Análise Comparativa de Estruturas de Dados para Contagem de Frequência de Palavras

Antonio Willian Silva Oliveira[1]

[1]Universidade Federal Do Ceará (UFC) – Campus Quixadá
Av. José de Freitas Queiroz, 5003 – Cedro, Quixadá – CE – CEP 63902-580

williansilva@alu.ufc.br

Abstract. *This paper presents a comparative analysis of four advanced data structures—AVL Tree, Red-Black Tree, Chained Hash Table, and Open Addressing Hash Table—applied to the problem of word frequency counting. A C++ application was developed to process text files and measure key performance metrics, including build time, key comparisons, rotations (for trees), and collisions (for hash tables). The results, obtained through automated benchmarking in a CI/CD environment, demonstrate that hash-based structures offer significantly superior performance in terms of speed and number of comparisons for this specific task, while balanced trees provide guaranteed logarithmic worst-case complexity.*

Resumo. *Este artigo apresenta uma análise comparativa de quatro estruturas de dados avançadas — Árvore AVL, Árvore Rubro-Negra, Tabela Hash com Encadeamento e Tabela Hash com Endereçamento Aberto — aplicadas ao problema de contagem de frequência de palavras. Uma aplicação em C++ foi desenvolvida para processar arquivos de texto e medir métricas de desempenho chave, incluindo tempo de construção, comparações de chaves, rotações (para árvores) e colisões (para tabelas hash). Os resultados, obtidos através de benchmarking automatizado em um ambiente de CI/CD, demonstram que as estruturas baseadas em hash oferecem desempenho significativamente superior em velocidade e número de comparações para esta tarefa, enquanto as árvores balanceadas garantem uma complexidade de pior caso logarítmica.*

1. Introdução

A contagem de frequência de palavras é um problema clássico e fundamental na ciência da computação, com aplicações que vão desde a análise de texto e processamento de linguagem natural (PLN) até a otimização de motores de busca e a análise de dados em larga escala. A eficiência com que essa tarefa é executada depende diretamente da estrutura de dados subjacente, responsável por armazenar e atualizar a contagem de cada palavra única. A escolha da estrutura correta impacta não apenas o tempo de execução, mas também o consumo de memória e a complexidade da implementação.

Este trabalho tem como objetivo principal realizar uma análise comparativa de desempenho entre quatro estruturas de dados avançadas, implementadas como dicionários genéricos para resolver o problema da contagem de frequência. As estruturas analisadas são:

- Árvore AVL (Adelson-Velsky e Landis)

- Árvore Rubro-Negra (Red-Black Tree)
- Tabela Hash com Encadeamento Separado
- Tabela Hash com Endereçamento Aberto (Sondagem Quadrática)

Para isso, foi desenvolvida uma aplicação em C++20 que processa arquivos de texto e utiliza cada uma dessas estruturas para registrar a frequência das palavras. O desempenho foi avaliado com base em métricas empíricas, como tempo de construção, número de comparações de chaves, rotações (para árvores) e colisões (para tabelas hash), permitindo uma análise prática dos trade-offs de cada abordagem.

O restante deste documento está organizado da seguinte forma: a Seção 2 detalha a arquitetura do software e a implementação de cada estrutura de dados. A Seção 3 apresenta a metodologia de testes, o ambiente utilizado e a análise dos resultados obtidos no benchmark. Por fim, a Seção 4 sumariza as conclusões do trabalho e as principais dificuldades encontradas.

2. Implementação das Estruturas

O projeto foi desenvolvido em C++20, seguindo os princípios de Programação Orientada a Objetos (POO) para garantir modularidade e reusabilidade. A implementação foi estruturada em duas frentes: a criação de um conjunto de classes de dicionário genéricas e uma aplicação para a contagem de frequência que as consome.

2.1. Arquitetura Geral do Projeto

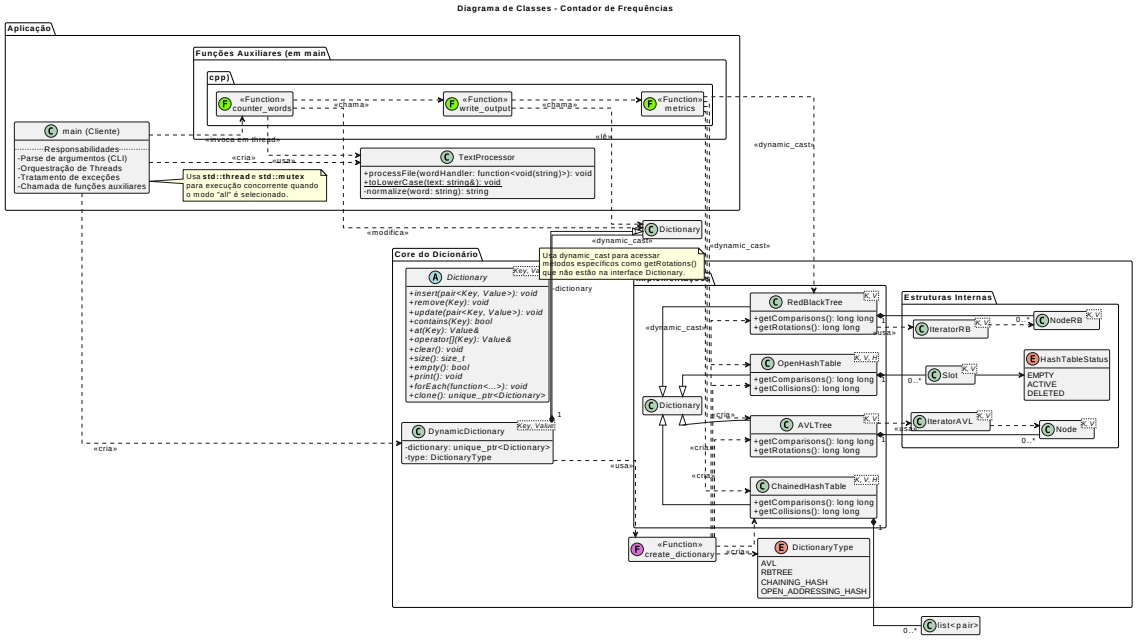
A arquitetura do sistema foi projetada para ser modular e extensível, separando claramente as responsabilidades. A Figura 1 apresenta um diagrama de classes UML que ilustra os principais componentes e suas interações.

O sistema é dividido em dois pacotes principais:

- **Aplicação (Cliente):** Contém a lógica de alto nível, incluindo o ponto de entrada (‘main‘), o processador de texto (‘TextProcessor‘) e as funções auxiliares para orquestrar o benchmark.
- **Core do Dicionário:** Contém o núcleo do projeto, incluindo a interface abstrata ‘Dictionary‘, suas quatro implementações concretas, as estruturas internas (como ‘Node‘ e ‘Slot‘) e os padrões de projeto (‘Factory‘ e ‘Facade‘) que facilitam o uso.

Esta separação garante que o núcleo do dicionário seja totalmente independente da aplicação de contagem de frequência, promovendo o reuso.

Figura 1. Diagrama UML geral da arquitetura do projeto.



2.2. Interface Principal (Dictionary)

O pilar da arquitetura é a classe base abstrata `Dictionary<Key, Value>` que funciona como uma interface para todas as estruturas de dados implementadas. O design desta interface foi inspirado nos contêineres da Biblioteca Padrão do C++, como `std::unordered_map`[cplusplus.com 2023] e `std::map`[cplusplus.com 2025], garantindo um conjunto de operações familiar e robusto. Ao definir um contrato comum, o uso de polimorfismo é facilitado, permitindo que a aplicação principal alterne entre as diferentes estruturas de forma transparente para a análise comparativa.

As principais funções que compõem este contrato são:

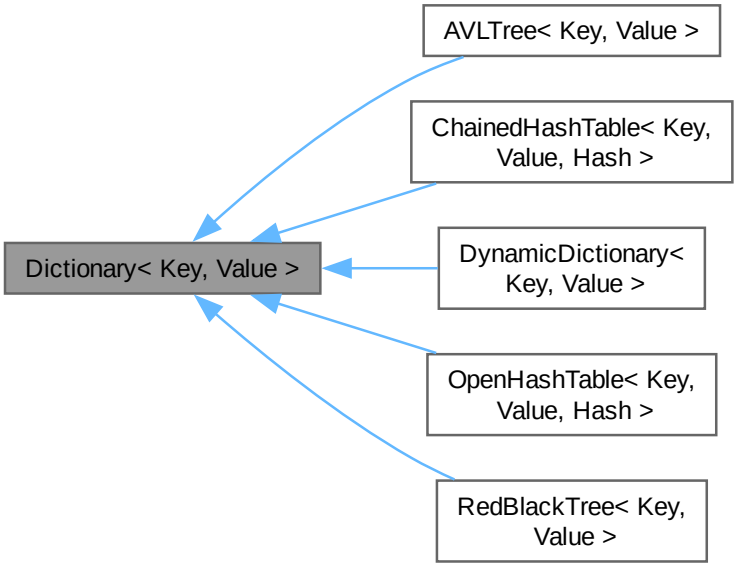
- `void insert(const std::pair<Key, Value>& key_value)`: Insere um par chave-valor no dicionário.
- `void remove(const Key& key)`: Remove um elemento do dicionário pela chave.
- `void update(const std::pair<Key, Value>& key_value)`: Atualiza o valor associado a uma chave existente no dicionário.
- `bool contains(const Key& key)`: Verifica se uma chave está presente no dicionário.
- `Value& at(const Key& key)`: Obtém o valor associado a uma chave, lançando uma exceção se a chave não for encontrada.
- `Value& operator[](const Key& key)`: Sobrecarga do operador de indexação para acessar o valor associado a uma chave. Se a chave não existir, uma nova entrada é criada.
- `void clear()`: Limpa todos os elementos do dicionário.
- `size_t size() const noexcept`: Obtém o número de elementos no dicionário.
- `bool empty() const noexcept`: Verifica se o dicionário está vazio.

- `void print() const`: Imprime o conteúdo do dicionário.
- `void forEach(const std::function<void(const std::pair<Key, Value>&)>& func) const`: Itera sobre todos os pares chave-valor no dicionário, aplicando uma função de callback a cada um. Este método oferece um mecanismo de iteração seguro e polimórfico, contornando a complexidade de implementar iteradores virtuais na interface base.
- `std::unique_ptr<Dictionary<Key, Value» clone() const`: Cria uma cópia profunda (deep copy) do dicionário atual, retornando um ponteiro inteligente para a nova instância.

Todas as quatro estruturas de dados desenvolvidas neste projeto herdam da interface Dictionary<Key, Value>, implementando o contrato definido. Essa abordagem polimórfica é o que permite à aplicação principal tratar todas as estruturas de maneira uniforme, facilitando a troca entre elas para a análise de desempenho.

A Figura 2 ilustra essa relação de herança, mostrando como cada classe concreta se estende da interface base.

Figura 2. Diagrama de herança da classe Dictionary.



Conforme exigido pelo projeto, todas as 4 estruturas contém contadores internos (comparisons e rotations/colisions) são incrementados durante essas operações para permitir a análise de desempenho posterior. O número de comparações é atualizado durante as buscas, e o de rotações, nas funções de balanceamento das árvores. (Mais detalhes na seção 3)

A seguir, uma breve descrição de cada implementação é fornecida.

2.3. Árvore AVL (Adelson-Velsky e Landis)

A classe `AVLTree` implementa um dicionário utilizando uma Árvore AVL, uma das estruturas de busca binária auto-balanceáveis exigidas pelo projeto. Sua principal característica é a manutenção rigorosa do balanceamento: a diferença entre as alturas das subárvores esquerda e direita de qualquer nó é mantida em, no máximo, -1 ou 1[Szwarcfiter and Markenzon 2015]

Embora essa rigidez resulte em árvores mais otimizadas em altura, ela pode exigir um número maior de rotações durante as operações de inserção e remoção em comparação com outras abordagens, como a Árvore Rubro-Negra. O principal benefício desta estrutura é a garantia de que as operações de busca, inserção e remoção tenham uma complexidade de tempo de pior caso de $O(\log n)$, ou seja, de ordem logarítmica.

Para implementar essa lógica, a classe `TreeAVL` gerencia um ponteiro para o nó raiz (`root`) e utiliza uma estrutura auxiliar `Node` para representar cada elemento. Cada `Node` armazena um par chave-valor, sua altura na árvore e ponteiros para os filhos direito e esquerdo.

A seguir na figura 3 é possível ver a estrutura visualmente.

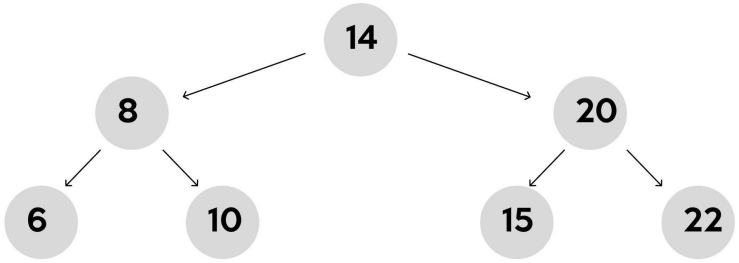


Figura 3. Demonstração simples de uma Árvore AVL

As operações de inserção e remoção são implementadas de forma recursiva. O processo se inicia na raiz e desce pela árvore comparando as chaves para encontrar a posição correta para a operação. O aspecto crucial da Árvore AVL entra em ação no retorno da recursão: o rebalanceamento.

Após cada modificação na estrutura, uma função privada (`fixup_node`) é chamada para cada nó no caminho de volta à raiz. Esta função executa duas tarefas principais:

1. Atualiza a Altura: A altura do nó é recalculada com base nas alturas de suas subárvores.
2. Verifica o Fator de Balanço: O fator de balanço (diferença de altura entre as subárvores direita e esquerda) é verificado. Se um desbalanceamento é detectado (fator de balanço -2 ou 2), as rotações são acionadas.

Para corrigir o desbalanceamento, as funções `rightRotation` e `leftRotation` são aplicadas. Dependendo da configuração dos nós, pode ser necessária uma rotação simples ou uma rotação dupla (uma combinação de duas rotações simples) para restaurar a propriedade da AVL.

2.4. Árvore Rubro-Negra (Red-Black Tree)

A classe `RedBlackTree` implementa a segunda abordagem de dicionário baseada em árvore balanceada. Diferente da AVL, que controla a altura, a Árvore Rubro-Negra garante o balanceamento através de um conjunto de cinco propriedades, aplicadas por meio da coloração de cada nó como "vermelho"ou "preto"[Cormen et al. 2022].

Essas regras são consideradas menos estritas que as da AVL, o que significa que a árvore não é mantida em um estado de balanceamento tão perfeito. A vantagem prática é que, em média, menos rotações são necessárias durante as operações de inserção e remoção, o que pode levar a um desempenho mais rápido para operações de escrita. Assim como a AVL, a complexidade de pior caso para as operações permanece $O(\log n)$.

Na implementação, como na figura 4, um nó sentinela (`nil`) é utilizado para representar todas as folhas e o pai da raiz. Essa técnica simplifica o código, eliminando a necessidade de verificar ponteiros nulos em muitos casos. A inserção de um novo nó é sempre feita com a cor vermelha. Após a inserção, uma função de correção (`fixup_node`) é chamada para percorrer o caminho de volta à raiz, realizando rotações e recolorindo os nós para restaurar as propriedades da árvore que possam ter sido violadas.

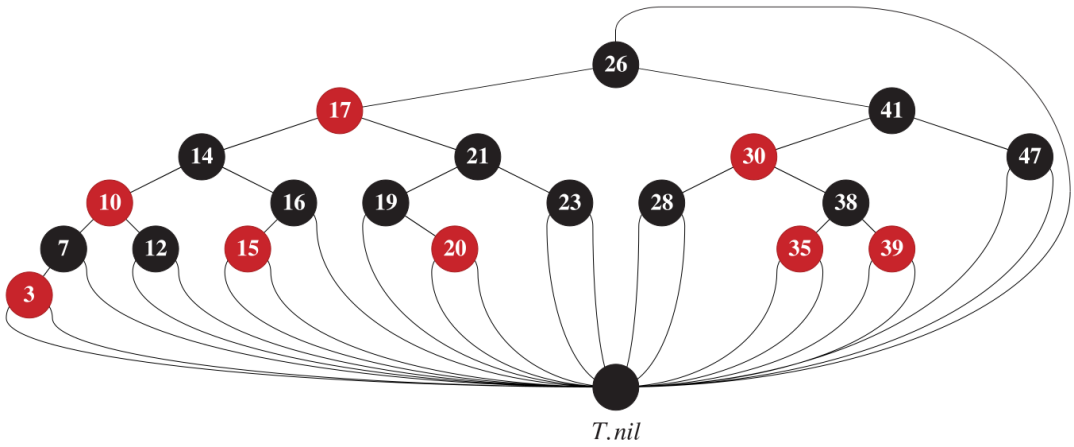


Figura 4. Demonstração de uma Árvore Rubro-Negra com o nó `nil`

2.5. Tabela Hash com Encadeamento Separado

A classe `ChainedHashTable` implementa um dicionário com uma Tabela de Dispersão (Hash Table) com encadeamento separado. Esta é uma das estratégias clássicas para o tratamento de colisões, que ocorrem quando a função de hash mapeia duas chaves distintas para o mesmo índice da tabela. [Szwarcfiter and Markenzon 2015]

O princípio de funcionamento é simples: em vez de armazenar um único elemento em cada posição (ou "bucket") da tabela, cada posição contém uma estrutura de dados secundária, é usada uma lista encadeada (`std::list`), que armazena todos os pares chave-valor que colidiram naquele índice, como na figura 5. Quando uma operação de inserção, busca ou remoção é realizada, a função de hash primeiro calcula o índice. Em seguida, a operação é efetuada na lista correspondente àquele índice, o que pode exigir uma busca sequencial curta dentro da lista.

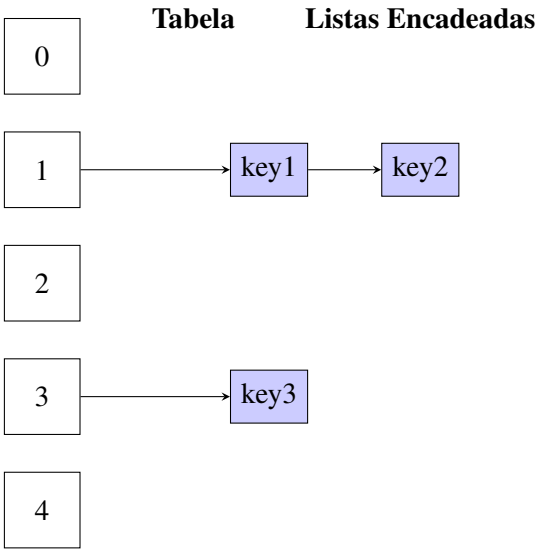


Figura 5. Diagrama de uma Tabela Hash com Encadeamento.

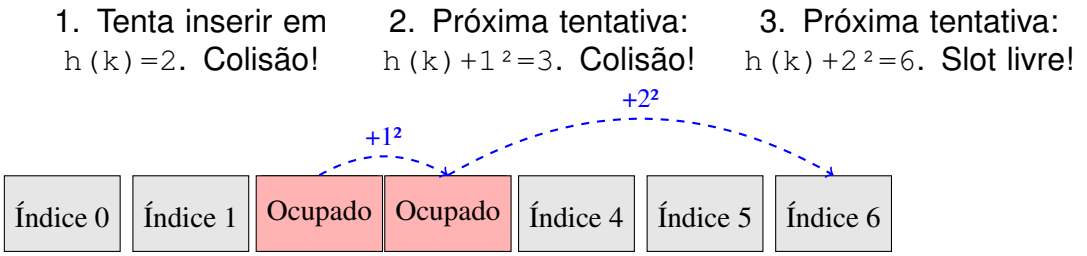
Uma característica crucial para o bom desempenho desta estrutura é o **fator de carga** (α), definido como a razão entre o número de elementos (N) e o tamanho da tabela (M), ficando $\frac{N}{M}$. Para evitar que as listas se tornem muito longas e o desempenho degrade para $O(n)$, a tabela é redimensionada automaticamente através de uma operação de **rehash**. Uma nova tabela maior (geralmente com o dobro do tamanho) é alocada, e todos os elementos existentes são re-hashados para suas novas posições.

2.6. Tabela Hash com Endereçamento Aberto (Sondagem Quadrática)

E por último, a classe `OpenHashTable` implementa um dicionário usando uma Tabela de Dispersão com Endereçamento Aberto. Diferente do encadeamento, onde múltiplos elementos podem ocupar o mesmo slot da tabela em uma lista secundária, no endereçamento aberto, cada slot armazena, no máximo, um elemento.

Quando ocorre uma colisão, a tabela procura por um slot alternativo dentro da própria tabela, seguindo uma sequência de sondagem pré-definida. Nesta implementação, foi utilizada a **Sondagem Quadrática** para solucionar as colisões. Se a posição inicial $h(k)$ já estiver ocupada, a sequência de posições verificadas será $h(k) + 1^2$, $h(k) + 2^2$, $h(k) + 3^2$, e assim por diante, até que um slot vazio seja encontrado, como na figura 6. Essa abordagem ajuda a mitigar o problema de "agrupamento primário"que pode ocorrer na sondagem linear, mas agora pode ocorrer o problema de "agrupamento secundário".[Szwarcfiter and Markenzon 2015]

Figura 6. Ilustração de uma colisão resolvida com Sondagem Quadrática.



Para gerenciar as operações de remoção de forma eficiente, cada slot na tabela possui três estados, gerenciados pela struct auxiliar `Slot`:

Tabela 1. Estados da struct `Slot`

Estados	Descrição
EMPTY	O slot está vazio e nunca foi usado.
ACTIVE	O slot contém um par chave-valor ativo.
DELETED	O slot continha um par chave-valor que foi removido (lápide).

Assim como na implementação com encadeamento, a `OpenHashTable` também monitora o fator de carga e realiza a operação de rehash para uma tabela maior quando o limiar é atingido, garantindo que o desempenho se mantenha eficiente.

2.7. Implemtações adicionais

Para aumentar a flexibilidade e desacoplar o código cliente das implementações concretas dos dicionários, foram utilizados dois padrões de projeto: Factory Method e Facade.

2.7.1. Fábrica de Dicionário (`DictionaryFactory`)

Para centralizar a lógica de criação dos dicionários e permitir que o código cliente solicite uma instância sem conhecer os detalhes de sua classe concreta, foi implementada uma `DictionaryFactory`.

Essa fábrica segue o padrão de projeto *Factory Method*, o qual "sugere que você substitua chamadas diretas de construção de objetos (usando o operador `new`) por chamadas para um método fábrica especial"[Shvets 2020]. Embora os objetos ainda sejam criados via `new`, essa operação é realizada internamente no método fábrica, promovendo o encapsulamento da lógica de instanciação e o desacoplamento entre o cliente e as classes concretas.

A `DictionaryFactory` fornece uma função que retorna um `std::unique_ptr<Dictionary<...>` com base em um enumerador (`DictionaryType`) que especifica a estrutura desejada (como AVL ou Rubro-Negra, por exemplo).

Essa abordagem simplifica drasticamente a experimentação e os testes, pois a troca de uma estrutura de dados por outra se resume à alteração de um único parâmetro na chamada à fábrica, como demonstrado na Listagem 1.

```
#include "dictionary/DictionaryFactory.hpp"
#include <iostream>
#include <string>
#include <memory> // Para std::unique_ptr

int main() {
    // Usa a factory para criar uma AVL Tree.
    // O tipo é Dictionary<string, int> para contar palavras.
    std::unique_ptr<Dictionary<std::string, int>> dict(
   create_dictionary<string, int>(DictionaryType::AVL));

```

```
    // Insere ou atualiza a frequência de palavras
    dict->insert("sbc", 1);
    dict->insert("template", 5);
    dict->insert("sbc", 2); // Atualiza o valor para 2

    // Busca a frequência de uma palavra
    try {
        int frequency = dict->at("template");
        std::cout << "Frequencia de 'template': "
                  << frequency << std::endl;
    } catch (const std::out_of_range& e) {
        std::cout << e.what() << std::endl;
    }

    return 0;
}
```

Listing 1. Exemplo de uso da DictionaryFactory para criar e manipular um dicionário.

Além disso, evita acoplamentos rígidos entre o código cliente e as classes concretas, concentrando a responsabilidade de criação em um único ponto do sistema (em conformidade com o princípio da responsabilidade única).[Shvets 2020] Outro benefício importante é a aderência ao princípio aberto/fechado, permitindo a introdução de novas implementações de dicionário sem modificar o código cliente existente, favorecendo a extensibilidade e a manutenção da aplicação.[Shvets 2020]

2.7.2. Dicionário dinâmico (`DynamicDictionary`)

Para simplificar ainda mais a utilização no código cliente, a classe `DynamicDictionary` foi implementada seguindo o padrão *Facade* (ou *Wrapper*). Esse padrão define "uma classe que fornece uma interface simples para um subsistema complexo que contém muitas partes que se movem. [...] Ela inclui apenas aquelas funcionalidades que o cliente se importa" [Shvets 2020]. Nesse contexto, a `DynamicDictionary` atua como uma fachada unificada, encapsulando a complexidade de escolher e instanciar uma estrutura de dados específica.

Internamente, a `DynamicDictionary` utiliza a `DictionaryFactory` para criar o dicionário concreto no momento de sua construção. Todas as chamadas de métodos (como `insert`, `at`, etc.) feitas a uma instância de `DynamicDictionary` são simplesmente delegadas para o objeto de dicionário que ela envolve.

O resultado é um código cliente mais limpo e com um acoplamento ainda menor, pois ele passa a interagir apenas com uma única classe, `DynamicDictionary`, configurando a estrutura de dados subjacente apenas uma vez, durante a inicialização.

2.8. Funcionamento Geral do Programa

O fluxo de execução da aplicação é o seguinte:

1. **Leitura de Argumentos:** O programa recebe via linha de comando o tipo de estrutura de dados a ser usada (ex: ‘avl‘, ‘rbt‘, ‘all‘) e o nome do arquivo de texto.

```
./Dictionary all bible.txt
```

Listing 2. Exemplo de execução via linha de comando

2. **Criação da Estrutura:** Uma fábrica de dicionários (‘DictionaryFactory‘) instancia a estrutura de dados escolhida. A fábrica será usada para operar os dois modos:

   (a) **Modo específico**: Se um tipo de estrutura de dados (ex: "avl", "rbt") é fornecido, a contagem de palavras é realizada usando apenas essa estrutura.

   (b) **Modo "all"**: Se o argumento for "all", a função cria quatro threads, cada uma executando a contagem de palavras em paralelo com uma estrutura de dados diferente (AVL, Red-Black Tree, Chaining Hash, Open Addressing Hash) para fins de comparação de desempenho.

3. **Processamento do Texto:** A classe `TextProcessor` é responsável por abrir o arquivo, ler palavra por palavra, normalizá-las (converter para minúsculas, mantendo hífens internos) e passar cada palavra válida para a estrutura de dados. O processo de normalização, implementado no método privado `normalize()`, segue duas etapas:

   (a) A palavra é convertida para minúsculas.

   (b) Uma expressão regular (‘regex‘) é aplicada para extrair o conteúdo válido. A regex foi projetada para aceitar caracteres alfabéticos (incluindo acentuados como ‘à‘ ou ‘ÿ‘) e permitir hífens e apóstrofos no meio das palavras (ex: "sub-hepático"), descartando qualquer outro tipo de pontuação ou símbolo.

   O método público principal, `processFile()`, utiliza um padrão de *callback* para se manter desacoplado da lógica de contagem. Ele aceita uma função como argumento (‘wordHandler‘), que é chamada para cada palavra normalizada e válida encontrada no texto. Esse design permite que o `TextProcessor` seja reutilizado em outros contextos, bastando fornecer uma função diferente para processar as palavras.

4. **Contagem de Frequência:** Para cada palavra recebida, o programa utiliza o operador ‘[]‘ do dicionário, que incrementa a contagem se a palavra já existe ou a insere com contagem 1 caso contrário.

5. **Geração de Saída:** Ao final do processamento, o programa grava em um arquivo de saída a lista de palavras em ordem alfabética com suas respectivas frequências, além das métricas de desempenho coletadas.

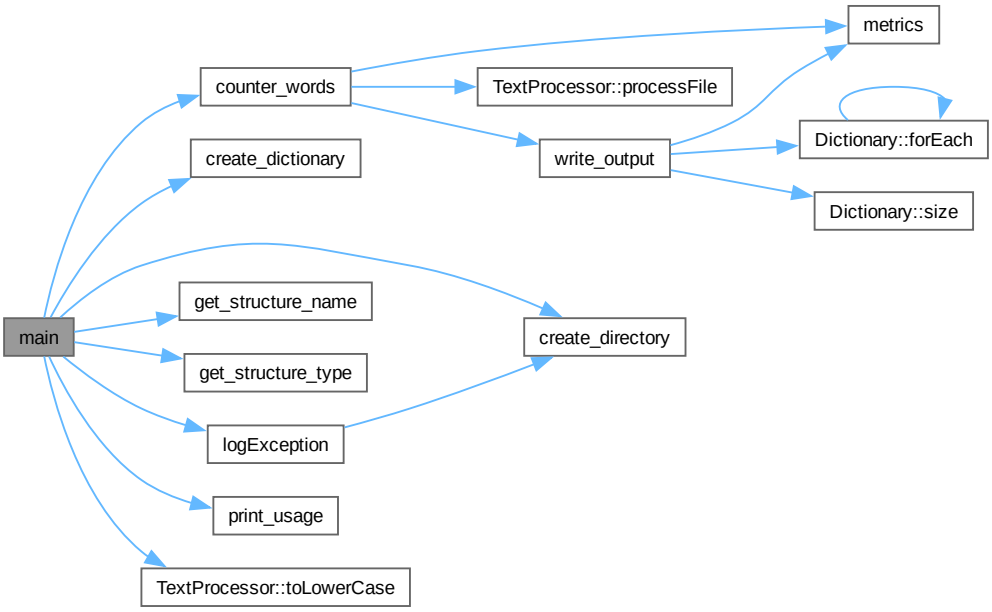


Figura 7. Grafo de chamadas da função `main`

2.9. Formato de Entrada e Saída

Entrada: Um arquivo de texto (‘.txt‘) contendo o corpus a ser analisado. O arquivo deve ser colocado no diretório ‘files/‘.

Saída: Um arquivo de texto gerado no diretório ‘out/‘. O arquivo contém duas seções principais:

1. **Lista de Frequências:** Uma lista de todas as palavras únicas encontradas, em ordem alfabética, seguidas por sua frequência. Ex: ‘[palavra, 2]‘.
2. **Métricas de Desempenho:** Um resumo contendo o nome da estrutura, o tempo total para montar a tabela, o número de elementos únicos e as métricas específicas (comparações, rotações/colisões).

```
================================================================================

Word Count for file: file.txt
================================================================================


[a, 1]
[igual, 1]
[mais, 1]
[oito, 1]
[quatro, 2]
[são, 1]

```

```
12 ================================================================================

13 Metrics:
14 Structure: RB TREE
15 Build time: 0.292956 ms
16 Size: 6
17 Rotations: 1
18 Comparisons: 65
19 ================================================================================
```

Listing 3. Exemplo de Saída para o texto "Quatro mais quatro são igual a oito."

3. Testes e Resultados

3.1. Testes Unitários

Para garantir a corretude e a robustez das estruturas de dados implementadas, uma suíte de testes unitários abrangente foi desenvolvida utilizando o framework Google Test. A estratégia de teste foi dividida em três categorias principais para validar diferentes aspectos das estruturas de dados.

3.1.1. Testes de Interface (Typed Tests)

A abordagem principal para validar a funcionalidade comum a todas as estruturas foi o uso de **Testes Tipados** do Google Test. Foi criada uma *test fixture* genérica, `DictionaryTest<T>`, que define um conjunto de testes para a interface `Dictionary`. Casos de teste como `InsertAndSize`, `Remove`, `Contains`, `At`, `Clear` e `Clone` foram escritos uma única vez. Em seguida, esta suíte foi instanciada para cada uma das quatro implementações (`AVLTree`, `RedBlackTree`, `ChainedHashTable`, `OpenHashTable`), garantindo que todas elas cumprissem o contrato da interface de forma consistente e correta.

3.1.2. Testes de Lógica Específica

Além dos testes de interface, foram criadas suítes específicas para validar a lógica interna e os casos de borda de certas estruturas:

- **AVLTreeSpecificTest:** Esta suíte foca exclusivamente em verificar se os algoritmos de rebalanceamento da Árvore AVL estão corretos. Testes foram projetados para forçar cada um dos quatro cenários de rotação (simples à direita, simples à esquerda, dupla direita-esquerda e dupla esquerda-direita), validando a integridade da árvore após as operações.
- **HashTableStressTest:** Uma suíte tipada dedicada a estressar as duas implementações de Tabela Hash. Os testes focam em cenários de alta colisão e na verificação do mecanismo de *rehash* quando o fator de carga excede o limite, garantindo que a tabela se redimensione e mantenha a consistência dos dados.

3.1.3. Testes de Estresse

Finalmente, uma suíte de testes de estresse, `GeneralStressTest`, foi aplicada a todas as quatro estruturas. Este teste insere um grande volume de dados (5000 elementos) em ordem aleatória, verifica a presença de todos, remove metade deles (também em ordem aleatória) e, por fim, valida a consistência final da estrutura. O objetivo é garantir a estabilidade, o gerenciamento correto de memória e a ausência de *memory leaks* ou falhas de segmentação sob carga intensa.

3.2. Benchmark de Desempenho

Para avaliar e comparar o desempenho das estruturas de dados, foram executados testes utilizando o arquivo 'bible.txt' como entrada. O benchmark foi automatizado com um script ('benchmark.sh') que executa o programa 1000 vezes para cada estrutura e calcula a média das métricas.

3.2.1. Ambiente de Teste

Os testes foram executados em um ambiente de integração contínua do GitHub Actions, garantindo um ambiente de teste consistente e reprodutível. As especificações do runner foram:

- **Sistema Operacional:** Ubuntu 24.04.2 LTS
- **Imagem do Runner:** ubuntu-24.04 (Versão: 20250622.1.0)
- **CPU:** 4 vCPUs [GitHub Docs 2025]
- **RAM:** 16 GB [GitHub Docs 2025]
- **SSD:** 16 GB [GitHub Docs 2025]
- **Arquitetura:** x64 [GitHub Docs 2025]
- **Compilador:** g++ (GCC) com as flags '-std=c++20 -Wall -Wextra -O3 -DNDEBUG -MMD -MP -Iinclude'

3.2.2. Métricas de Desempenho Coletadas

A análise teórica de complexidade (e.g., $O(\log n)$ ou $O(1)$) é essencial, mas a performance no mundo real também é influenciada por constantes e custos de operações específicas. Para capturar essas nuances, cada estrutura de dados foi instrumentada para contar operações fundamentais.

3.2.3. Árvores Balanceadas: AVL e Rubro-Negra

Para as árvores de busca balanceadas, as operações de inserção, busca e remoção dependem da navegação na árvore e da manutenção de seu balanceamento. As métricas coletadas são:

- **Comparações**: Representa o número de vezes que a chave de um nó é comparada com a chave sendo buscada, inserida ou removida. Esta é a métrica mais fundamental para avaliar o custo de uma busca, pois está diretamente relacionada à

profundidade dos nós acessados. Em uma árvore balanceada, o número esperado de comparações é proporcional a $O(\log n)$.

- **Rotações**: Mede o número de operações de rotação (simples ou duplas) realizadas para manter o fator de balanceamento (no caso da AVL) ou as propriedades rubro-negras. As rotações são o principal custo adicional das operações de inserção e remoção em árvores balanceadas. Comparar o número de rotações entre a AVL e a Rubro-Negra oferece um insight prático sobre o quão "rígido"é o critério de balanceamento de cada uma e o custo associado a ele.

3.2.4. Tabelas Hash

Para as tabelas hash, o desempenho ideal é alcançado quando as chaves são distribuídas uniformemente, minimizando colisões. As métricas focam em medir o quão perto a implementação chega desse ideal:

- **Colisões**: Ocorre quando a função de hash mapeia duas chaves distintas para o mesmo índice na tabela. Este é o evento primário que degrada o desempenho de uma tabela hash, afastando-o da complexidade $O(1)$. Contar o número de colisões é essencial para avaliar a qualidade da função de hash e a eficácia da estratégia de tratamento de colisão (encadeamento ou endereçamento aberto).
- **Comparações**: Em tabelas hash, as comparações ocorrem somente após uma colisão.
  - No **encadeamento separado**, mede o número de chaves percorridas na lista ligada de um determinado índice.
  - No **endereçamento aberto**, mede o número de "saltos"(sondagens) necessários para encontrar a chave ou um slot vazio.

  Em ambos os casos, esta métrica quantifica o custo real do tratamento de colisões.

3.3. Resultados Obtidos

Os resultados médios das 1000 execuções com o arquivo ‘bible.txt‘ são apresentados na Tabela 2.

Tabela 2. Resultados comparativos (média de 1000 execuções).

Estrutura	Tempo (ms)	Rotações/Colisões	Comparações
AVL Tree	450.28	9.521	15.603.115
Red-Black Tree	434.99	8.032	15.105.025
Chained Hash Table	388.91	10.930	1.043.630
Open Addressing Hash	380.41	11.768	979.059

3.3.1. Análise de Desempenho

Os dados da Tabela 2 permitem extrair várias conclusões sobre o desempenho prático de cada estrutura para esta aplicação:

- **Tempo de Execução:** As Tabelas Hash foram significamente mais rápidas que as árvores balanceadas, com a a Tabela de Endereçamento Aberto apresentando o melhor tempo médio (380.41ms), como é possível ver na figura 3.3.1

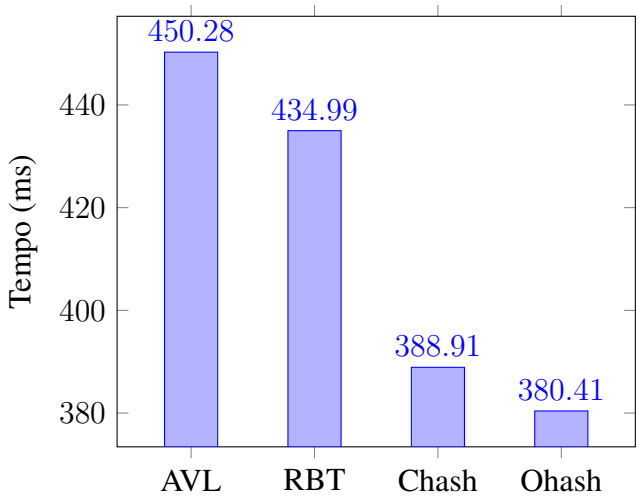


Figura 8. Tempo médio por estrutura

Isso confirma a vantagem teórica da complexidade média de $O(1)$ para operações de hash em cenários práticos. A Árvore AVL, com seu balanceamento mais rígido, foi a mais lenta (450.28ms).

- **Comparações:** A diferença no número de comparações é a mais expressiva. As árvores AVL e Rubro-Negra realizaram mais de 15 milhões de comparações, refletindo sua natureza de busca logarítmica ($O(\log n)$) a cada inserção, como visto na figura 3.3.1. Em contraste, as Tabelas Hash realizaram pouca mais de 1 milhão de comparações, demonstrando como a função de Hash direciona a maioria das inserções diretamente para o local correto, com comparações ocorrendo apenas durante o tratamento de colisões.

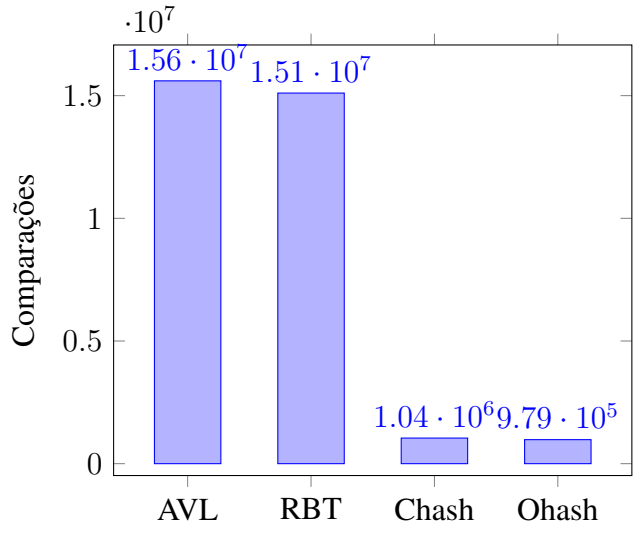


Figura 9. Comparações médias por estrutura

- **Rotações (Árvores):** A Árvore AVL realizou aproximadamente 18.5% mais rotações que a Árvore Rubro-Negra (9.521 vs. 8.032). Este resultado é esperado, pois o critério de balanceamento mais estrito da AVL a força a realizar rebalance-

amentos com mais frequência para manter as alturas das subárvores perfeitamente ajustadas.

- **Colisões (Tabelas Hash):** Ambas as implementações de hash tiveram um número similar de colisões. A Tabela de Endereçamento Aberto teve um número ligeiramente maior de colisões, o que pode ser atribuído à forma como a sondagem quadrática ocupa novos slots, potencialmente aumentando a probabilidade de colisões futuras em comparação com as listas isoladas do encadeamento. No entanto, seu menor número de comparações e tempo de execução sugere que a resolução dessas colisões foi mais eficiente (provavelmente devido a melhor localidade de cache e menor sobrecarga de alocação de nós de lista.)

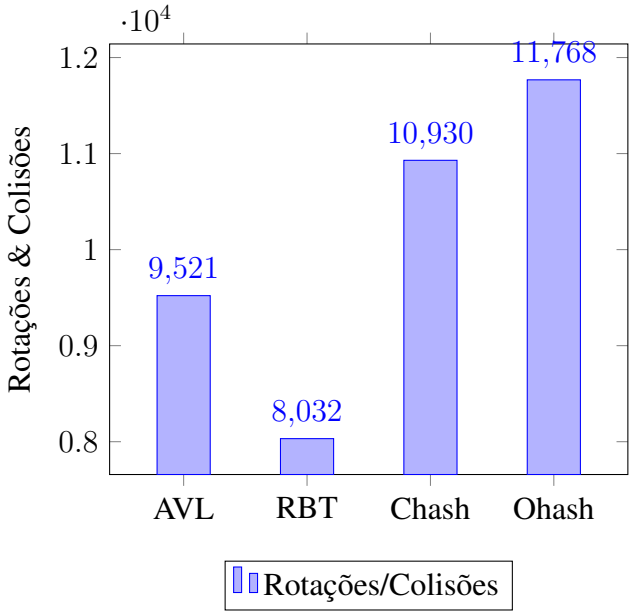


Figura 10. Média de rotações/colisões por estrutura

4. Conclusão

Este trabalho demonstrou com sucesso a implementação e a análise de desempenho de quatro estruturas de dados avançadas aplicadas a um problema prático de contagem de frequência de palavras. A partir dos resultados, foi possível observar que as Tabelas Hash apresentaram desempenho superior em termos de tempo de execução e número de comparações, tornando-as a escolha ideal para esta aplicação específica onde a velocidade de inserção é crítica. As árvores balanceadas, embora mais lentas, garantem uma complexidade de pior caso logarítmica e mantêm os dados ordenados naturalmente, o que pode ser uma vantagem em outros cenários.

As principais dificuldades encontradas durante o desenvolvimento podem ser sumarizadas em alguns pontos chave. A implementação de iteradores para as árvores e a busca por uma solução que permitisse seu uso de forma polimórfica através da interface ‘Dictionary‘ representou um desafio de design significativo. Adicionalmente, o gerenciamento de memória na Árvore Rubro-Negra se mostrou sensível, com vazamentos de memória (memory leaks) sendo identificados e corrigidos durante a fase de testes unitários. Por fim, a criação de uma rotina de normalização de texto robusta, capaz de lidar

com diferentes codificações e casos especiais como o hífen, também foi uma tarefa não trivial.

Referências

[Cormen et al. 2022] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2022). *Introduction to Algorithms - Fourth Edition*. The MIT Press.

[cplusplus.com 2023] cplusplus.com (2023). *std::unordered_map — cplusplus.com*. cplusplus.com. Acesso à referência C++ – contêiner associativo não ordenado.

[cplusplus.com 2025] cplusplus.com (2025). *std::map - cplusplus.com*. cplusplus.com. Acesso à referência C++ – contêiner associativo ordenado.

[GitHub Docs 2025] GitHub Docs (2025). *GitHub-hosted runners*. GitHub. Referência oficial dos runners hospedados pelo GitHub Actions. Acesso em: 15 jul. 2025.

[Shvets 2020] Shvets, A. (2020). *Mergulho nos Padrões de Projeto*, chapter 4. Refactoring.Guru.

[Szwarcfiter and Markenzon 2015] Szwarcfiter, J. L. and Markenzon, L. (2015). *Estruturas de dados e seus algoritmos - 3.ed.* LTC — Livros Técnicos e Científicos Editora Ltda.