Análise Comparativa de Desempenho de Estruturas de Dados para Dicionários

Iago de Oliveira Lô - 565321

[1] Universidade Federal do Ceará - Campus Quixadá (UFC)
Quixadá – CE – Brazil

iagooliveiralo07@alu.ufc.br

Abstract. *This work presents the implementation and comparative performance analysis of four data structures for dictionaries: AVL Tree, Red-Black Tree, Chained Hash Table, and Open Addressing Hash Table with Double Hashing. The structures were applied to the problem of word frequency counting. Key performance metrics such as execution time, key comparisons, and structure-specific operations (rotations, color changes, collisions) were collected and analyzed. The results confirm the superior performance of Hash Tables in average-case scenarios, while balanced trees offer more predictable worst-case guarantees.*

Resumo. *Este trabalho apresenta a implementação e a análise de desempenho comparativa de quatro estruturas de dados para dicionários: Árvore AVL, Árvore Rubro-Negra, Tabela Hash com Encadeamento e Tabela Hash com Endereçamento Aberto por Hash Duplo. As estruturas foram aplicadas ao problema de contagem de frequência de palavras. Métricas de desempenho chave como tempo de execução, comparações de chave e operações específicas de cada estrutura (rotações, trocas de cor, colisões) foram coletadas e analisadas. Os resultados confirmam a superioridade de desempenho das Tabelas Hash em cenários de caso médio, enquanto as árvores auto-balanceadas oferecem um comportamento mais previsível e garantias de pior caso.*

1. Introdução

A organização e o acesso eficiente a grandes volumes de dados são problemas fundamentais em projetos de software. A escolha da estrutura de dados correta é um fator crítico que impacta diretamente o desempenho de um sistema. Este trabalho foca na implementação e na análise de desempenho comparativa de quatro estruturas de dados distintas, aplicadas ao problema de contagem de frequência de palavras em um corpus textual. As estruturas analisadas são: a Árvore AVL, a Árvore Rubro-Negra, a Tabela Hash com Encadeamento Separado e a Tabela Hash com Endereçamento Aberto por Hash Duplo.

Este relatório está estruturado da seguinte forma: a Seção 2 detalha a implementação de cada estrutura de dados, destacando suas características e as decisões de projeto. A Seção 3 descreve a implementação da interface dicionário, responsável por coordenar a aplicação, a seção 4 descreve os experimentos feitos, a metodologia usada para avaliar o desempenho de cada estrutura afim de usar os dados coletados para uma análise mais precisa, o que é feito na seção 5. A Seção 6 apresenta a conclusão do trabalho, sumarizando os achados e discutindo os trade-offs de cada abordagem.

2. Implementação das Estruturas de Dados

Esta seção visa apresentar como cada estrutura foi pensada e construída, mostrando suas individualidades.

2.1. Árvore AVL

A Árvore AVL (Adelson-Velsky e Landis) é uma estrutura de árvore de busca binária (ABB) auto-balanceada. Sua propriedade fundamental é a manutenção de um fator de balanceamento para cada nó, definido como a diferença entre a altura da subárvore direita e a da subárvore esquerda. A propriedade da AVL estipula que este fator deve pertencer ao conjunto $\{-1, 0, 1\}$. Esta restrição garante que a altura da árvore permaneça logarítmica, $O(\log n)$, em relação ao número de nós, assegurando um desempenho eficiente para as operações. O rebalanceamento é mantido através de rotações simples ou duplas, acionadas quando uma inserção ou remoção viola a propriedade de altura.

A estrutura foi implementada como uma classe template em C++, `AVL<Key, Value>`, para garantir generalidade. Cada nó armazena um par chave-valor (`std::pair<Key, Value>`), ponteiros para os filhos esquerdo e direito, e um inteiro para sua altura. Na aplicação final deste projeto, o par será utilizado para armazenar uma palavra (chave) e sua frequência (valor).

As operações de inserção e remoção foram implementadas de forma recursiva. Após a modificação em uma subárvore, as chamadas recursivas retornam, permitindo a atualização da altura de cada nó ancestral. Durante este retorno, o fator de balanceamento é verificado. Caso um desbalanceamento seja detectado (fator -2 ou +2), a operação de rebalanceamento apropriada é executada para restaurar a invariante da AVL.

2.2. Árvore Rubro-Negra

A Árvore Rubro-Negra é uma ABB auto-balanceada que utiliza um sistema de coloração de nós (vermelho ou preto) para garantir seu balanceamento. O equilíbrio é mantido pela aplicação de cinco propriedades fundamentais, que, entre outras coisas, estipulam que a raiz é preta e que nenhum nó vermelho pode ter um filho vermelho. A propriedade mais importante é a da "altura negra": todo caminho de um nó até uma de suas folhas descendentes deve conter o mesmo número de nós pretos. Isso garante a altura logarítmica da árvore, $O(\log n)$.

A implementação da classe `RB<Key, Value>` utiliza um nó sentinela, `TNULL`, que representa todas as folhas (NIL) da árvore e também serve como pai da raiz. Esta abordagem de projeto simplifica o código ao evitar verificações de ponteiros nulos. Cada nó armazena o par chave-valor, sua cor, e ponteiros para os filhos esquerdo, direito e para o pai.

As operações de inserção e remoção são seguidas pela chamada de funções de correção (`insertFix` e `deleteFix`). Estas funções são responsáveis por restaurar as cinco propriedades da árvore, que podem ser violadas durante uma modificação, através de uma série de rotações e trocas de cor que se propagam pela árvore conforme necessário.

2.3. Tabela Hash com Encadeamento Separado

A Tabela Hash é uma estrutura que mapeia chaves a valores através de uma função de hash, permitindo, em média, operações em tempo constante. Para o tratamento de co-

lisões — que ocorrem quando chaves distintas são mapeadas para o mesmo índice —, foi utilizada a estratégia de Encadeamento Separado. Nesta abordagem, cada índice da tabela (ou ”bucket”) aponta para uma lista ligada que armazena todos os elementos cujo hash corresponde àquele índice.

A classe `ChainedHashTable<Key, Value>` foi implementada utilizando um `std::vector` de `std::list<std::pair<Key, Value>>`. O mapeamento de chaves para índices é realizado em duas etapas: primeiro, a função de hash padrão do C++, `std::hash<Key>`, gera um código hash; em seguida, o método da divisão (operador módulo %) é aplicado sobre o tamanho da tabela para determinar o bucket.

Para garantir que o desempenho se mantenha próximo de $O(1)$, a tabela implementa um mecanismo de redimensionamento (`rehash`). Quando o fator de carga (definido como o número de elementos dividido pelo tamanho da tabela) excede um limiar pré-definido, a tabela é reconstruída com um tamanho maior — nesta implementação, o próximo número primo maior que o dobro do tamanho anterior — e todos os elementos são re-inseridos.

2.4. Tabela Hash com Endereçamento Aberto (Hash Duplo)

O Endereçamento Aberto é uma estratégia de tratamento de colisões em que todos os elementos são armazenados no próprio vetor da tabela. Quando uma colisão acontece, o algoritmo sonda (prova) posições subsequentes na tabela de forma sistemática até encontrar um slot disponível.

Para esta implementação, foi escolhida a técnica de Hash Duplo. Esta abordagem de sondagem utiliza uma segunda função de hash para calcular o intervalo (ou ”passo”) entre cada prova, o que mitiga eficientemente os problemas de agrupamento (*clustering*) que afetam métodos mais simples, como a sondagem linear.

Um pilar desta implementação é o gerenciamento do estado de cada slot, que pode ser `EMPTY`, `OCCUPIED` ou `DELETED`. O estado `DELETED` é fundamental para o correto funcionamento da operação de remoção. Em vez de remover fisicamente um elemento (o que poderia quebrar uma cadeia de sondagem existente), o slot é apenas marcado como deletado. Isso garante que as operações de busca por chaves que colidiram e passaram por aquele slot continuem a sondar corretamente, sem parar prematuramente. A remoção, portanto, é uma operação ”preguiçosa” que apenas altera o estado do slot.

3. Arquitetura da Aplicação

Para além da implementação isolada das estruturas de dados, foi desenvolvida uma arquitetura de software coesa que permite a sua utilização e comparação sistemática. Esta seção detalha os componentes de software que compõem a aplicação final, incluindo a interface polimórfica, o módulo de processamento de texto e o formato de interação com o utilizador.

3.1. Interface `IDictionary` e Polimorfismo

Com o objetivo de permitir a comparação intercambiável entre as múltiplas estruturas de dados, o projeto foi concebido com uma arquitetura orientada a objetos baseada em polimorfismo. O núcleo dessa arquitetura é a interface genérica `IDictionary<Key,`

Value>, uma classe base abstrata em C++ que define um contrato comum a todas as implementações.

Essa interface declara um conjunto de funções virtuais puras — como add, remove, get e get_all_keys_sorted — que devem ser obrigatoriamente implementadas por qualquer classe concreta que a herde. As classes AVL, RB, ChainedHashTable e OpenAddressingHashTable seguem este padrão, herdando publicamente de IDictionary.

Graças a essa abordagem, a aplicação principal pode manipular qualquer uma das estruturas por meio de ponteiros polimórficos (neste caso, std::unique_ptr<IDictionary<...>>), sem acoplamento à implementação específica. Isso favorece a modularidade, a legibilidade e a extensibilidade do sistema, permitindo a fácil integração de novas estruturas sem modificar o código principal.

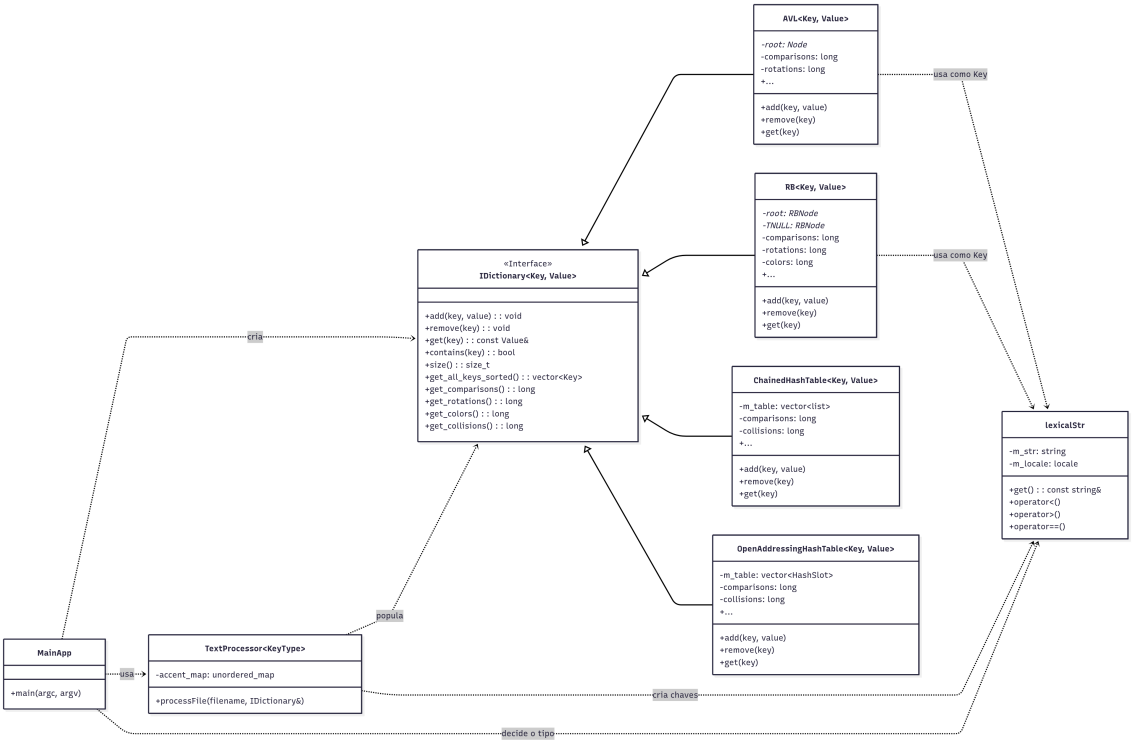

Figure 1. Diagrama de classes do projeto

3.2. Processamento de Texto e Normalização

O processamento textual é realizado pela classe ReadTxt, cuja responsabilidade é ler um corpus de um ficheiro .txt, realizar a normalização de cada palavra e atualizar o dicionário com as suas respetivas frequências.

A função interna clean_word executa a normalização do texto, respeitando as particularidades da língua portuguesa. As principais transformações realizadas são:

- **Remoção de Pontuação:** Símbolos como vírgulas, pontos, aspas e o travessão de diálogo (—) são removidos ou substituídos por espaços, assegurando a correta tokenização das palavras.

- **Preservação de Hífens Internos:** Palavras compostas, como “pão-de-ló”, mantêm o hífen, desde que este esteja entre caracteres alfabéticos, preservando assim o significado do token.
- **Tratamento de Acentuação e Caixa:** A função converte letras maiúsculas (inclusive as acentuadas) para as suas equivalentes minúsculas, mas preserva os acentos. Desta forma, palavras como “É” e “é” são consideradas equivalentes, enquanto “é” e “e” permanecem distintas, conforme os requisitos do problema.

Essa estratégia de normalização garante que a contagem de frequência reflita com precisão as variações linguísticas do português, respeitando as distinções fonéticas e semânticas.

3.3. Classe `lexicalStr` e Ordenação Alfabética

A ordenação alfabética correta de palavras com acentos em português não é garantida pelo `std::string` padrão, cujo operador < realiza uma comparação lexicográfica baseada nos valores binários dos caracteres. Tal comparação resulta em ordenações incorretas, como ”árvore” a aparecer depois de ”zebra”.

Para solucionar este problema, foi implementada a classe `lexicalStr`, um encapsulamento (*wrapper*) para `std::string` que redefine os operadores de comparação (‘¡‘, ‘==‘, etc.). Esta classe utiliza a biblioteca `std::locale`, configurada com o locale `"pt_BR.UTF-8"`, para realizar as comparações de acordo com as regras de collação da língua portuguesa.

Com esta abordagem, as estruturas de dados ordenadas, como as Árvores AVL e Rubro-Negra, são instanciadas com `lexicalStr` como tipo de chave (ex: `AVL<lexicalStr, size_t>`). Consequentemente, a sua ordenação interna é mantida de forma compatível com a ordem alfabética esperada, permitindo que a função `get_all_keys_sorted` obtenha o resultado correto através de uma simples travessia *in-order*, sem a necessidade de um passo de pós-processamento para ordenação.

3.4. Formato de Entrada e Saída

A aplicação é executada via linha de comando, com uma sintaxe simples e flexível. O utilizador deve fornecer, no mínimo, dois argumentos: o tipo de estrutura de dados a ser utilizada e o caminho para o ficheiro de entrada. Opcionalmente, pode-se especificar um nome personalizado para o ficheiro de saída através da flag `--out`.

Sintaxe de uso:

```
./build/main <estrutura> <arquivo\_entrada> [--out <arquivo\_saida>]
```

- `<estrutura>`: Um dos quatro tipos suportados: `avl`, `rb`, `chained_hash` ou `open_hash`.
- `<arquivo_entrada>`: O caminho para o ficheiro `.txt` contendo o texto a ser analisado.
- `--out <arquivo_saida>`: (Opcional) O nome do ficheiro onde o relatório de saída será guardado.

Saída gerada: O programa produz um ficheiro de texto `.txt` que contém um relatório formatado, incluindo:

- As métricas de desempenho da execução (tempo, comparações, rotações, colisões, etc.).
- A lista de todas as palavras únicas encontradas, em ordem alfabética, juntamente com as suas respetivas frequências.

Esta estrutura de entrada e saída torna o sistema robusto, versátil e fácil de integrar em pipelines automatizados de teste ou avaliação.

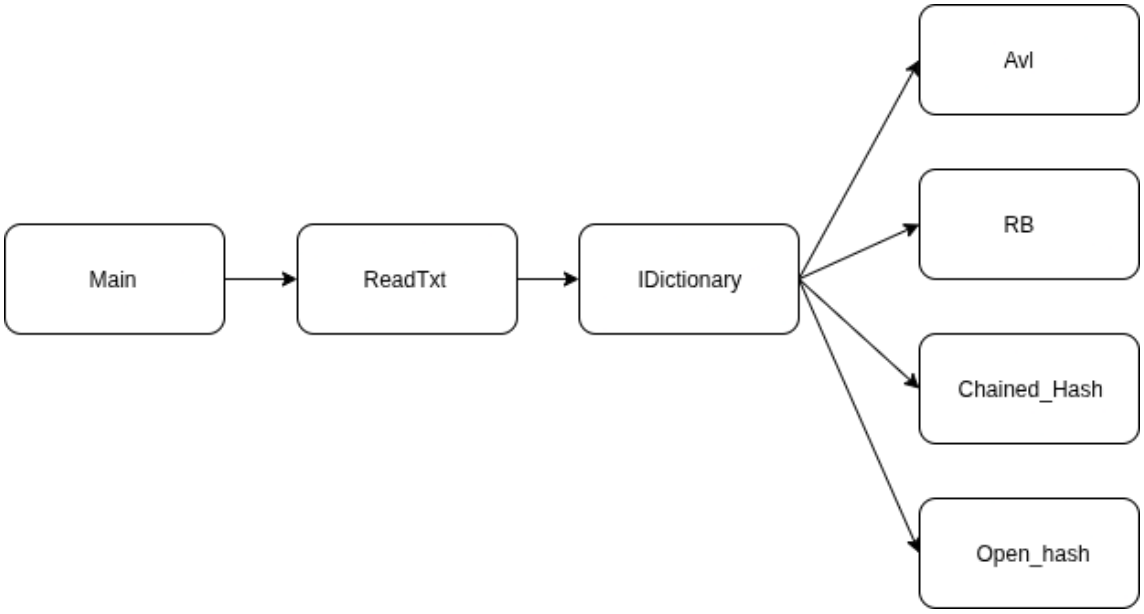

Figure 2. Arquitetura do projeto.

4. Análise experimental

Esta seção descreve o experimento realizado para avaliar o desempenho das estruturas de dados implementadas. O procedimento foi projetado para ser claro e replicável, garantindo a validade dos resultados apresentados. Primeiramente, foram feitos testes usando strings aleatórios e, após isso, através da leitura de arquivos txt.

4.1. Ambiente de Teste

Todos os testes de desempenho foram executados em um único ambiente de hardware e software para garantir a consistência e a comparabilidade dos resultados. As especificações do ambiente são detalhadas a seguir:

- **Hardware:**
  - Processador: `Intel(R) Core(TM) i3-4005U CPU @ 1.70GHz`
  - Memória RAM: `3.7 GiB`
- **Software:**
  - Sistema Operacional: `Ubuntu 24.04`
  - Compilador: `GNU G++ versão 13.3.0`
  - Flags de Compilação: `-Wall -Wextra -std=c++17`. É importante notar que nenhuma flag de otimização (como `-O2` ou `-O3`) foi utilizada, a fim de medir o desempenho bruto dos algoritmos implementados sem interferência de otimizações do compilador.

4.2. Conjunto de Dados para o teste automático

Para realizar uma análise de desempenho robusta, foi gerado um conjunto de dados sintético que simula um cenário desafiador para as estruturas de dados, especialmente para as tabelas hash. O corpus de teste consistiu em:

- **Quantidade de Elementos:** 50.000 chaves únicas.
- **Tipo de Chave:** `std::string`.
- **Geração dos Dados:** As chaves foram geradas como strings aleatórias de 10 caracteres, compostas por caracteres alfanuméricos (maiúsculos e minúsculos) e dígitos.

A escolha por strings aleatórias, em vez de palavras de um texto real ou inteiros sequenciais, foi deliberada. Este tipo de dado garante uma alta entropia e uma distribuição que não favorece nenhuma estrutura em particular, além de aumentar a probabilidade de colisões na tabela hash, permitindo uma avaliação mais rigorosa de suas estratégias de tratamento de colisão.

4.3. Benchmark com Dados Sintéticos

Para mitigar a variabilidade causada por processos do sistema operacional e outros fatores estocásticos, cada benchmark foi executado 5 (cinco) vezes. Os valores apresentados na seção de Resultados correspondem à média aritmética das métricas coletadas ao longo dessas cinco execuções.

O benchmark para cada estrutura de dados foi dividido em duas fases distintas e sequenciais: Inserção e Busca.

1. **Fase de Inserção:** Para cada estrutura, um cronômetro de alta precisão (`std::chrono::high_resolution_clock`) foi iniciado. Em seguida, todos os 50.000 elementos do conjunto de dados foram inseridos na estrutura. Ao final do processo, o cronômetro foi parado, e o tempo total de inserção foi registrado. Durante esta fase, as métricas de comparações e as métricas específicas de cada estrutura (rotações, trocas de cor ou colisões) foram acumuladas.
2. **Fase de Busca:** Imediatamente após a fase de inserção, e com a estrutura já populada, um novo ciclo de medição foi iniciado. O mesmo conjunto de 50.000 chaves foi utilizado para realizar uma operação de busca para cada chave. O tempo total e o número de comparações para esta fase foram registrados separadamente. Este procedimento mede o desempenho de buscas bem-sucedidas no caso médio.

4.4. Teste de Aplicação com arquivos txt

Para avaliar o desempenho em um cenário do mundo real, a aplicação foi executada com quatro obras literárias clássicas: *Dom Casmurro*, a *Bíblia King James (KJV)*, as aventuras de *Sherlock Holmes* e *The Secret Garden*. Este teste avalia como as estruturas lidam com a distribuição de frequência de cada palavra, além de ordená-las em ordem alfabética.

4.5. Métricas Coletadas

Para avaliar e comparar o desempenho de cada estrutura, as seguintes métricas foram coletadas:

- **Tempo de Execução (ms):** Medido em milissegundos, representa o tempo de parede (*wall-clock time*) total para completar todas as operações de uma determinada fase (inserção ou busca).
- **Comparações de Chave:** Um contador que é incrementado a cada vez que uma chave de busca é comparada com uma chave armazenada na estrutura. Esta métrica é um indicador direto do custo computacional do algoritmo de busca da estrutura.
- **Métrica Específica da Estrutura:**
  - **Rotações (Árvores AVL e Rubro-Negra):** Para as árvores auto-balanceadas, este contador mede o número total de operações de rotação realizadas para manter a propriedade de balanceamento durante a fase de inserção. É um indicador do "trabalho" de manutenção da estrutura.
  - **Trocas de Cor (Árvore Rubro-Negra):** Métrica adicional para a Árvore Rubro-Negra, contando o número de vezes que a cor de um nó foi alterada durante as operações de correção.
  - **Colisões (Tabelas Hash):** Para as tabelas hash, este contador mede a frequência com que a função de hash mapeou uma nova chave para um bucket que já estava ocupado (no caso do Encadeamento) ou a frequência com que a sondagem teve que procurar um novo slot após a tentativa inicial (no caso do Endereçamento Aberto).

5. Resultados e análise

A partir dos testes foram obtidos dados que apresentam os resultados numéricos obtidos a partir da execução dos benchmarks e da leitura do arquivos .txt, conforme as metodologias descritas anteriormente.

5.1. Desempenho com Dados Sintéticos

Os valores representam a média aritmética de 5 (cinco) execuções, utilizando um conjunto de dados de 50.000 chaves de strings aleatórias.

A Tabela 1 resume o desempenho de cada estrutura de dados para as fases de inserção e busca.

Table 1. Resultados de Desempenho (Média de 5 Execuções)

Estrutura de Dados	Tempo Inserção (ms)	Tempo Busca (ms)	Comparações (Inserção)	Comparações (Busca)	Métrica Adicional
Árvore AVL	195.07	101.77	1.074.684	1.034.248	Rotações: 34.809
Árvore Rubro-Negra	118.23	68.44	1.080.842	1.139.559	Rotações: 28.859
Tabela Hash Encadeada	91.91	23.07	37.535	74.981	Colisões: 27.576
Tabela Hash End. Aberto	36.25	30.64	69.458	69.458	Colisões: 12.255

5.2. Desempenho com arquivos txt

A seguir apresenta-se tabelas com os dados obtidos na leitura de arquivos txt, onde são colocadas as métricas: Tempo em segundos, comparações, rotações, trocas de cor e colisões.

Nota: todos os tempos nesta seção estão expressos em segundos com 2 casas decimais.

A Tabela 2 apresenta os resultados obtidos ao processar o ficheiro `kjv-bible.txt`, que contém 12.754 palavras únicas.

Table 2. Resultados de Desempenho (File: kjv-bible.txt)

Estrutura de Dados	Tempo (s)	Comparações	Rotações	Trocas de Cor	Colisões
Árvore AVL	9.66	40 435 756	9738	-	-
Árvore Rubro-Negra	8.79	40 267 663	8208	31 133	-
Tabela Hash Encadeada	1.00	3 390 756	-	-	10 997
Tabela Hash End. Aberto	1.11	3 933 724	-	-	4177

A Tabela 3 apresenta os resultados obtidos ao processar o ficheiro `dom-casmurro.txt`, que contém 10.118 palavras únicas.

Table 3. Resultados de Desempenho (File: dom-casmurro.txt)

Estrutura de Dados	Tempo (s)	Comparações	Rotações	Trocas de Cor	Colisões
Árvore AVL	0.93	3 122 023	7474	-	-
Árvore Rubro-Negra	0.83	3 198 148	6248	24 184	-
Tabela Hash Encadeada	0.11	244 718	-	-	7541
Tabela Hash End. Aberto	0.19	338 621	-	-	2587

A Tabela 4 apresenta os resultados obtidos ao processar o ficheiro `sherlock_holmes.txt`, que contém 8.438 palavras únicas.

Table 4. Resultados de Desempenho (File: sherlock_holmes.txt

Estrutura de Dados	Tempo (s)	Comparações	Rotações	Trocas de Cor	Colisões
Árvore AVL	1.57	4 843 249	6067	-	-
Árvore Rubro-Negra	1.53	4 826 073	5069	19 823	-
Tabela Hash Encadeada	0.32	397 255	-	-	6515
Tabela Hash End. Aberto	0.36	505 828	-	-	1815

A Tabela 5 apresenta os resultados obtidos ao processar o ficheiro `the-secret-garden.txt`, que contém 5.342 palavras únicas.

Table 5. Resultados de Desempenho (File: the-secret-garden.txt)

Estrutura de Dados	Tempo (s)	Comparações	Rotações	Trocas de Cor	Colisões
Árvore AVL	1.02	3 743 798	3966	-	-
Árvore Rubro-Negra	0.81	3 657 310	3245	12 671	-
Tabela Hash Encadeada	0.15	332 854	-	-	3928
Tabela Hash End. Aberto	0.16	397 927	-	-	1453

5.3. Análise dos Resultados

Agora, os dados de desempenho apresentados nas Tabelas serão interpretados e discutidos. A análise foca em comparar o desempenho entre as classes de estruturas (árvores vs. tabelas hash) e dentro de cada classe, relacionando sempre os resultados práticos com a teoria de complexidade e as características de cada implementação.

5.3.1. Análise Geral de Desempenho

A análise dos tempos de execução revela uma distinção de desempenho inequívoca entre as duas classes de estruturas de dados. As Tabelas Hash, tanto a de Encadeamento quanto a de Endereçamento Aberto, demonstraram uma performance vastamente superior à das árvores auto-balanceadas em todos os cenários de teste. No benchmark com dados sintéticos, por exemplo, a Tabela Hash com Endereçamento Aberto obteve o menor tempo de inserção (36.25 ms), sendo aproximadamente 5.4 vezes mais rápida que a Árvore AVL (195.07 ms).

Este padrão se manteve nos testes com corpus textuais, validando a complexidade de tempo teórica. As tabelas hash oferecem uma complexidade média de O(1), enquanto as árvores operam em O(logn). A natureza dos textos reais, que seguem uma distribuição de Zipf (poucas palavras muito frequentes e muitas palavras raras), tende a favorecer ainda mais as Tabelas Hash. As inserções de palavras repetidas resultam em operações de atualização extremamente rápidas (uma busca de complexidade média O(1) seguida de uma soma), enquanto nas árvores, mesmo uma atualização exige uma travessia logarítmica.

5.4. Análise Comparativa das Árvores Balanceadas

Ao comparar as duas implementações de árvores auto-balanceadas, a Árvore Rubro-Negra se mostrou consistentemente mais eficiente na inserção em todos os cenários, tanto com dados sintéticos quanto com os corpus textuais. Este fenômeno é explicado pela natureza fundamentalmente diferente de suas operações de rebalanceamento.

A Árvore AVL mantém um critério de balanceamento mais estrito (fator de balanceamento de no máximo 1), o que a força a realizar um número maior de rotações para manter sua invariante. No teste com a Bíblia KJV, por exemplo, a AVL precisou de 9.738 rotações, enquanto a Rubro-Negra necessitou de apenas 8.208. Esta diferença de aproximadamente 18% no número de rotações é um padrão que se repete em todos os testes.

A vantagem da Árvore Rubro-Negra reside na sua flexibilidade. Com suas regras de coloração, ela frequentemente consegue resolver desequilíbrios com operações de troca de cor, que são computacionalmente muito mais baratas do que as rotações (que envolvem a reestruturação de múltiplos ponteiros). No mesmo teste com a Bíblia KJV, a Árvore Rubro-Negra realizou 31.133 trocas de cor. Este ”trabalho invisível” e de baixo custo é o que permite que ela execute menos rotações, resultando em uma construção mais rápida.

Na fase de busca, o desempenho de ambas foi da mesma ordem de magnitude, como esperado, pois ambas garantem uma altura logarítmica. O número de comparações foi muito similar em todos os testes (ex: 40.4 milhões para a AVL vs. 40.2 milhões para a RB na Bíblia KJV). No entanto, a Árvore Rubro-Negra manteve uma ligeira, mas consistente, vantagem no tempo de busca. Embora a AVL seja, em teoria, mais rigidamente balanceada, o que poderia sugerir caminhos de busca mais curtos, a vantagem de desempenho da Rubro-Negra pode ser atribuída a fatores de menor complexidade constante em suas operações ou a uma melhor localidade de cache, resultante de reestruturações menos frequentes e menos drásticas durante a fase de inserção.

5.5. Análise Comparativa das Tabelas Hash

A análise comparativa entre as duas implementações de Tabela Hash revela um trade-off fascinante entre diferentes otimizações. A Tabela Hash com Endereçamento Aberto (usando Hash Duplo) demonstrou ser a mais rápida na inserção com dados sintéticos (36.25 ms), um resultado atribuído à sua superior localidade de cache. Como todos os elementos residem em um único vetor contíguo, ela evita o custo de alocação de memória dinâmica para os nós de uma lista ligada e os saltos na memória (*cache misses*) que ocorrem ao percorrer as listas do encadeamento separado.

A eficácia da estratégia de Hash Duplo é validada pelo número consistentemente menor de colisões em todos os cenários. No teste com a Bíblia KJV, por exemplo, o Hash Duplo gerou apenas 4.177 colisões, enquanto o Encadeamento Separado resultou em 10.997, mais do que o dobro. Isso confirma que a sondagem com passo variável do Hash Duplo é superior em mitigar o agrupamento.

Curiosamente, um menor número de colisões não se traduziu em menos comparações durante a inserção. A Tabela com Endereçamento Aberto realizou mais comparações em todos os testes (ex: 69.458 vs. 37.535 nos dados sintéticos). Isso ocorre porque, nesta implementação, cada sondagem para encontrar um slot vago conta como uma comparação, enquanto na tabela encadeada, as comparações de inserção só ocorrem ao percorrer a lista de um bucket que já sofreu colisão.

De forma consistente, a Tabela Hash Encadeada apresentou os melhores tempos de busca em todos os testes com corpus reais. Em ”Dom Casmurro”, por exemplo, ela foi aproximadamente 72% mais rápida que a de Endereçamento Aberto (0.11s vs. 0.19s). Isso sugere que, para buscas bem-sucedidas, o custo de percorrer listas de colisão curtas e bem distribuídas é, em média, menor do que o custo de múltiplos saltos de sondagem e do cálculo de hashes secundários exigidos pelo Endereçamento Aberto

5.6. Relação entre Métricas e Desempenho

É fundamental observar a correlação direta entre as métricas de complexidade coletadas e o tempo de execução observado. Nas árvores, o número total de comparações, na ordem de milhões para os corpus textuais (ex: 40,4 milhões para a AVL no ficheiro `kjv-bible.txt`), é o principal fator que determina o tempo de execução, refletindo a natureza de sua travessia logarítmica, $O(\log n)$. Na inserção, o custo adicional das rotações, embora menor em magnitude, contribui para o tempo total e explica a diferença de desempenho entre a AVL e a Rubro-Negra.

Nas tabelas hash, o número de colisões durante a inserção é o indicador mais direto da degradação de desempenho. Cada colisão representa um trabalho adicional — seja percorrendo uma lista no Encadeamento Separado ou sondando novos slots no Endereçamento Aberto — que se reflete no tempo de inserção e no número de comparações. A superioridade das tabelas hash é evidenciada pelo número de comparações, que, mesmo em cenários de alta colisão como o do Encadeamento Separado no teste com a Bíblia (10.997 colisões resultando em 3,3 milhões de comparações), ainda é uma ordem de magnitude menor que as 40 milhões de comparações das árvores.

As métricas observadas nos testes com ficheiros reais corroboram as conclusões obtidas nos testes sintéticos. O número de comparações permaneceu como um forte indi-

cador do tempo de execução. A análise dos textos reais, no entanto, revela uma nuance importante na Tabela Hash com Endereçamento Aberto: a eficiência não depende apenas do número de colisões, mas também da distância da sondagem. Colisões que são resolvidas em slots próximos no vetor são menos custosas (devido à localidade de cache) do que sondagens que precisam de saltar para posições distantes. Isso pode explicar as pequenas variações de desempenho entre as duas tabelas hash, mesmo em cenários com diferentes contagens de colisões.

6. Conclusão

Este trabalho realizou com sucesso a implementação e a análise de desempenho rigorosa de quatro estruturas de dados fundamentais — Árvore AVL, Árvore Rubro-Negra, Tabela Hash com Encadeamento e Tabela Hash com Endereçamento Aberto por Hash Duplo — aplicadas ao problema de contagem de frequência de palavras. Os resultados experimentais não só confirmaram, em grande parte, as expectativas da análise de complexidade teórica, mas também revelaram nuances importantes sobre os trade-offs práticos de cada abordagem.

O principal achado deste estudo é a clara superioridade de desempenho das Tabelas Hash em termos de tempo de execução, sendo a implementação com Endereçamento Aberto a mais rápida na construção do dicionário. Em contrapartida, as árvores auto-balanceadas, embora mais lentas, demonstraram um comportamento mais previsível, com um custo de comparações logarítmico consistente, independente da distribuição das chaves. A análise das métricas específicas, como rotações e colisões, foi fundamental para explicar as diferenças de tempo observadas.

Conclui-se, portanto, que não existe uma estrutura de dados universalmente ”melhor”, mas sim uma escolha mais adequada dependendo dos requisitos da aplicação. Para cenários onde a velocidade de inserção e busca é o fator crítico, como na contagem de frequência em massa, as Tabelas Hash são a escolha indiscutível. No entanto, para aplicações que exigem um desempenho de pior caso garantido ou a capacidade de percorrer os elementos de forma ordenada, as Árvores AVL ou Rubro-Negra seriam mais apropriadas.

O desenvolvimento deste projeto reforçou a importância de não apenas compreender a teoria dos algoritmos, mas também de analisar os fatores práticos que influenciam o desempenho, como a localidade de cache e os custos de alocação de memória.

7. Referências

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, 3rd edition.

cplusplus.com (2025). The c++ resources network. `https://cplusplus.com/reference/`. Acessado em: 04 jul. 2025.

cppreference.com (2025a). std::ifstream - cppreference.com. `https://en.cppreference.com/w/cpp/io/basic_ifstream`. Acessado em: 11 jul. 2025.

cppreference.com (2025b). std::locale - cppreference.com. https://en.cppreference.com/w/cpp/locale/locale. Acessado em: 10 jul. 2025.

cppreference.com (2025c). std::unique_ptr - cppreference.com. https://en.cppreference.com/w/cpp/memory/unique_ptr. Acessado em: 10 jul. 2025.