МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ИНСТИТУТ МАТЕМАТИКИ И ИНФОРМАЦИОННЫХ СИСТЕМ
ФАКУЛЬТЕТ КОМПЬЮТЕРНЫХ И ФИЗИКО-МАТЕМАТИЧЕСКИХ НАУК
КАФЕДРА ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

ДОПУЩЕН К ЗАЩИТЕ
Заведующий кафедрой ПМИ
_____ Е.В. Разова

“ СРАВНИТЕЛЬНЫЙ АНАЛИЗ АЛГОРИТМОВ ПОИСКА В СЛОВАР “

Выполнил студент группы ФИБ-2301-51-00 / Зуа Анжело Абреу /
Руководитель, доцент /Татаринова Александра Геннадьевна /

Работа защищена с оценкой _____ ____.____.2024 г.

Члены комиссии: _____ / _____ /
_____ / _____ /

КИРОВ 2024

Содержание

Введение

Поиск в словаре является фундаментальной операцией в области информационных технологий, компьютерных наук и алгоритмов. Эффективность алгоритмов поиска в словаре имеет огромное значение для оптимизации процессов обработки данных, поиска информации и решения различных задач например:

- Проверка Орфографии
- Поисковые системы
- Автозаполнение и поиск предложений
- Криптографические приложения
- Игры в слова

Существует множество различных алгоритмов поиска, каждый из которых обладает своими уникальными характеристиками и применим в определенных сценариях. Целью данного исследования является проведение глубокого сравнительного анализа различных алгоритмов поиска в словаре с целью выявления их особенностей, преимуществ, недостатков и областей применения. Мы изучим их временную и пространственную сложность, а также проведем сравнительный анализ производительности в различных условиях.

Полученные результаты позволят определить наиболее эффективные алгоритмы поиска в словаре для конкретных задач и ситуаций, а также выявить возможности для улучшения и оптимизации существующих методов. Таким образом, проблема исследования заключается в необходимости проведения сравнительного анализа различных алгоритмов поиска в словаре с целью выявления их эффективности, производительности и применимости в различных сценариях использования. Несмотря на обширное количество существующих алгоритмов поиска, не всегда ясно, какой из них является наиболее оптимальным для конкретной задачи или набора данных. Поэтому важно провести систематическое исследование, которое позволит сравнить различные алгоритмы поиска и выявить их преимущества и недостатки в различных условиях.

Таким образом, основной задачей исследования является выявление наиболее эффективных алгоритмов поиска в словаре на основе сравнительного анализа, что позволит оптимизировать процессы поиска информации и обработки данных в различных областях применения.

Для достижения поставленной цели необходимо решить следующие задачи:

- Провести обзор алгоритмов поиска в словаре
- Реализовать исследуемые алгоритмы поиска в словаре
- Сформировать наборы данных для проведения экспериментальных исследований
- Выполнить исследование алгоритмов по производительности
- Сделать выводы по полученным результатам

В первой главе должно быть сформулирована задача поиска в словаре, приведено описание алгоритмов, примеры работы, достоинства и недостатки, теоретические оценки

производительности. Во второй главе описываем условия экспериментов, данные и сами эксперименты с результатами и основные выводы по полученным результатам.

Разработка

1 - Главе № 1:

В вычислениях словарь-это структура данных, в которой хранятся пары ключ-значение. Он используется для эффективного хранения и извлечения данных. Каждое значение, хранящееся в словаре, связано с уникальным ключом, который используется для доступа к этому значению. В этой работе мы будем извлекать данные из локального файла с расширением .json, где данные были подготовлены стратегически для того чтобы показать искомый key и value

В области информатики выбор алгоритма и структуры данных для поиска строк в словаре зависит от различных факторов, включая количество данных, структуру данных, порядок данных и вычислительные ресурсы. Рассмотрим три различных подхода: бинарное дерево поиска (Binary Search Tree - BST), последовательный поиск в хэш-таблице (Sequential Search in Hash Table) и дерево Tree-Trie (Trie).

1.1- **Binary Search Tree**

1.1.2 - History:

Алгоритм бинарного дерева поиска был независимо открыт несколькими исследователями, включая П.Ф. Уиндли, Эндрю Дональда Бута, Эндрю Колина, Томаса Н. Хиббарда. Этот алгоритм приписывается Конвею Бернерсу-Ли и Дэвиду Уилеру, которые использовали его для хранения помеченных данных на магнитных лентах в 1960 году. Одним из самых ранних и популярных алгоритмов бинарного дерева поиска является алгоритм Хиббарда. Временные сложности дерева бинарного поиска безгранично возрастают с увеличением высоты дерева, если узлы вставляются в произвольном порядке, поэтому были введены самобалансирующиеся деревья бинарного поиска, чтобы привязать высоту дерева к $O(\log\ n)$. Для ограничения высоты дерева были введены различные бинарные деревья поиска с балансировкой по высоте, такие как AVL-деревья, Treaps и красно–черные деревья. Деревья бинарного поиска также являются фундаментальной структурой данных, используемой при построении абстрактных структур данных, таких как наборы, мультимножества и ассоциативные массивы.

В вычислительной технике, двоичным деревом поиска (по британскому летнему времени), которая также называется упорядоченный и сортировка двоичным деревом, - это корневое двоичное дерево структура данных с ключом каждого внутреннего узла, который превышает все ключи в соответствующий узел в левом поддереве и меньше, чем в его правом поддереве. Временная сложность операций с деревом бинарного поиска линейна по отношению к высоте дерева.

Двоичные деревья поиска позволяют двоичный поиск для быстрого поиска, добавления и удаления элементов данных. Так как узлы по британскому летнему времени размечаются так, чтобы каждый сравнению пропускает около половины из оставшихся на дереве, результативность поиска пропорционален двоичному логарифму. BSTS были

разработаны в 1960-х годах для решения проблемы эффективного хранения помеченных данных и приписываются Конвею Бернерсу-Ли и Дэвиду Уилеру.

1.1.3 - **Search Time in BST for a Dictionary of Words**

Дан словарь слов, рассмотрим линейное время поиска по британскому летнему времени. Предположим, у нас есть словарь с **n** слов. Временная сложность операции поиска в бинарном дереве поиска (по британскому летнему времени) существенно зависит от структуры дерева. В лучшем случае происходит, когда дерево отлично сбалансирован, а в худшем случае происходит, когда дерево совершенно неуравновешенный, напоминающие связанный список.

- Best-Case Scenario (Balanced BST)

  - Предполагая, что дерево достаточно сбалансировано с помощью случайных вставок, время поиска слова составляет приблизительно:

  Time Complexity = O(logn)

- Worst-Case Time Complexity

  - Если по британскому летнему времени совершенно неуравновешенный, время искать слово является:

  Time Complexity = O(n)

1.1.4 - **Tree height**

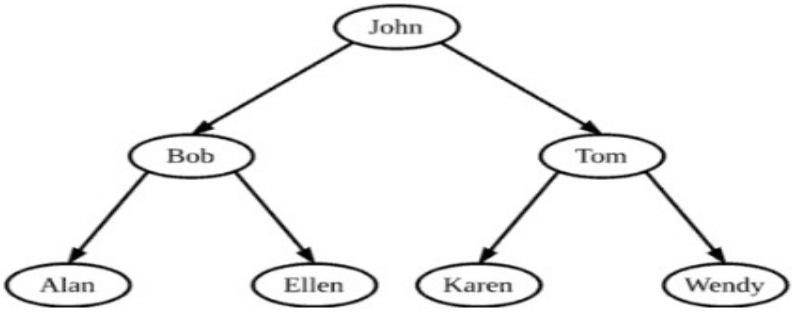
Высота по британскому летнему времени-максимальное число ребер от корневого узла до конечного узла. В худшем случае, если дерево несбалансировано (аналогично связанному списку), высота может составлять N-1, где N - количество узлов (строк) в дереве:

- Best case (balanced tree): height = O (log N)
- Worst Case (Unbalanced Tree): Height = O(N)

1.1.5 - **Storage space**

Пространства для хранения по британскому летнему времени пропорциональна числу узлов в дереве. O(N)

1.1.6 - **BST Struture**



1.1.7 - **Search Pseudo-code**

```
algorithm BSTSearch(node, S):
    // INPUT
    //    node = the current node in the BST
    //    S = the query string to search for
    // OUTPUT
    //    Node if S is found, otherwise a message indicating S is not found

    if node is null:
        return "S not found"
    if S = node.key:
        return node
    if S < node.key:
        return BSTSearch(node.left, S)
    return BSTSearch(node.right, S)
```

2.1 - **Hash table**

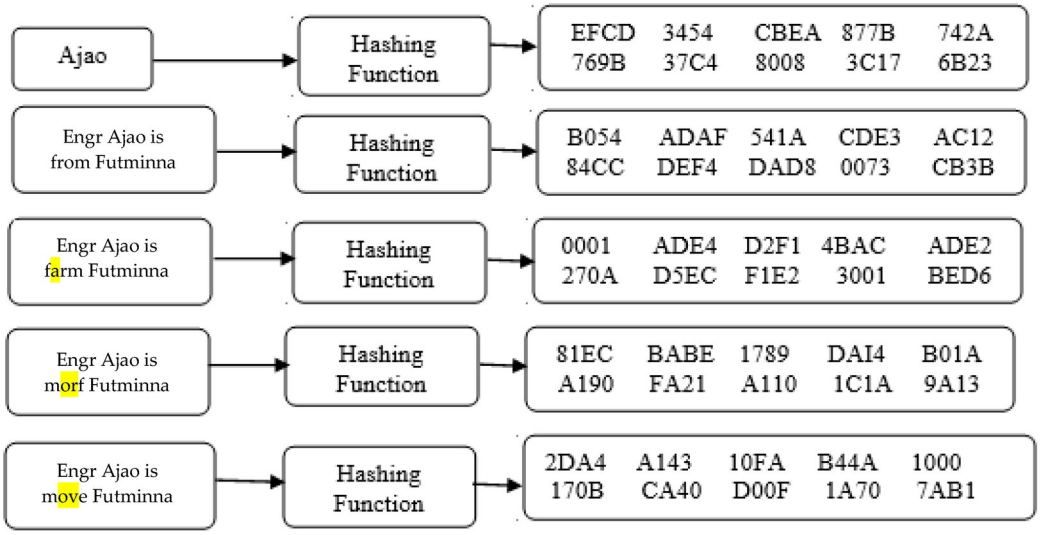
2.1.1 **- History**

Идея хеширования возникла независимо друг от друга в разных местах. В январе 1953 года Ханс Петер Лун написал внутренний меморандум IBM, в котором использовалось хеширование с использованием цепочки. Первый пример открытой адресации был предложен А. Д. Лином, основываясь на меморандуме Луна. Примерно в то же время Джин Амдал, Элейн М. Макгроу, Натаниэль Рочестер и Артур Сэмюэл из IBM Research внедрили хэширование для ассемблера IBM. Открытая адресация с линейным зондированием приписывается Амдалу, хотя Андрею Ершову независимо пришла в голову та же идея. Термин "открытая адресация" был введен У. Уэсли Питерсон в своей статье, в которой обсуждается проблема поиска в больших файлах.

Первая опубликованная работа по хэшированию с использованием цепочки принадлежит Арнольду Дюми, который обсуждал идею использования остатка по модулю

простого числа в качестве хэш-функции. Слово "хэширование" впервые было опубликовано в статье Роберта Морриса. Теоретический анализ линейного зондирования был первоначально представлен Конхаймом и Вайссом. В вычислительной технике хэш-таблица, также известная как хэш-карта или набор хэшей, представляет собой структуру данных, реализующую ассоциативный массив, также называемый словарем, который представляет собой абстрактный тип данных, преобразующий ключи в значения. Хэш-таблица использует хэш-функцию для вычисления индекса, также называемого хэш-кодом, в массиве сегментов или ячеек, из которых можно найти желаемое значение. Во время поиска ключ хэшируется,

и результирующий хэш указывает, где хранится соответствующее значение. В идеале хэш-функция должна присваивать каждому ключу уникальное значение, но в большинстве проектов хэш-таблиц используется несовершенная хэш-функция, что может привести к коллизиям хэшей, когда хэш-функция генерирует один и тот же индекс для нескольких ключей. Такие коллизии обычно каким-то образом устраняются.



В хорошо структурированной хэш-таблице средняя трудоемкость выполнения каждого поиска не зависит от количества элементов, хранящихся в таблице. Многие конструкции хэш–таблиц также допускают произвольные вставки и удаления пар ключ-значение при амортизированных постоянных средних затратах на операцию.

Хэширование - это пример компромисса между пространством и временем. Если объем памяти бесконечен, весь ключ может быть использован непосредственно в качестве индекса для определения его значения при однократном обращении к памяти. С другой стороны, если доступно бесконечное время, значения могут храниться независимо от их ключей, а для извлечения элемента можно использовать двоичный или линейный поиск.

2.1.2 - **Load factor**

Коэффициент загрузки a является критической статистикой хэш-таблицы и определяется следующим образом:

$$\text{load factor } (\alpha) = \frac{n}{m},$$

Где: **n** - это количество записей, занятых в хэш-таблице; **m** - это количество ведер.

Производительность хэш-таблицы ухудшается по отношению к фактору нагрузки α Программное обеспечение, как правило, гарантирует, что коэффициент загрузки α - прежнему ниже некой константы α**max** Это помогает поддерживать хорошую производительность. Поэтому общепринятым подходом является изменение размера или "перефразирование" хэш-таблицы всякий раз, когда значение параметра загрузки α достигает αmax. Аналогично таблице также может быть изменен, если коэффициент загрузки капель α**max/4**

2.1.2 - Hash function

Хэш-функция h: $U \rightarrow$ { 0 ,…,m-1 } отображает вселенную U ключей к индексам или ячейкам в таблице, то есть, $h(x)$ ∈ 0 ,…,m-1 для $x \in U$. Обычные реализации хэш-функций основаны на предположении о целочисленном универсуме, что все элементы таблицы происходят из универсума U = { zero, ...,u-1 } где длина в битах u ограничен размером слова в компьютерной архитектуре. Считается, что хэш-функция идеально подходит для данного набора S , если он является инъективным для S то есть, если каждый элемент $x \in S$ соответствует другому значению в zero,… ,m-1. Идеальная хэш-функция может быть создана, если все ключи известны заранее.

2.1.3 - **Complexy Time**

Временная сложность операций с хэш-таблицей для словарей строк зависит от нескольких факторов, включая качество хэш-функции и способ обработки коллизий. Вот основные операции и их временная сложность:

- Вставка:

  - В среднем случае: O(1); В худшем случае: O(n)

В среднем случае вставка элемента в хэш-таблицу занимает постоянное время, потому что хэш-функция равномерно распределяет записи по таблице. Однако в худшем случае, если многие ключи хэшируются в один и тот же индекс (высокая вероятность коллизий), временная сложность может ухудшиться до O(n), где n — это количество элементов в хэш-таблице.

- Поиск:

  - В среднем случае: O(1); В худшем случае: O(n)

Поиск элемента также занимает постоянное время в среднем, по тем же причинам, что и вставка. В худшем случае поиск может занять линейное время, если все элементы хэшируются в один и тот же индекс и хранятся в виде списка.

2.1.4 - **Factors Affecting Time Complexity**

Качество хэш-функции: Хорошая хэш-функция минимизирует количество коллизий, равномерно распределяя ключи по хэш-таблице. Плохие хэш-функции приводят к большему количеству коллизий, увеличивая вероятность худших сценариев.

- Коэффициент загрузки: Коэффициент загрузки (отношение числа записей к количеству корзин) влияет на производительность. Более высокий коэффициент загрузки означает больше записей на корзину, что может привести к большему числу коллизий и увеличению временной сложности.
- Стратегия разрешения коллизий:

  Цепочки (Chaining): Использует связанные списки для хранения нескольких элементов, которые хэшируются в один и тот же индекс. Временная сложность может ухудшиться до O(n) в худшем случае.
- Открытая адресация (Open Addressing): Использует пробинг для нахождения пустого слота. Распространенные техники включают линейный пробинг, квадратичный пробинг и двойное хэширование. Худшая временная сложность также составляет O(n), но производительность варьируется в зависимости от метода пробинга.

3.1 **- Tree Trie**

3.1.1 - **History**

Идея дерева для представления набора строк была впервые абстрактно описана Акселем Туэ в 1912 году. Деревья были впервые описаны в компьютерном контенте Рене де ла Бриандес в 1959 году.

Идея была независимо описана в 1960 году Эдвардом Фредкином, который ввел термин trie, произнося его /ˈtriː/ (как "дерево") после среднего слога слова "извлечение". Однако другие авторы произносят это слово /ˈtraɪ/ (как "пытаться"), пытаясь отличить его от слова "дерево".

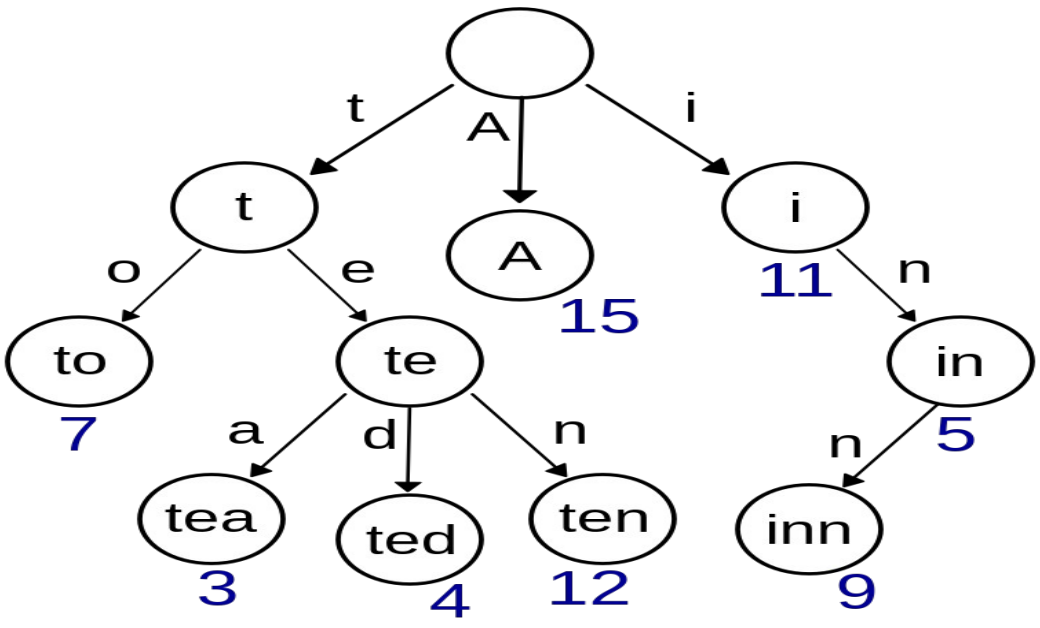
В информатике дерево (/ˈtraɪ/, /ˈtriː/), также называемое цифровым деревом или префиксным деревом, представляет собой тип дерева поиска по k-арному типу, древовидную структуру данных, используемую для поиска определенных ключей в наборе.Эти ключи чаще всего представляют собой строки, причем связи между узлами определяются не всем ключом целиком, а отдельными символами. Чтобы получить доступ к ключу (восстановить его значение, изменить его или удалить), сначала выполняется поиск в глубину по ссылкам между узлами, которые представляют каждый символ в ключе. В

отличие от дерева бинарного поиска, узлы в trie не хранят связанный с ними ключ. Вместо этого позиция узла в trie определяет ключ, с которым он связан. Это распределяет значение каждого ключа по всей структуре данных и означает, что не каждый узел обязательно имеет соответствующее значение. Все дочерние элементы узла имеют общий префикс строки, связанный с этим родительским узлом, а корневой элемент связан с пустой строкой. Задача сохранения данных, доступных по префиксу, может быть выполнена оптимизированным для памяти способом с использованием дерева корней.

Хотя попытки могут быть заданы с помощью символьных строк, это необязательно. Одни и те же алгоритмы могут быть адаптированы для упорядоченных списков любого базового типа, например, для перестановок цифр или фигур. В частности, побитовая последовательность преобразуется в отдельные биты, составляющие часть двоичных данных фиксированной длины, таких как целое число или адрес памяти. Сложность поиска ключа в последовательности остается пропорциональной размеру ключа. Специализированные реализации trie, такие как сжатые попытки, используются для удовлетворения огромных требований к пространству trie в наивных реализациях.

3.1.2 - **Implementation strategies**

Попытки могут быть представлены несколькими способами, соответствующими различным компромиссам между использованием памяти и скоростью выполнения операций. Использование вектора указателей для представления дерева занимает огромное пространство; однако объем памяти может быть уменьшен за счет времени выполнения, если для каждого вектора узлов используется односвязный список, поскольку большинство записей вектора содержит nil. Técnicas como a redução do alfabeto podem aliviar a alta complexidade do espaço, reinterpretando a string original como uma string longa sobre um



alfabeto menor, ou seja, uma string de n bytes pode alternativamente ser considerada como uma string de 2n unidades de quatro bits e armazenadas em um trie com dezesseis ponteiros por nó. No entanto, as pesquisas precisam visitar o dobro de nós no pior dos casos, embora os requisitos de espaço diminuam um fator de oito. outras técnicas incluem o armazenamento de um vetor de 256 ponteiros ASCII como um bitmap de 256 bits representando o alfabeto ASCII, o que reduz drasticamente o tamanho dos nós individuais.

3.1.3 - **Time Complexity of Trie Operations**

Trie (также известный как префиксное дерево) - это специализированная древовидная структура данных, которая используется для хранения динамического набора или ассоциативного массива, где ключами обычно являются строки. Попытки особенно эффективны в сценариях, связанных с поиском ключей по их префиксам, поэтому они широко используются в словарях.

3.1.3.1 - **Вставка** :

- **Временная сложность**: O(L)
- **Объяснение**: Здесь L — длина вставляемой строки. Чтобы вставить строку в Trie, нужно пройти (или создать) до L узлов, где каждый узел представляет символ строки.

3.1.3.2 - **Поиск**:

- **Временная сложность**: O(L)
- **Объяснение**: Аналогично вставке, поиск строки в Trie включает прохождение до L узлов для проверки наличия строки.

3.1.3.3 - **Удаление** :

- **Временная сложность**: O(L)
- **Объяснение**: Удаление включает сначала поиск строки (который занимает O(L)), а затем возможное удаление узлов (до L узлов), если они больше не являются частью какой-либо другой строки в Trie.

3.1.3.1 - **Поиск по префиксу**:

- **Временная сложность**: O(L)
- **Объяснение**: Поиск всех ключей, начинающихся с заданного префикса, также требует прохождения до L узлов, где L — длина префикса.

3.1.4 - Пространственная сложность

Пространственная сложность Trie зависит от количества узлов, которые оно содержит. В худшем случае пространственная сложность может быть представлена как O(AL), где:

- A — размер алфавита.
- L — средняя длина строк в Trie.

Каждый узел в Trie может иметь до A детей, представляющих каждый символ алфавита.

2 - Главе № 2

2.1 - **Implementation Binary Search Tree**

```
#pragma once
#ifndef binary_search_algaritmo

#include <iostream>
#include <string>
#include "../headers/json.hpp"
using namespace std;
using json = nlohmann::json;
// Estrutura do nó da árvore
struct Node {
	string val;
	json meanings;
	Node* left;
	Node* right;
	Node(const string& key, const json& mean) : val(key), meanings(mean), left(nullptr), right(nullptr) {}
};
// Classe da árvore binária de busca
class BinarySearchTree {
public:
	BinarySearchTree() : root(nullptr) {}

	void insert(const string& key, const json& meanings) {
		if (root == nullptr) {
			root = new Node(key, meanings);
		} else {
			_insert(root, key, meanings);
		}
	}

	bool search(const string& key) {
```

```
                int position = 1;
                Node* result = _search(root, key, position);
                if (result != nullptr) {
                        std::cout << "\tMeanings:" << endl;
                for (auto& meaning : result->meanings.items()) {
                        std::cout<<"\t\t"<< meaning.key() << ": " << meaning.value()[1] << endl;
                }
                std::cout << "\tPosition in tree: \n\t\t" << position << endl;
                        return true;
                } else {
                        std::cout << "Key don't found: " << key << endl;
                        return false;
                }
        }
        void inorder() {
                _inorder(root);
        }
private:
        Node* root;
        void _insert(Node* root, const string& key, const json& meanings) {
                if (key < root->val) {
                        if (root->left == nullptr) {
                                root->left = new Node(key, meanings);
                        } else {
                                _insert(root->left, key, meanings);
                        }
                } else {
                        if (root->right == nullptr) {
                                root->right = new Node(key, meanings);
                        } else {
                        _insert(root->right, key, meanings);
```

```
                }
        }
}

Node* _search(Node* root, const string& key, int& position) {
        if (root == nullptr || root->val == key) {
                return root;
        }
        position++;
        if (key < root->val) {
                return _search(root->left, key, position);
        } else {
                return _search(root->right, key, position);
        }
}

void _inorder(Node* root) {
        if (root != nullptr) {
                _inorder(root->left);
                std::cout << root->val << ": " << endl;
                for (auto& meaning : root->meanings.items()) {
                        std::cout << meaning.key() << ": " << meaning.value()[1] << endl;
                }
                _inorder(root->right);
        }
        }
};
#endif
```

2.1.1 - **Implementation Test Binary Search Tree**

```
void testeAppSize_30kB(){

    BinarySearchTree bst;

    string filename = "data/DX.json";

    cout << "TESTE WITH 1526 LINE"<<endl;


    auto start = std::chrono::high_resolution_clock::now();

    // Carregando palavras do arquivo e inserindo na árvore

    loadWordsFromJsonApp1(filename, bst);

    auto end = std::chrono::high_resolution_clock::now();

    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

    cout << "\n\nTempo de execução for load the data and insert in the struture: " << duration.count()<< "

    microseconds.\n\              n" << endl;

    cout << "Busca por 'X': in the start of tree" << endl;

    start = std::chrono::high_resolution_clock::now();

    bst.search("X");

    end = std::chrono::high_resolution_clock::now();

    duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

    cout << "Tempo de execução for search the start in the tree: " << duration.count()<< " microseconds.\

    n\n" << endl;

    cout << endl;


    // Buscando palavras no dicionário

    cout << "Busca por 'XYRIDACEAE': in the center of tree" << endl;

    start = std::chrono::high_resolution_clock::now();

    bst.search("XYRIDACEAE");

    end = std::chrono::high_resolution_clock::now();

    duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

    cout << "Tempo de execução for search the center in the tree: " << duration.count()<< " microseconds.\

    n\n" << endl;
```

```
                cout << endl;

                cout << "Busca por 'XX': in the and the tree" << endl;

                start = std::chrono::high_resolution_clock::now();

                bst.search("XX");

                end = std::chrono::high_resolution_clock::now();

                duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

                cout << "Tempo de execução for search the and in the tree: " << duration.max << " microseconds.\n" <<
endl;

                cout << endl
}
```

result:

```
zua@zua-HP-Laptop:~/Documentos/GitHub/work/code$ time ./main1
TESTE WITH 1526 LINE


Tempo de execução for load the data and insert in the struture: 9549 microseconds.


Busca por 'X': in  the start of tree
        Meanings:
                2: "the 24th letter of the Roman alphabet"
        Position in tree:
                1
Tempo de execução for search the start in the tree: 138 microseconds.


Busca por 'XYRIDACEAE': in the center of tree
        Meanings:
                1: "plants of tropical to temperate regions; usually in wet places"
        Position in tree:
                107
Tempo de execução for search the center in the tree: 133 microseconds.


Busca por 'XX': in the and the tree
        Meanings:
                2: "(genetics) normal complement of sex chromosomes in a female"
        Position in tree:
                83
Tempo de execução for search the and in the tree: 1 microseconds.


real    0m0,824s
user    0m0,811s
sys     0m0,008s
zua@zua-HP-Laptop:~/Documentos/GitHub/work/code$
```

File with Size 30 Kb 1526 Line	Time (s)
In the start	0,000138 s
In the center	0,000187 s
In the and	0,000083 s
Insert data in the Struture	0,000954 s
Time run application	0,00224 s

```
// file with 150000 line and size 5MB
void testeAppSize_5MB(){
        BinarySearchTree bst;
        string filename = "data/DP.json";
        cout << "\n\nTESTE WITH 150000 LINE"<<endl;
        auto start = std::chrono::high_resolution_clock::now();
        // Carregando palavras do arquivo e inserindo na árvore
        loadWordsFromJsonApp1(filename, bst);
        auto end = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        cout << "\nTempo de execução for load the data and insert in the struture: " << duration.count()<< "
microseconds.\n\n" << endl;
        cout << "Busca por 'PACE': in the start of tree" << endl;
        start = std::chrono::high_resolution_clock::now();
        bst.search("PACE");
        end = std::chrono::high_resolution_clock::now();
        duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        cout << "Tempo de execução for search the start in the tree: " << duration.count()<< " microseconds.\n\n" <<
endl;
        cout << endl;
        // Buscando palavras no dicionário
        cout << "Busca por 'PLAYER': in the center of tree" << endl;
        start = std::chrono::high_resolution_clock::now();
        bst.search("PLAYER");
        end = std::chrono::high_resolution_clock::now();
```

```
        duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

        cout << "Tempo de execução for search the center in the tree: " << duration.count()<< " microseconds.\n\n" <<
endl;

        cout << endl;

        cout << "Busca por 'PINER': in the and the tree" << endl;

        start = std::chrono::high_resolution_clock::now();

        bst.search("PINER");

        end = std::chrono::high_resolution_clock::now();

        duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

        cout << "Tempo de execução for search the and in the tree: " << duration.max << " microseconds.\n" << endl;

        cout << endl;
}
```

result:

```
zua@zua-HP-Laptop:~/Documentos/GitHub/work/code$ time ./main1

TESTE WITH 150000 LINE

Tempo de execução for load the data and insert in the struture: 6265356 microseconds.


Busca por 'PACE': in  the start of tree
        Meanings:
                1: "walk with slow or fast paces"
                2: "go at a pace"
                3: "measure (distances) by pacing"
                4: "regulate or set the pace of"
        Position in tree:
                14
Tempo de execução for search the start in the tree: 71 microseconds.


Busca por 'PLAYER': in the center of tree
        Meanings:
                1: "a person who participates in or is skilled at some game"
                4: "a person who pursues a number of different social and sexual partners simultaneously"
                5: "an important participant (as in a business deal)"
        Position in tree:
                4837
Tempo de execução for search the center in the tree: 691 microseconds.


Busca por 'PINER': in the and the tree
        Meanings:
                1: "In England, a rather strong breeze from the north or northeast."
                2: "(in Tasmania) a person employed in felling Huon pines and transporting the timber."
        Position in tree:
                4115
Tempo de execução for search the and in the tree: 1 microseconds.


real    0m6,281s
user    0m6,089s
sys     0m0,060s
zua@zua-HP-Laptop:~/Documentos/GitHub/work/code$
```

File with Size 5 MB and 150000 Line	Time (s)
In the start	0,000071 s
In the center	0,000691 s
In the and	0,000001 s
Read and Insert data in the Struture	0,60 s
Time run application	0,62 s

```
// file with 1526 line and size 40MB
void testeAppSize_40MB(){
BinarySearchTree bst;
string filename = "data/Data.json";
cout << "\n\nTESTE WITH 2000000 LINE"<<endl;
auto start = std::chrono::high_resolution_clock::now();
// Carregando palavras do arquivo e inserindo na árvore
loadWordsFromJsonApp1(filename, bst);
auto end = std::chrono::high_resolution_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
cout << "\nTempo de execução for load the data and insert in the struture: " << duration.count()<< " microseconds.\n\n"
<< endl;
cout << "Busca por 'AARON': in the start of tree" << endl;
start = std::chrono::high_resolution_clock::now();
bst.search("AARON");
end = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
cout << "Tempo de execução for search the start in the tree: " << duration.count()<< " microseconds.\n\n" << endl;
cout << endl;

// Buscando palavras no dicionário
cout << "Busca por 'LIMITATION': in the center of tree" << endl;
start = std::chrono::high_resolution_clock::now();
bst.search("LIMITATION");
end = std::chrono::high_resolution_clock::now();
```

```
duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

cout << "Tempo de execução for search the center in the tree: " << duration.count()<< " microseconds.\n\n" << endl;

cout << endl;

cout << "Busca por 'PINER': in the and the tree" << endl;

start = std::chrono::high_resolution_clock::now();

bst.search("ZWORYKIN");

end = std::chrono::high_resolution_clock::now();

duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

cout << "Tempo de execução for search the and in the tree: " << duration.max << " microseconds.\n" << endl;

cout << endl;

}
```

File with Size 40 MB and 2,000,000 Line	Time (s)
In the start	More 5 minute
In the center	More 5 minute
In the and	More 5 minute
Read and Insert data in the Struture	More 5 minute
Time run application	More 5 minute

2.2 - **Implementation Hash Table**

```
#pragma once

#ifndef algaritmo_hash_table_My


#include <iostream>

#include <vector>

#include <list>

#include <string>

#include <fstream>

#include <unordered_map>

#include "../headers/json.hpp"

#include <stdexcept>


using json = nlohmann::json;
```

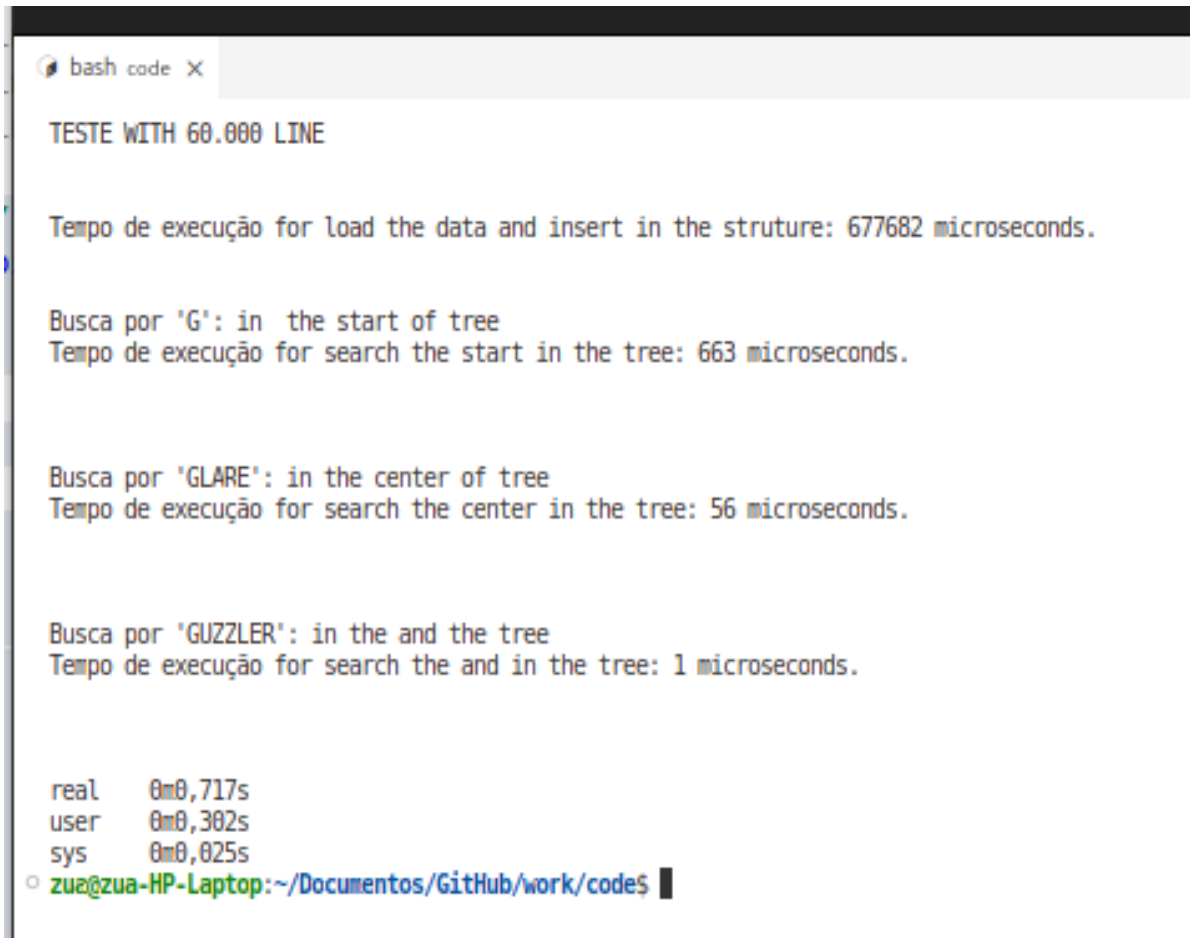
```
using namespace std;



class HashDictionary {
private:
	int numBuckets;
	std::vector<std::list<std::pair<std::string, std::unordered_map<std::string, nlohmann::json>>>> table;

	int hashFunction(const std::string& key) {
		int hash = 0;
		for (char c : key) {
			hash += c;
		}
	return hash % numBuckets;
}
public:
	HashDictionary(int size) {
		numBuckets = size;
		table.resize(numBuckets);
	}

	void insert(const std::string& key, const std::unordered_map<std::string, nlohmann::json>& value) {
		int index = hashFunction(key);
		for (auto& kv : table[index]) {
			if (kv.first == key) {
				kv.second = value;
				return;
			}
		}
		table[index].emplace_back(key, value);
	}
```

```
        void printMeanings(const nlohmann::json& meanings) {
                for (auto& [key, value] : meanings.items()) {
                        std::cout << "Meaning ID: " << key << std::endl;
                        std::cout << "\tPart of Speech: " << value[0] << std::endl;
                        std::cout << "\tDefinition: " << value[1] << std::endl;
                        std::cout << "\tExamples: " << std::endl;
                                for (const auto& example : value[2]) {
                                        std::cout << "\t\t - " << example << std::endl;
                                }
                        std::cout << "Additional Notes: " << std::endl;
                        for (const auto& note : value[3]) {
                                std::cout << "\t\t - " << note <<std::endl;
                        }
                        cout <<"\n";
                }
        }

        std::unordered_map<std::string, nlohmann::json> search(const std::string& key) {
                int index = hashFunction(key);
                for (const auto& kv : table[index]) {
                        if (kv.first == key) {
                                return kv.second;
                        }
                }
                throw std::runtime_error("Key dont't found");
        }

};
#endif
```

2.2.1 - **Implementation Test Hash Table**

```
// file with 1526 line and size 30KB

void testeApp2Size_1MB(){
        HashDictionary dict = HashDictionary(4024);
        string filename = "data/DG.json";


        cout << "TESTE WITH 60.000 LINE"<<endl;
        auto start = std::chrono::high_resolution_clock::now();
        // Carregando palavras do arquivo e inserindo na árvore
        loadWordsFromJsonApp2(filename, dict);
        auto end = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        cout << "\n\nTempo de execução for load the data and insert in the struture: " << duration.count()<< "
microseconds.\n\n" << endl;
        cout << "Busca por 'G': in the start of tree" << endl;
        start = std::chrono::high_resolution_clock::now();
        dict.search("G");
        end = std::chrono::high_resolution_clock::now();
        duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        cout << "Tempo de execução for search the start in the tree: " << duration.count()<< " microseconds.\n\n" <<
endl;
        cout << endl;


/       / Buscando palavras no dicionário
        cout << "Busca por 'GLARE': in the center of tree" << endl;
        start = std::chrono::high_resolution_clock::now();
        dict.search("GLARE");
        end = std::chrono::high_resolution_clock::now();
        duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
```

```
        cout << "Tempo de execução for search the center in the tree: " << duration.count()<< " microseconds.\n\n" <<
endl;
        cout << endl;



        cout << "Busca por 'GUZZLER': in the and the tree" << endl;
        start = std::chrono::high_resolution_clock::now();
        dict.search("GUZZLER");
        end = std::chrono::high_resolution_clock::now();
        duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        cout << "Tempo de execução for search the and in the tree: " << duration.max << " microseconds.\n" << endl;
        cout << endl;
}
```



result:

File with Size 1 MB and 60,000 Line	Time (s)
In the start	0,000663 s
In the center	0,000663 s
In the and	0,000065 s
Read and Insert data in the Struture	0,677682 s
Time run application	0,0717 s

```
// file with 15921 line and size 300KB

void testeApp2Size_4MB(){

        HashDictionary dict = HashDictionary(10024);

        string filename = "data/DS.json";

        cout << "TESTE WITH 200.000 LINE"<<endl;


        auto start = std::chrono::high_resolution_clock::now();

        // Carregando palavras do arquivo e inserindo na árvore

        loadWordsFromJsonApp2(filename, dict);

        auto end = std::chrono::high_resolution_clock::now();

        auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

        cout << "\n\nTempo de execução for load the data and insert in the struture: " << duration.count()<< "
microseconds.\n\n" << endl;

        cout << "Busca por 'SAARINEN': in the start of tree" << endl;

        start = std::chrono::high_resolution_clock::now();

        dict.search("SAARINEN");

        end = std::chrono::high_resolution_clock::now();

        duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

        cout << "Tempo de execução for search the start in the tree: " << duration.count()<< " microseconds.\n\n" <<
endl;

        cout << endl;

        // Buscando palavras no dicionário

        cout << "Busca por 'SIT': in the center of tree" << endl;

        start = std::chrono::high_resolution_clock::now();
```

```
        dict.search("SIT");
        end = std::chrono::high_resolution_clock::now();
        duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        cout << "Tempo de execução for search the center in the tree: " << duration.count()<< " microseconds.\n\n" <<
endl;
        cout << endl;
        cout << "Busca por 'SZILARD': in the and the tree" << endl;
        start = std::chrono::high_resolution_clock::now();
        dict.search("SZILARD");
        end = std::chrono::high_resolution_clock::now();
        duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        cout << "Tempo de execução for search the and in the tree: " << duration.max << " microseconds.\n" << endl;
        cout << endl;}
```

result:

```
bash code

TESTE WITH 200.000 LINE

Tempo de execução for load the data and insert in the struture: 941905 microseconds.

Busca por 'SAARINEN': in  the start of tree
Tempo de execução for search the start in the tree: 37 microseconds.

Busca por 'SIT': in the center of tree
Tempo de execução for search the center in the tree: 45 microseconds.

Busca por 'SZILARD': in the and the tree
Tempo de execução for search the and in the tree: 1 microseconds.

real    0m0,998s
user    0m0,901s
sys     0m0,036s
zua@zua-HP-Laptop:~/Documentos/GitHub/work/code$
```

File with Size 1 MB and 200,000 Line	Time (s)
In the start	0,000037 s
In the center	0,000045 s
In the and	0,000001 s
Read and Insert data in the Struture	0,941904 s
Time run application	0,998 s

```
// file with 1526 line and size 40MB
void testeApp2Size_40MB(){
        HashDictionary dict = HashDictionary(1000);
        string filename = "data/data.json";
        cout << "TESTE WITH 2.000.000 LINE"<<endl;
        auto start = std::chrono::high_resolution_clock::now();
        // Carregando palavras do arquivo e inserindo na árvore
        loadWordsFromJsonApp2(filename, dict);
        auto end = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        cout << "\n\nTempo de execução for load the data and insert in the struture: " << duration.count()<< "
microseconds.\n\n" << endl;
        cout << "Busca por 'AFFECTION': in the start of tree" << endl;
        start = std::chrono::high_resolution_clock::now();
        dict.search("AFFECTION");
        end = std::chrono::high_resolution_clock::now();
        duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        cout << "Tempo de execução for search the start in the tree: " << duration.count()<< " microseconds.\n\n" <<
endl;
        cout << endl;
        // Buscando palavras no dicionário
        cout << "Busca por 'FLAKE': in the center of tree" << endl;
        start = std::chrono::high_resolution_clock::now();
        dict.search("FLAKE");
        end = std::chrono::high_resolution_clock::now();
        duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
```

```
        cout << "Tempo de execução for search the center in the tree: " << duration.count()<< " microseconds.\n\n" <<
endl;

        cout << endl;

        cout << "Busca por 'ZWINGLI': in the and the tree" << endl;

        start = std::chrono::high_resolution_clock::now();

        dict.search("ZWINGLI");

        end = std::chrono::high_resolution_clock::now();

        duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

        cout << "Tempo de execução for search the and in the tree: " << duration.max << " microseconds.\n" << endl;

        cout << endl;

}
```

```
zua@zua-HP-Laptop:~/Documentos/GitHub/work/code$ time ./main2
TESTE WITH 2.000.000 LINE


Tempo de execução for load the data and insert in the struture: 11203734 microseconds.


Busca por 'AFFECTION': in  the start of tree
Tempo de execução for search the start in the tree: 43 microseconds.



Busca por 'FLAKE': in the center of tree
Tempo de execução for search the center in the tree: 30 microseconds.



Busca por 'ZWINGLI': in the and the tree
Tempo de execução for search the and in the tree: 1 microseconds.



real    0m11,940s
user    0m10,745s
sys     0m0,382s
zua@zua-HP-Laptop:~/Documentos/GitHub/work/code$
```

File with Size 1 MB and 2000,000 Line	Time (s)
In the start	0,000043 s
In the center	0,000030 s
In the and	0,000001 s
Read and Insert data in the Struture	0,11 s
Time run application	0,1203 s

2.3 - **Implementation Tree Trie**

```
#pragma once

#ifndef algaritmo_tree_Trei


#include <iostream>

#include <fstream>

#include <unordered_map>

#include <string>

#include "../headers/json.hpp"


using json = nlohmann::json;


// Classe TrieNode

class TrieNode {

public:

	std::unordered_map<char, TrieNode*> children;

	bool isEndOfWord;

	json meanings; // Armazena significados associados ao final da palavra


	TrieNode() : isEndOfWord(false) {}

};


// Classe Trie

class Trie {

private:

	TrieNode* root;


public:

	Trie() {
```

```
        root = new TrieNode();
    }

// Função para inserir uma palavra na Trie e associar significados
    void insert(const std::string& word, const json& meanings) {
        TrieNode* currentNode = root;
        for (char ch : word) {
            if (currentNode->children.find(ch) == currentNode->children.end()) {
                currentNode->children[ch] = new TrieNode();
            }
            currentNode = currentNode->children[ch];
        }
        currentNode->isEndOfWord = true;
        currentNode->meanings = meanings;
    }

// Função para pesquisar uma palavra na Trie e retornar o nó correspondente
    TrieNode* searchNode(const std::string& word) const {
        TrieNode* currentNode = root;
            for (char ch : word) {
            if (currentNode->children.find(ch) == currentNode->children.end()) {
                return nullptr;
            }
            currentNode = currentNode->children[ch];
        }
        return currentNode->isEndOfWord ? currentNode : nullptr;
    }

    // Função para exibir os significados de uma palavra
    void displayMeanings(const std::string& word) const {
        TrieNode* node = searchNode(word);
```

```
        if (node) {
            std::cout << "Meanings of '" << word << "':\n";
            for (const auto& [key, meaning] : node->meanings.items()) {
                std::cout << "Meaning " << key << ":\n";
                std::cout << " Type: " << meaning[0] << "\n";
                std::cout << " Definition: " << meaning[1] << "\n";
                std::cout << " Synonyms: ";
                if (meaning[2].empty()) {
                    std::cout << "None";
                } else {
                    for (const auto& synonym : meaning[2]) {
                        std::cout << synonym << " ";
                    }
                }
            std::cout << "\n Examples: ";
                if (meaning[3].empty()) {
                    std::cout << "None";
                } else {
                    for (const auto& example : meaning[3]) {
                        std::cout << example << " ";
                    }
                }
            std::cout << "\n";
        }
    } else {
        std::cout << "No meanings found for '" << word << "'\n";
    }
}
    // Destrutor para desalocar memória
    ~Trie() {
        deleteTrie(root);
```

```
        }
private:
        void deleteTrie(TrieNode* node) {
                for (auto& pair : node->children) {
                        deleteTrie(pair.second);
                }
                delete node;
                }
        };
#endif
```

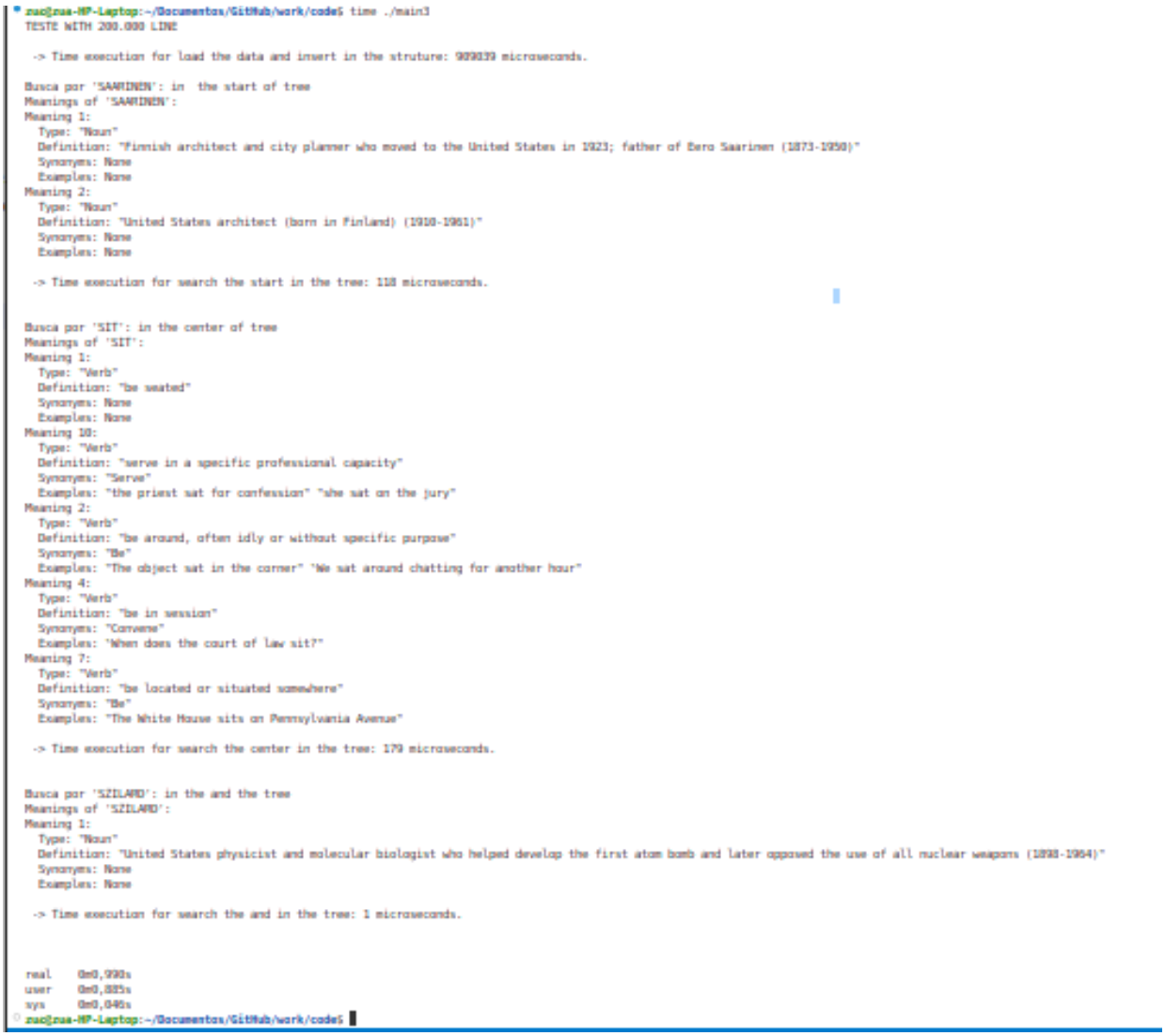
2.3.1 - **Implementation Test Tree Trie**

```
// file with 1526 line and size 1MB

void testeApp3Size_1MB(){
        Trie trie = Trie();
        string filename = "data/DG.json";
        cout << "TESTE WITH 60.000 LINE"<<endl;
        auto start = std::chrono::high_resolution_clock::now();
        // Carregando palavras do arquivo e inserindo na árvore
        try {
                loadWordsFromJsonApp3(filename, trie);
        } catch (const std::exception& e) {
                std::cerr << "Error: " << e.what() << std::endl;
        }
        auto end = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        cout << "\n\n -> Time execution for load the data and insert in the struture: " << duration.count()<< "
microseconds.\n\n" << endl;
        cout << "Busca por 'G': in the start of tree" << endl;
        start = std::chrono::high_resolution_clock::now();
        trie.displayMeanings("G");
```

```
        end = std::chrono::high_resolution_clock::now();
        duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        cout << "\n\n -> Time execution for search the start in the tree: " << duration.count()<< " microseconds.\n\n" <<
endl;
        cout << endl;
        // Buscando palavras no dicionário
        cout << "Busca por 'GLARE': in the center of tree" << endl;
        start = std::chrono::high_resolution_clock::now();
        trie.displayMeanings("GLARE");
        end = std::chrono::high_resolution_clock::now();
        duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        cout << "\n\n -> Time execution for search the center in the tree: " << duration.count()<< " microseconds.\n\n"
<< endl;
        cout << endl;
        cout << "Busca por 'GUZZLER': in the and the tree" << endl;
        start = std::chrono::high_resolution_clock::now();
        trie.displayMeanings("GUZZLER");
        end = std::chrono::high_resolution_clock::now();
        duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        cout << "\n\n -> Time execution for search the and in the tree: " << duration.max << " microseconds.\n" << endl;
        cout << endl;
}
```

result:

```
TESTE WITH 60.000 LINE

 -> Time execution for load the data and insert in the struture: 286637 microseconds.

Busca por 'G': in  the start of tree
Meanings of 'G':
Meaning 5:
  Type: "Noun"
  Definition: "a unit of force equal to the force exerted by gravity; used to indicate the force to which a body is subjected when it is accelerated"
  Synonyms: "Force unit"
  Examples: None
Meaning 9:
  Type: "Noun"
  Definition: "the 7th letter of the Roman alphabet"
  Synonyms: "Letter" "Letter of the alphabet" "Alphabetic character"
  Examples: None

 -> Time execution for search the start in the tree: 98 microseconds.


Busca por 'GLARE': in the center of tree
Meanings of 'GLARE':
Meaning 1:
  Type: "Noun"
  Definition: "a light within the field of vision that is brighter than the brightness to which the eyes are adapted"
  Synonyms: "Brightness"
  Examples: "a glare of sunlight"
Meaning 2:
  Type: "Verb"
  Definition: "be sharply reflected"
  Synonyms: "Reflect" "Shine"
  Examples: "The moon glared back at itself from the lake's surface"
Meaning 3:
  Type: "Verb"
  Definition: "shine intensely"
  Synonyms: "Shine" "Beam"
  Examples: "The sun glared down on us"

 -> Time execution for search the center in the tree: 324 microseconds.


Busca por 'GUZZLER': in the and the tree
Meanings of 'GUZZLER':
Meaning 1:
  Type: "Noun"
  Definition: "someone who drinks heavily (especially alcoholic beverages)"
  Synonyms: "Drinker" "Imbiber" "Toper" "Juicer"
  Examples: "he's a beer guzzler every night"

 -> Time execution for search the and in the tree: 1 microseconds.



real    0m0,322s
user    0m0,264s
sys     0m0,019s
zua@zua-HP-Laptop:~/Documentos/GitHub/work/code$
```



File with Size 1 MB and 60,000 Line	Time (s)
In the start	0,000098 s
In the center	0,000324 s
In the and	0,000001 s
Read and Insert data in the Struture	0,286637 s
Time run application	0,32203 s

```
// file with 200,000 line and size 4MB

void testeApp3Size_4MB(){

        Trie trie = Trie();

        string filename = "data/DS.json";

        cout << "TESTE WITH 200.000 LINE"<<endl;

        auto start = std::chrono::high_resolution_clock::now();
```

```
        // Carregando palavras do arquivo e inserindo na árvore
        loadWordsFromJsonApp3(filename, trie);
        auto end = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        cout << "\n -> Time execution for load the data and insert in the struture: " << duration.count()<< "
microseconds.\n" << endl;
        cout << "Busca por 'SAARINEN': in the start of tree" << endl;
        start = std::chrono::high_resolution_clock::now();
        trie.displayMeanings("SAARINEN");
        end = std::chrono::high_resolution_clock::now();
        duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        cout << "\n -> Time execution for search the start in the tree: " << duration.count()<< " microseconds.\n" <<
endl;
        cout << endl;
        // Buscando palavras no dicionário
        cout << "Busca por 'SIT': in the center of tree" << endl;
        start = std::chrono::high_resolution_clock::now();
        trie.displayMeanings("SIT");
        end = std::chrono::high_resolution_clock::now();
        duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        cout << "\n -> Time execution for search the center in the tree: " << duration.count()<< " microseconds.\n" <<
endl;
        cout << endl;
        cout << "Busca por 'SZILARD': in the and the tree" << endl;
        start = std::chrono::high_resolution_clock::now();
        trie.displayMeanings("SZILARD");
        end = std::chrono::high_resolution_clock::now();
        duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        cout << "\n -> Time execution for search the and in the tree: " << duration.max << " microseconds.\n" << endl;
        cout << endl;
}
```

result

```
zua@zua-HP-Laptop:~/Documentos/GitHub/work/code$ time ./main3
TESTE WITH 2.000.000 LINE


 -> Time execution for load the data and insert in the struture: 8119587 microseconds.


Busca por 'AFFECTION': in  the start of tree
Meanings of 'AFFECTION':
Meaning 1:
  Type: "Noun"
  Definition: "a positive feeling of liking"
  Synonyms: "Feeling"
  Examples: "he had trouble expressing the affection he felt" "the child won everyone's heart" "the warmness of his welcome made us feel right at home"


 -> Time execution for search the start in the tree: 73 microseconds.



Busca por 'FLAKE': in the center of tree
Meanings of 'FLAKE':
Meaning 1:
  Type: "Verb"
  Definition: "form into flakes"
  Synonyms: "Form"
  Examples: "The substances started to flake"
Meaning 2:
  Type: "Verb"
  Definition: "cover with flakes or as if with flakes"
  Synonyms: "Cover"
  Examples: None
```

File with Size 4 MB and 200,000 Line	Time (s)
In the start	0,000118 s
In the center	0,000179s
In the and	0,000001 s
Read and Insert data in the Struture	0,909039 s
Time run application	0,990 s

```
// file with 2.000.000 line and size 40MB

void testeApp3Size_40MB(){

        Trie trie = Trie();

        string filename = "data/data.json";

        cout << "TESTE WITH 2.000.000 LINE"<<endl;
```

```
        auto start = std::chrono::high_resolution_clock::now();
        // Carregando palavras do arquivo e inserindo na árvore
        loadWordsFromJsonApp3(filename, trie);
        auto end = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        cout << "\n -> Time execution for load the data and insert in the struture: " << duration.count()<< "
microseconds.\n" << endl;
        cout << "Busca por 'AFFECTION': in the start of tree" << endl;
        start = std::chrono::high_resolution_clock::now();
        trie.displayMeanings("AFFECTION");
        end = std::chrono::high_resolution_clock::now();
        duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        cout << "\n -> Time execution for search the start in the tree: " << duration.count()<< " microseconds.\n" <<
endl;
        cout << endl;
        // Buscando palavras no dicionário
        cout << "Busca por 'FLAKE': in the center of tree" << endl;
        start = std::chrono::high_resolution_clock::now();
        trie.displayMeanings("FLAKE");
        end = std::chrono::high_resolution_clock::now();
        duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        cout << "\n-> Time execution for search the center in the tree: " << duration.count()<< " microseconds.\n" <<
endl;
        cout << endl;
        cout << "Busca por 'ZWINGLI': in the and the tree" << endl;
        start = std::chrono::high_resolution_clock::now();
        trie.displayMeanings("ZWINGLI");
        end = std::chrono::high_resolution_clock::now();
        duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
        cout << "\n -> Time execution for search the and in the tree: " << duration.max << " microseconds.\n" << endl;
        cout << endl;
```

```
  }
Busca por 'FLAKE': in the center of tree
Meanings of 'FLAKE':
Meaning 1:
  Type: "Verb"
  Definition: "form into flakes"
  Synonyms: "Form"
  Examples: "The substances started to flake"
Meaning 2:
  Type: "Verb"
  Definition: "cover with flakes or as if with flakes"
  Synonyms: "Cover"
  Examples: None


 -> Time execution for search the center in the tree: 84 microseconds.



Busca por 'ZWINGLI': in the and the tree
Meanings of 'ZWINGLI':
Meaning 1:
  Type: "Noun"
  Definition: "Swiss theologian whose sermons began the Reformation in Switzerland (1484-1531)"
  Synonyms: None
  Examples: None


 -> Time execution for search the and in the tree: 1 microseconds.



real    0m9,097s
user    0m8,598s
sys     0m0,312s
zua@zua-HP-Laptop:~/Documentos/GitHub/work/code$
```

File with Size 40MB and 2,000,000 Line	Time (s)
In the start	0,000073 s
In the center	0,000084 s
In the and	0,000001 s
Read and Insert data in the Struture	0,8119587 s
Time run application	0,9097 s

Заключение

Исследование производительности алгоритмов BST (Двоичное Дерево Поиска), Hash Table (Хеш-таблица) и Trie (Префиксное Дерево) для поиска строк в словаре фокусируется на том, как каждая структура данных справляется с вставкой, поиском и удалением элементов (строк), а также учитывает такие факторы, как временная и пространственная сложность.

Двоичное Дерево Поиска (BST)

Сложность:

- Вставка: O(log n) в среднем, O(n) в худшем случае.
- Поиск: O(log n) в среднем, O(n) в худшем случае.
- Удаление: O(log n) в среднем, O(n) в худшем случае.

Характеристики:

- Производительность зависит от сбалансированности дерева. Несбалансированное дерево (например, связанный список) имеет ухудшенную производительность.
- Самобалансирующиеся деревья, такие как AVL и Красно-Черные деревья, поддерживают высоту дерева на уровне O(log n), обеспечивая лучшую производительность.

Преимущества:

- Относительно простая структура, легко реализуемая.
- Хорошая средняя производительность для операций со словарем.

Недостатки:

- Может стать неэффективным, если не сбалансировано.
- Дополнительная сложность для поддержания сбалансированности дерева (в сбалансированных вариантах).

Хеш-таблица (Hash Table)

Сложность:

- Вставка: O(1) в среднем, O(n) в худшем случае (из-за коллизий).
- Поиск: O(1) в среднем, O(n) в худшем случае (из-за коллизий).
- Удаление: O(1) в среднем, O(n) в худшем случае (из-за коллизий).

Характеристики:

- Использует хеш-функцию для отображения ключей в индексы в массиве.
- Справляется с коллизиями с помощью таких техник, как цепочки или открытая адресация.

Преимущества:

- Отличная средняя производительность для вставки, поиска и удаления.
- Очень эффективна для больших объемов данных, если хеш-функция хорошо спроектирована.

Недостатки:

- Производительность может ухудшиться при большом количестве коллизий.
- Требуется хорошо спроектированная хеш-функция.
- Использование пространства может быть менее эффективным из-за хранения списков коллизий или пустых ячеек.

Префиксное Дерево (Trie)

Сложность:

- Вставка: O(L), где L - длина строки.
- Поиск: O(L), где L - длина строки.
- Удаление: O(L), где L - длина строки.

Характеристики:

- Каждый узел представляет собой символ строки.
- Структура особенно эффективна для работы с строками и префиксами.

Преимущества:

- Очень эффективна для операций с префиксами и поиска строк.
- Может компактно хранить строки, особенно с использованием таких методов сжатия, как сжатые префиксные деревья (radix trees).

Недостатки:

- Потребляет больше памяти, особенно если хранятся длинные и разнообразные строки.
- Сложнее реализовать и управлять по сравнению с BST и Hash Table.

Заключение

- **BST**: Лучше всего использовать, когда требуется хороший баланс между простотой и эффективностью при поиске и вставке, особенно если дерево сбалансировано.
- **Hash Table**: Идеальна для ситуаций, когда требуется быстрая и равномерная средняя производительность поиска и вставки, и когда дополнительное пространство для управления коллизиями не является проблемой.
- **Trie**: Отлично подходит для операций с префиксами строк и для быстрого и эффективного поиска в словарях строк, хотя требует больше памяти.

Каждая структура имеет свои преимущества и недостатки в зависимости от конкретного случая использования и требований к производительности и пространству.

Приложение

Чтобы получить доступ к приложению в комплекте с тестами, перейдите по ссылке на моем github

https://github.com/2000zua/work_2024.

Библиография

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms* (4th ed.). MIT Press.
- Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms* (4th ed.). MIT Press.
- Weiss, M. A. (2013). *Data Structures and Algorithm Analysis in C++* (4th ed.). Pearson.
- Sedgewick, R., & Wayne, K. (2016). *Algorithms, Part II* [Online Course]. Coursera.
- Karumanchi, N. (2016). *Data Structures and Algorithms Made Easy* (6th ed.). CareerMonk Publications.
- Weiner, P. (1973). Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM, 16*(6), 333-340.