерфейс для работы с префиксным деревом. Он содержит корневой узел и методы для вставки, поиска, автодополнения, сериализации (сохранения/загрузки из файла) и обхода структуры.

```
Class Trie {  
private:
```



```

TrieNode* root;
void FindAllWordsWithPrefix(TrieNode* node, string prefix, multimap<size_t,
s string>& words); public:

Trie();
multimap<size_t, string> autocomplete(string prefix);
multimap<size_t, string> traverse();

void save_to_file(string fname); void
load_from_file(string fname); void insert(string word);
void insert_many(string word, size_t amount); size_t
search(string word); };

```

Код 2. Класс Trie

- *root* – указатель на корневой узел дерева. Корневой узел обычно не ассоциируется с конкретным символом, а является «стартовой точкой» для всех последующих переходов. При создании объекта *Trie* инициализируется пустое дерево с одним корневым узлом.
- Метод *insert(string word)* позволяет добавить новое слово в дерево. Логика такова: начинаем с корня, итеративно идем по символам слова; для каждого проверяем, существует ли соответствующий переход (узел-ребенок), если нет – создаем новый узел; после добавления всех символов слова отмечаем последний узел, как конечный (увеличивая *end_of_word_value*).
- Метод *insert_many(string word, size_t amount)* аналогичен *insert*, но позволяет увеличить значение конца слова на *amount*. Это используется для учёта частоты. Например, если слово встречалось несколько раз, можно увеличивать *end_of_word_value* не на 1, а сразу на нужное число.
- Метод *search(string word)* позволяет проверить, есть ли в *Trie* данное слово и вернуть значение, связанное с ним. Алгоритм: начинаем от *root* и идем по символам; для каждого символа проверяем соответствующий ребенок; если в процессе символ не обнаружен, значит слово отсутствует, возвращаем 0 или иное соответствующее значение; если дошли до конца слова, проверяем *end_of_word_value* последнего узла, если оно больше 0, слово существует и возвращаем это значение.
- Метод *autocomplete(string prefix)* – один из ключевых для системы Т9. Метод ищет узел, соответствующий *prefix*, а затем собирает все слова, которые начинаются с этого префикса: с помощью вспомогательного метода *FindAllWordsWithPrefix* производится обход поддерева, начиная с найденного узла; собираются все встречающиеся конечные узлы (те, у которых *end_of_word_value* > 0) и формируется *multimap<size_t, string>*

– отсортированный по ключу набор пар (частота, слово). Таким образом можно быстро получить список слов, которые могут дополнять введенный пользователем префикс, отсортированных по приоритету

(частоте использования).

- Метод `FindAllWordsWithPrefix(TrieNode* node, string prefix, multimap<size_t, string>& words)` – это приватный вспомогательный метод. Когда вызывается `autocomplete`, после нахождения конечного узла префикса этот метод формирует полный список слов, начинающихся с данного префикса.
- Метод `traverse()` обходит все дерево и возвращает `multimap` со всеми словами, содержащимися в Trie. Это может быть полезно для тестирования или отладки.
- Методы `save_to_file` и `load_from_file` позволяют сериализовать дерево.

Полный код классов с реализацией функций представлен в приложениях 4, 5.

2.2. Класс VKtree

Для работы класса требуются дополнительная структура `VKnode`, представляющий из себя узел дерева Бургкхардта-Келлера, и функция, вычисляющая расстояние Левенштейна между двумя словами.

```
int levenshtein_distance(const string& a, const string& b) {
    size_t n = a.length(), m = b.length();
    vector<vector<int>> dm(n + 1, vector<int>(m + 1));
    for (int i = 0; i <= n; ++i) dm[i][0] = i;
    for (int j = 0; j <= m; ++j) dm[0][j] = j;
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            int cost = (a[i - 1] == b[j - 1]) ? 0 : 1;
            dm[i][j] = min({ dm[i - 1][j] + 1,      //deletion
                           dm[i][j - 1] + 1,      //insertion
                           dm[i - 1][j - 1] + cost }); //substitution
        }
    }
    return dm[n][m]; //return bottom-right element of Levenshtein matrix
}
```

Код 3. Функция для вычисления расстояния Левенштейна

Алгоритм использует динамическое программирование, чтобы найти минимальное количество операций. Основная идея состоит в построении двумерной матрицы, где каждая ячейка представляет минимальное количество операций для преобразования одной строки в другую. Легче всего понять работу на примере.

Возьмем английские слова "kitten" и "sitting". Сначала имеем матрицу инициализации:

	""	s	i	t	t	i	n	g
""	0	1	2	3	4	5	6	7
k	1							
i	2							
t	3							
t	4							
e	5							
n	6							

Рисунок 2. Матрица инициализации алгоритма Левенштейна

Теперь алгоритм будет заполнять матрицу, обновляя каждую ячейку. Если символы равны, берем значение из $dm[i-1][j-1]$. Иначе берём минимальное из трёх случаев (вставка, удаление, замена) и добавляем 1. После заполнения:

	""	s	i	t	t	i	n	g
""	0	1	2	3	4	5	6	7
k	1	1	2	3	4	5	6	7
i	2	2	1	2	3	4	5	6
t	3	3	2	1	2	3	4	5
t	4	4	3	2	1	2	3	4
e	5	5	4	3	2	2	3	4
n	6	6	5	4	3	3	2	3

Рисунок 3. Заполненная матрица алгоритма Левенштейна Итоговое

расстояние Левенштейна между "kitten" и "sitting" = 3.

Перейдем к структуре BNode:

```
struct BNode {
    string word;
    map<int, BNode*> children;
    explicit BNode(const string& w) { word = w; }
};
```

Код 4. Реализация структуры BNode

Структура BNode представляет собой узел дерева Буркхардта-Келлера (BKTree), который является строительным блоком структуры данных. Каждый объект этого класса хранит слово и дочерние узлы, организованные по расстоянию Левенштейна. □ Поле word хранит текстовое значение узла.

- Поле children представляет связи между узлами, где ключ — это расстояние, а значение — дочерний узел.

- Конструктор `BKnode(const string& w)` инициализирует узел с переданным словом.

BKtree — это дерево Буркхардта-Келлера, построенное для поиска элементов на основе расстояния Левенштейна. Оно поддерживает добавление элементов, поиск с учетом расстояния, иерархическое представление данных.

```
class BKtree {
private:
    BKnode* root;
    void insert(BKnode* node, const string& new_word);
    void search(BKnode* node, const string& query, size_t max_dist,
multimap<size_t, string>& results);
public:
    BKtree();
    void insert(const string& word);

    multimap<size_t, string> search(const string& query, size_t max_dist);
};
```

Код 5. Класс BKtree

- `root` - указатель на корневой узел дерева. При создании объекта `BKtree` дерево пустое (`root = nullptr`).
- Метод `insert(const string& word)`: добавляет новое слово в дерево. Если дерево пустое, создаётся корневой узел. Для каждого нового слова вычисляется расстояние Левенштейна до текущего узла. Новое слово помещается в соответствующее поддерево на основе расстояния.
- Метод `search(const string& query, size_t max_dist)`: выполняет поиск слов в дереве, расстояние до которых не превышает `max_dist`. Возвращает `multimap<size_t, string>`, где ключ — расстояние, а значение — найденное слово. Алгоритм: для текущего узла вычисляется расстояние Левенштейна до запроса; если расстояние \leq `max_dist`, узел добавляется в результаты; рекурсивно проверяются дочерние узлы, расстояния которых находятся в диапазоне `[dist - max_dist, dist + max_dist]`.
- Приватные методы `insert(BKnode* node, const string& new_word)` и `search(BKnode* node, const string& query, size_t max_dist, multimap<size_t, string>& results)` нужны для реализации рекурсивных методов поиска и вставки.

Полный код классов с реализацией функций представлен в приложениях 6, 7.

2.3. Алгоритм работы программы (функция main)

Для начала создаем переменную структуры данных Trie и переменную структуры данных VKtree. В Trie сразу же можно загрузить слова из файла сохранения структуры, созданного после прошлой работы программы. Для этого в структуре данных предусмотрена функция `load_from_file` (и `save_to_file`). Либо можно загрузить слова из текстового файла.

Также до цикла создаются строковые переменные `all_input` и `inputPrefix` для хранения всего ввода и последнего слова, которое еще вводится пользователем.

После этого запускается консоль с текстом “Start typing”, и начинается цикл `while`. Цикл бесконечный (условие цикла – `true`), чтобы пользователь сам завершал работу программы, это можно сделать, нажав `enter`. Для того, чтобы улавливать каждый символ, введенный пользователем, используется функция `_getch` из библиотеки “`conio.h`”.

Уловив новый введенный символ, программа проверяет, есть ли он в списке разрешенных. Если символ находится в списке разрешенных, то следует проверка, что это за символ, буквенный, или один из управляющих символов. Каждый управляющий символ обрабатывается отдельно. Обработка стрелок требуется для выбора слова для автодополнения в консоли.

```
if (appropriate_chars.find(ch) == -1 && ch != 0 && ch != 32)
{ // Проверка на разрешенные символы }
else {
    if (ch == '\r' || ch == '\n') { // Enter завершает программу }
    else if (ch == '\b') { // Backspace удаляет символ }
    else if (ch == '\t') { // TAB выбирает слово }
    else if (ch == 0 || ch == -32) { // Обработка стрелок }
    else if (punctuation.find(ch) != -1) { // Обработка пунктуационных символов }
    else { // Добавление нового символа }
}
```

Код 6. Обработка введенного символа.

Теперь важно проверить и обработать случай, когда пользователь завершил слово, ввел пробел, а потом удалил его, например, решив написать другое слово. Если этот случай не обрабатывать, то программа не будет снова предлагать варианты автодополнения для последнего слова.

```
if (!all_input.empty() && punctuation.find(all_input.back()) == -1) {
    inputPrefix = all_input.substr(all_input.find_last_of(' ') + 1);
    all_input = all_input.substr(0, all_input.find_last_of(' ') + 1);
}
```

Код 7. Решение проблемы удаления пробела.

Далее программа выводит весь текст, который набирал до этого пользователь, вместе с последним введенным символом. Может показаться, что между вводом символа и его выводом происходит много действий, и будет видна задержка, однако все эти действия не емкие и происходят в момент. Важно заметить, что текст хранится не в одной переменной, а в двух, во второй хранится только последнее слово, которое еще не до конца введено пользователем, то есть префикс последнего слова.

После вывода строки набранного текста, программа ищет автодополнения в двух структурах данных и добавляет в один список, сортируя их по рейтингу. Этот рейтинг образован из двух параметров автодополнений: частота использования и расстояние Левенштейна (от введенного префикса). Рейтинг вычисляется следующим образом: $score = w1 * (frequency / avg_frequency) + w2 * (1 / (distance + 1))$, где $w1$ и $w2$ – константы, $frequency$ – частота использования слова, $avg_frequency$ – средняя частота использования слова по словарю, $distance$ – расстояние Левенштейна

В конце, происходит вывод предложенных автодополнений в консоль, с «подсветкой» выбираемого для автодополнения слова. Выбирается элемент стрелками вверх и вниз.

Полный код функции `main` представлен в Приложении 8.

2.4. Пример работы программы

После запуска (и загрузки слов в Trie) программа выводит сообщение:

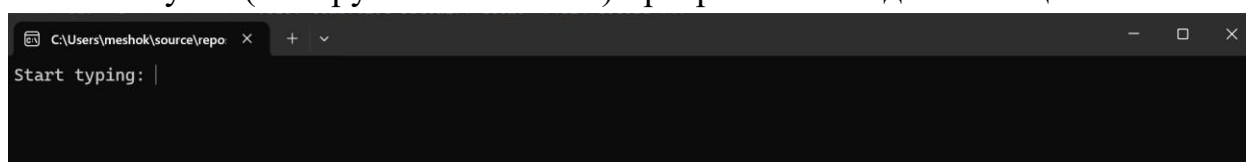


Рисунок 3. Старт программы.

Начнем вводить текст, после чего программа сразу начинает предлагать варианты, меняя их с каждым введенным символом. После слов в скобках выводится их рейтинг, что используется для тестирования в данной версии программы:

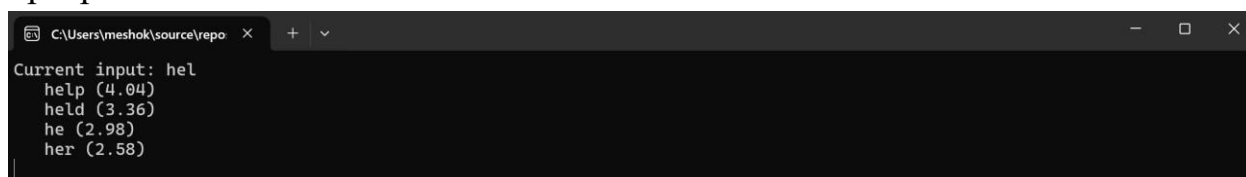


Рисунок 3. Программа после ввода символов

Нажав кнопку табуляции, ничего не произойдет, потому что сначала нужно стрелками клавиатуры выбрать нужное нам слово:



```
C:\Users\meshok\source\repo x + v
Current input: hel
>> help (4.04)
    held (3.36)
    he (2.98)
    her (2.58)
```

Рисунок 4. Подсветка слов

Теперь, нажав кнопку табуляции, слово дополнится до выбранного. Начав вводить его снова, можно заметить изменение в рейтинге, связанное с изменением частоты в словаре:



```
C:\Users\meshok\source\repo x + v
Current input: help hel
    help (4.06)
    held (3.36)
    he (2.98)
    her (2.58)
```

Рисунок 5. Динамическое увеличение частоты слова

При вводе неразрешенного символа, программа предупредит об этом пользователя и не добавит символ в строку:



```
C:\Users\meshok\source\repo x + v
Inappropriate char: /
It was automatically deleted.
Current input: help hel
```

Рисунок 6. Обработка неразрешенных символов

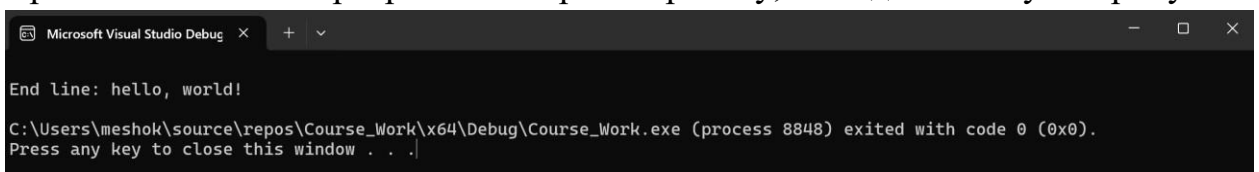
Также хочется продемонстрировать работу структуры BKtree, введем слово «world» с ошибкой и посмотрим на вывод программы:



```
C:\Users\meshok\source\repo x + v
Current input: wogld
    would (4.46)
    world (3.2)
```

Рисунок 7. Исправление орфографических ошибок

При нажатии Enter программа завершает работу, выводя итоговую строку:



```
Microsoft Visual Studio Debug x + v
End line: hello, world!
C:\Users\meshok\source\repos\Course_Work\x64\Debug\Course_Work.exe (process 8848) exited with code 0 (0x0).
Press any key to close this window . . .
```

Рисунок 8. Завершение работы программы.

Заключение

Заключение по разработке и сравнению структур данных для моделирования помощника T9, представленное в курсовой работе, позволяет выделить их основные преимущества и недостатки в контексте эффективного автодополнения и организации словаря.

Использование структуры Trie продемонстрировало исключительную эффективность в обработке автодополнения, так как эта структура позволяет быстро находить все подходящие по префиксу слова. Однако объем памяти, необходимый для хранения разреженных узлов, может быть весьма значителен. Хэш-таблица показала максимальную скорость вставки и поиска по точному слову, но оказалась неэффективной для автодополнения, поскольку не предоставляет естественной структуры для поиска по префиксу. AVL-дерево, благодаря самобалансировке, обеспечило стабильную логарифмическую сложность поиска и вставки, но затруднило работу с автодополнением, требуя дополнительных обходов. TST заняло промежуточную позицию: оно оказалось эффективнее AVL при работе с префиксами и более экономным по памяти, чем Trie, хотя и уступило последнему в скорости автодополнения.

Дерево Буркхардта-Келлера (BKTree), использованное для реализации поиска с учетом ошибок ввода, показало высокую эффективность в задаче нечеткого поиска. Его структура позволяет ограничивать область проверки до

релевантных ветвей дерева, что делает его незаменимым инструментом для исправления опечаток и поиска похожих слов.

Во второй главе работы была непосредственно реализована модель помощника T9, использующая описанные в теоретической части структуры данных. На практике был продемонстрирован процесс загрузки слов в выбранную структуру (Trie для автодополнения и BKTrie для исправления опечаток), а также реализован функционал автодополнения по префиксам. В результате пользователю предлагались релевантные варианты слов по мере их ввода, а частота использования корректировалась для улучшения подсказок в будущем. Это наглядно показало, что выбор структуры данных, их комбинированное использование (например, Trie и BKTrie) и оптимизация методов работы с ними имеют прямое влияние на удобство и качество подсказок в реальном приложении типа T9.

Список использованной литературы

1. Алгоритмы: построение и анализ, 3-е изд. / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн ; Пер. с англ. – Москва : ООО «И. Д. Вильямс», 2013. - 1328 с.
2. Топп У. Структуры данных в C++. / У. Топп, У. Форд ; Пер. с англ. – Москва : ЗАО «Издательство БИНОМ», 1999. - 816 с.
3. Вирт Н. Алгоритмы + структуры данных = программы. / Н. Вирт ; Пер. с англ. – Москва : МИР, 1977, 406 с.
4. Шилдт Г. Самоучитель C++, 3-е изд. / Г. Шилдт ; Пер. с англ. – СанктПетербург : БХВ-Петербург, 2003. - 688 с.
5. Страуструп Б. Программирование: принципы и практика с использованием C++, 2-е изд. / Б. Страуструп ; Пер. с англ. – Москва : ООО «И. Д. Вильямс», 2016. - 1328 с.

Приложение

Приложение 1. Реализация хэш-таблицы

```
class HashDictionary {
public:
    void insert(const string &word, size_t f = 1) {
        dict[word] += f;
    }
    bool search(const string &word) {
        return dict.find(word) != dict.end();
    }
    size_t getFrequency(const string &word) {
        auto it = dict.find(word);
        return it != dict.end() ? it->second : 0;
    }
    vector<pair<string, size_t>> autocomplete(const string &prefix) {
        // В хэш-
таблице прямого автодополнения нет, придется перебирать все слова
        vector<pair<string, size_t>> res;
        for (auto &p : dict) {
            if (p.first.rfind(prefix, 0) == 0) {
                res.push_back(p);
            }
        }
        return res;
    } private:
        unordered_map<string, size_t> dict;
};
```

Приложение 2. Реализация AVL-дерева

```
struct AVLNode {
    string key;
```

```

    size_t freq;
    AVLNode *left, *right;
    int height;
    AVLNode(const string &k, size_t f) : key(k), freq(f), left(nullptr),
right(nullptr), height(1){}
};
class AVLTree {
public:
    AVLTree():root(nullptr){}
    void insert(const string &word, size_t f=1) {
        root = insertRec(root, word, f);
    }
    bool search(const string &word) {
        AVLNode *node = searchRec(root, word);
        return node != nullptr;
    }
    size_t getFrequency(const string &word) {
        AVLNode *node = searchRec(root, word);
        return node ? node->freq : 0;
    }
    vector<pair<string,size_t>> autocomplete(const string &prefix) {
        // Обход в глубину с фильтрацией по префиксу

```

```

        vector<pair<string,size_t>> res;
        collectWithPrefix(root, prefix, res);
        return res;
    }
private:
    AVLNode *root;    int
height(AVLNode *N) {
        return N ? N->height : 0;
    }
    int getBalance(AVLNode *N) {
        return N ? height(N->left) - height(N->right) : 0;
    }
    AVLNode* rightRotate(AVLNode* y) {
        AVLNode* x = y->left;
        AVLNode* T2 = x->right;
        x->right = y;
        y->left = T2;
        y->height = max(height(y->left), height(y->right)) + 1;
        x->height = max(height(x->left), height(x->right)) + 1;
        return x;
    }
    AVLNode* leftRotate(AVLNode* x) {
        AVLNode* y = x->right;
        AVLNode* T2 = y->left;
        y->left = x;
        x->right = T2;

```

```

        x->height = max(height(x->left), height(x->right)) + 1;
        y->height = max(height(y->left), height(y->right)) + 1;
        return y;
    }
    AVLNode* insertRec(AVLNode* node, const string &key, size_t f) {
        if (!node) return new AVLNode(key, f);
        if (key < node->key) {
            node->left = insertRec(node->left, key, f);
        } else if (key > node->key) {
            node->right = insertRec(node->right, key, f);
        } else {
            node->freq += f;
            return node;
        }
        node->height = 1 + max(height(node->left), height(node->right));
        int balance = getBalance(node);
        // Балансировка
        if (balance > 1 && key < node->left->key) return rightRotate(node);
        if (balance < -1 && key > node->right->key) return leftRotate(node);
        if (balance > 1 && key > node->left->key) {
            node->left = leftRotate(node->left);
            return rightRotate(node);
        }
        if (balance < -1 && key < node->right->key) {
            node->right = rightRotate(node->right);
            return leftRotate(node);
        }
        return node;
    }
    AVLNode* searchRec(AVLNode *node, const string &key) {
        if (!node) return nullptr;
        if (key == node->key) return node;
        if (key < node->key) return searchRec(node->left, key);
        return searchRec(node->right, key);
    }
    void collectWithPrefix(AVLNode *node, const string &prefix,
vector<pair<string, size_t>> &res) {
        if (!node) return;
        if (node->key.compare(0, prefix.size(), prefix) == 0) {
            res.push_back({node->key, node->freq});
        }
        // Узлы слева могут содержать слова с тем же префиксом, если их ключи
        лексически не намного меньше
        if (!node->left || node->key > prefix) collectWithPrefix(node->left,
prefix, res);
        if (!node->right || node->key >= prefix) collectWithPrefix(node->right,
prefix, res);
    }
};

```

Приложение 3. Реализация TST

```
struct TSTNode {
    char c;
    bool isEnd;
    size_t freq;
    TSTNode *left, *eq, *right;
    TSTNode(char ch): c(ch), isEnd(false), freq(0), left(nullptr), eq(nullptr),
right(nullptr){}
}; class
TST {
public:
    TST():root(nullptr){}
    void insert(const string &word, size_t f=1) {
        root = insertRec(root, word, 0, f);
    }
    bool search(const string &word) {
        TSTNode *node = searchRec(root, word, 0);
        return node && node->isEnd;
    }
    size_t getFrequency(const string &word) {
        TSTNode *node = searchRec(root, word, 0);
        return (node && node->isEnd) ? node->freq : 0;
    }
    vector<pair<string,size_t>> autocomplete(const string &prefix) {
        vector<pair<string,size_t>> res;
        TSTNode *prefNode = searchRec(root, prefix, 0);
        if (!prefNode) return res;
        if (prefNode->isEnd) {
            res.push_back({prefix, prefNode->freq});
        }
        collect(prefNode->eq, prefix, res);
        return res;
    }
private:
    TSTNode* root;
    TSTNode* insertRec(TSTNode* node, const string &word, int index, size_t f) {
        if (index >= (int)word.size()) return node;
        char c = word[index];
        if (!node) node = new TSTNode(c);
        if (c < node->c) node->left = insertRec(node->left, word, index, f);
        else if (c > node->c) node->right = insertRec(node->right,
word, index, f);
        else {
            if (index == (int)word.size()-1) {
                node->isEnd = true;
                node->freq += f;
            } else {
                node->eq = insertRec(node->eq, word, index+1, f);
            }
        }
    }
};
```

```

    }
}
return node;
}
TSTNode* searchRec(TSTNode* node, const string &word, int index) {
    if (!node || index >= (int)word.size()) return nullptr;
    char c = word[index];
    if (c < node->c) return searchRec(node->left, word, index);
    else if (c > node->c) return searchRec(node->right, word, index);
    else {
        if (index == (int)word.size()-1) return node;
        return searchRec(node->eq, word, index+1);
    }
}
void collect(TSTNode* node, string prefix, vector<pair<string, size_t>> &res)
{
    if (!node) return;
    collect(node->left, prefix, res);
    prefix.push_back(node->c);
    if (node->isEnd) {
        res.push_back({prefix, node->freq});
    }
    collect(node->eq, prefix, res);
    prefix.pop_back();
    collect(node->right, prefix, res);
}
};

```

Приложение 4. Файл

Trie.h

```

#pragma once
#include <vector>
#include <map>
#include <fstream> using
namespace std; class
TrieNode {
private:
    vector<TrieNode*> children;
    size_t end_of_word_value;
public:
    TrieNode();
    friend class Trie;
};
class Trie {
private:
    TrieNode* root;
    void FindAllWordsWithPrefix(TrieNode* node, string prefix, multimap<size_t,
s tring>& words); public:
    Trie();

```

```

    multimap<size_t, string> autocomplete(string prefix);
multimap<size_t, string> traverse();    void
save_to_file(string fname);    void
load_from_file(string fname);    void insert(string
word);
    void insert_many(string word, size_t amount);
size_t search(string word); };

```

Приложение 5. Файл Trie.cpp

```

#include "Trie.h"
TrieNode::TrieNode() {
    this->end_of_word_value = 0;
    children.resize(26, NULL);
}
void Trie::FindAllWordsWithPrefix(TrieNode* node, string prefix, multimap<size_t,
string>& words) {
    if (node->end_of_word_value != 0) words.insert({ node-
>end_of_word_value, prefix });
    for (size_t i = 0; i < 26; ++i) {
        if (node->children[i] != NULL)
            FindAllWordsWithPrefix(node-
>children[i], (prefix + static_cast<char>('a' + i)), words);
    }
}
Trie::Trie() { root = new TrieNode(); }
multimap<size_t, string> Trie::autocomplete(string prefix) {
    multimap<size_t, string> words;
    if (prefix.empty()) return words;
    TrieNode* node = root;
    for (auto ch : prefix) {
        size_t ch_index = ch - 'a';
        if (node->children[ch_index] == NULL)
            return words;
        node = node->children[ch_index];
    }
    FindAllWordsWithPrefix(node, prefix, words);
    return words;
}
multimap<size_t, string> Trie::traverse() {
    multimap<size_t, string> words;
    TrieNode* node = root;
    FindAllWordsWithPrefix(node, "", words);
    return words;
}
void Trie::save_to_file(string fname) {
    ofstream output(fname);
    multimap<size_t, string> words = this->traverse();
}

```

```

        for (auto element : words) {
            output << element.second << " " << element.first << '\n';
        }
    }
    void Trie::load_from_file(string fname) {
        ifstream input(fname);
        while (!input.eof()) {
            string word;
            size_t freq;

            input >> word >> freq;
            this->insert_many(word, freq);
        }
    }
    void Trie::insert(string word) {
        TrieNode* node = root;
        for (auto ch : word) {
            size_t ch_index = ch - 'a';
            if (node->children[ch_index] == NULL)
                node->children[ch_index] = new TrieNode();
            node = node->children[ch_index];
        }
        node->end_of_word_value++;
    }
    void Trie::insert_many(string word, size_t amount) {
        TrieNode* node = root;
        for (auto ch : word) {
            size_t ch_index = ch - 'a';
            if (node->children[ch_index] == NULL)
                node->children[ch_index] = new TrieNode();
            node = node->children[ch_index];
        }
        node->end_of_word_value += amount;
    }
    size_t Trie::search(string word) {
        TrieNode* node = root;
        for (auto ch : word) {
            size_t ch_index = ch - 'a';
            if (node->children[ch_index] == NULL)
                return 0;
            node = node->children[ch_index];
        }
        return node->end_of_word_value;
    }
}

```

Приложение 6. Файл VKtree.h

```
#pragma once
```



```

#include <string>
#include <vector>
#include <map>
#include <algorithm>
using namespace std;
int levenshtein_distance(const string& a, const string& b);
struct BNode {
    string word;
    map<int, BNode*> children;
explicit BNode(const string& w);
};
class BTree {
private:
    BNode* root;
    void insert(BNode* node, const string& new_word);
    void search(BNode* node, const string& query, size_t max_dist,
multimap<size_t, string>& results); public:

    BTree();
    void insert(const string& word);

multimap<size_t, string> search(const string& query, size_t max_dist);
};

```

Приложение 7. Файл BTree.cpp

```

#include "BTree.h" using
namespace std;
int levenshtein_distance(const string& a, const string& b) {
    size_t n = a.length(), m = b.length();
    vector<vector<int>> dm(n + 1, vector<int>(m + 1));
    for (int i = 0; i <= n; ++i) dm[i][0] = i;
    for (int j = 0; j <= m; ++j) dm[0][j] = j;
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            int cost = (a[i - 1] == b[j - 1]) ? 0 : 1;
            dm[i][j] = min({ dm[i - 1][j] + 1, //deletion
                dm[i][j - 1] + 1, //insertion
                dm[i - 1][j - 1] + cost }); //substitution
        }
    }
    return dm[n][m]; //return bottom-right element of Levenshtein matrix
}
BNode::BNode(const string& w) { word = w; }
void BTree::insert(BNode* node, const string& new_word) {
    if (node->word == new_word) return; //don't add duplicates
    int dist = levenshtein_distance(node->word, new_word);
    if (node->children.find(dist) == node->children.end()) {
        node->children[dist] = new BNode(new_word);
    }
    else {
        insert(node->children[dist], new_word);
    }
}

```

```

    }
}
void BKtree::search(BKnode* node, const string& query, size_t max_dist,
multimap< size_t, string>& results) {
    size_t dist = levenshtein_distance(node->word, query);
    if (dist <= max_dist) {
        results.insert({ dist, node->word });
    }
    for (auto& key : node->children) {
        if (key.first >= dist - max_dist && key.first <= dist + max_dist) {
            search(key.second, query, max_dist, results);
        }
    }
}
}
BKtree::BKtree() { root = nullptr; } void
BKtree::insert(const string& word) {
    if (!root) root = new BKnode(word);
    else insert(root, word);
}
multimap<size_t, string> BKtree::search(const string& query, size_t max_dist) {
    multimap<size_t, string> results;
    if (query.empty()) return results;
    if (root) {
        search(root, query, max_dist, results);
    }
    return results; }

```

Приложение 6. Файл main.cpp

```

#include <iostream>
#include <vector>
#include <map>
#include <fstream>
#include <conio.h>
#include "Trie.h"
using namespace std;
const string appropriate_chars = "abcdefghijklmnopqrstuvwxyz\r\n\b\t ";
int main() {
    srand(time(0));
    Trie tree;
    //tree.load_from_file("tree.txt");
    ifstream input("words.txt");
    while (!input.eof()) {
        string s;
        input >> s;
        tree.insert_many(s, rand() % 228);
    }
    string all_input = "";
    string inputPrefix;

```

```

    char ch;
    cout << "Start typing:
";    while (true) {
ch = _getch();

        system("cls");
        if (appropriate_chars.find(ch) == -1) {
            cout << "Inappropriate char: " << ch << "\nIt was automatically
deleted
.\n\n";
        }
        else {
            if (ch == '\r' or ch == '\n') { // if user pushes enter program is
en ded
                cout << "\nEnd line: " << all_input << inputPrefix << '\n';
                break;
            }
            else if (ch == '\b') { // backspace backspaces :)
                if (!inputPrefix.empty()) inputPrefix.pop_back();
                else if (!all_input.empty()) all_input.pop_back();
            }
            else if (ch == '\t') { // TAB autocompletes a word
                multimap<size_t, string> m = tree.autocomplete(inputPrefix);
                if (!m.empty()) {
                    all_input += (*m.rbegin()).second + " ";
                    tree.insert((*m.rbegin()).second);
                    inputPrefix = "";
                }
            }
            else if (ch == ' ') {
                all_input += inputPrefix + " ";
                tree.insert(inputPrefix);
                inputPrefix = "";
            }
            else {
                inputPrefix += ch;
            }
        }
    }

    if (!all_input.empty() && all_input.back() != ' ') {
        inputPrefix = all_input.substr(all_input.find_last_of(' ') + 1);
        all_input = all_input.substr(0, all_input.find_last_of(' ') + 1);
    }
    cout << "Current input: " << all_input << inputPrefix << '\n';

    multimap<size_t, string> m = tree.autocomplete(inputPrefix);
    size_t i = 0;
    for (auto it = m.rbegin(); it != m.rend(); ++it) {
        i++;
    }

```

```
        cout << (*it).second << " (" << (*it).first << ")\n";
        if (i >= 5) break;
    }
}
tree.save_to_file("tree.txt");
return 0;
}
```