Contador de Frequências Usando Estruturas de Dados Avançadas

Victor Martins Vieira[1]

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

victormartinsbr@alu.ufc.br

Resumo. *Este trabalho foi proposto como avaliação final da disciplina de Estruturas de Dados Avançadas da Universidade Federal do Ceará, campus de Quixadá, ministrada pelo professor Atílio Gomes Luiz. O trabalho consiste na implementação de quatro estruturas de dados: as árvores de busca balanceadas AVL e Rubro-Negra, e as Tabelas Hash com tratamento de colisões por encadeamento externo e por endereçamento aberto. Além disso, o presente estudo busca comparar essas estruturas com base no número de comparações de chaves, no tempo de execução de cada uma, na quantidade de rotações realizadas — no caso das árvores — e na quantidade de colisões realizados para as tabelas.*

1. Introdução

Dicionários são estruturas de dados que armazenam pares chave-valor, onde cada chave é única e utilizada para acessar o valor associado a ela. Dessa forma, os dicionários são amplamente utilizados para operações de busca, inserção e exclusão de dados com eficiência, uma vez que o tempo médio dessas operações costuma ser $O(1)$ ou $O(\lg n)$, dependendo da estrutura utilizada como base.

Dentre suas múltiplas aplicações, a contagem de frequências é uma técnica que permite ampla aplicação em problemas práticos, tais como: análise de textos (contagem de palavras, letras, n-gramas); compressão de dados (como no algoritmo de Huffman); reconhecimento de padrões (como a identificação de elementos mais comuns); e, na área de *Machine Learning*/NLP, para a criação de vetores de características (ex.: Bag of Words).

No presente trabalho, objetiva-se a implementação de dicionários para a contagem de frequência de palavras a partir das seguintes estruturas:

- Árvore AVL;
- Árvore Rubro-Negra;
- Tabela Hash com tratamento de colisão por encadeamento externo;
- Tabela Hash com tratamento de colisão por endereçamento aberto.

Além disso, as seguintes métricas serão definidas como forma de comparação entre os dicionários implementados, a partir da execução com as mesmas entradas:

- Tempo de execução (para todos);
- Número de comparações de chaves necessárias para montar a tabela de frequências (para todos);
- Número de rotações realizadas (para os dicionários implementados com árvores);
- Número de colisões (para os dicionários implementados com Tabelas Hash).

As entradas utilizadas são arquivos `.txt` disponibilizados pelo professor, e seu processamento se dará da seguinte forma:

- A leitura do arquivo `.txt` deverá desprezar espaços em branco e sinais de pontuação.
- Os espaços em branco serão considerados como separadores de palavras.
- Todos os sinais de pontuação devem ser desconsiderados, ou seja, não são entradas do dicionário nem fazem parte das palavras. Entre os sinais a serem descartados, incluem-se:
  - Aspas simples e aspas duplas;
  - Sinal de interrogação (`?`);
  - Sinal de exclamação (`!`);
  - Vírgulas (`,`);
  - Dois-pontos (`:`);
  - Ponto final (`.`);
  - Ponto e vírgula (`;`);
  - Símbolos aritméticos como igualdade (=), soma (+), subtração (−), entre outros;
  - Travessões (`|`);
  - Chaves ({ });
  - Colchetes (`[ ]`).
- O hífen (−) possui um caso especial:
  - Quando usado para indicar a fala de um personagem (ex.: `-Olá!`), deve ser descartado.
  - Quando fizer parte de palavras compostas (ex.: *mostrá-lo*, *super-homem*, *pré-história*, *sub-hepático*, *auto-hipnose*, *neo-humanismo*), deve ser mantido e considerado parte integrante da palavra.
- Todas as letras maiúsculas devem ser convertidas para minúsculas durante a leitura.

2. Implementação

Toda a implementação foi feita utilizando a linguagem `C++`, devido às restrições da própria disciplina.

Todo o código implementado está disponível no seguinte repositório no GitHub: Diretório do GitHub.

2.1. Manipulação de Arquivos

As operações de leitura e escrita de arquivos foram implementadas por meio de uma classe auxiliar chamada *FileUtils* que, para simplificar o tratamento de *strings* no projeto, faz uso da biblioteca ICU [Unicode Consortium 2025].

De modo geral, a classe *FileUtils* possui dois métodos estáticos:

- **read_file**: responsável por ler um arquivo `.txt`, extrair suas palavras, aplicar os tratamentos necessários e montar um *vector* com todas as palavras.
- **write_file**: responsável por receber um iterator de pares (palavra, frequência) e o nome do novo arquivo, e criar esse arquivo contendo os dados.

2.2. Estruturas de Dados

Como forma de padronizar os comportamentos, foi criada uma classe pai para todas as estruturas que seriam implementadas. A classe foi definida como um *template*, permitindo que diferentes tipos de chaves (`K`) e valores (`V`) fossem utilizados, aumentando assim sua flexibilidade e reutilização.

Os métodos definidos foram os seguintes :

- `insert(K key)`: insere uma nova chave na estrutura.
- `update(K key, V value)`: atualiza o valor associado à chave informada.
- `get(K key)`: retorna o par chave-valor correspondente à chave buscada.
- `remove(K key)`: remove a chave (e seu valor associado) da estrutura.
- `exists(K key)`: verifica se a chave está presente.
- `size()`: retorna o tamanho atual da estrutura (geralmente o número de elementos armazenados).
- `clear()`: limpa todos os elementos da estrutura.

Além disso, um tipo `Iterator` foi criado de forma genérica, para que todas as estruturas tivessem uma classe derivada que estendesse essa interface, possibilitando a iteração ordenada sobre os dados de cada uma delas.

2.2.1. Árvore AVL

As árvores AVL receberam esse nome por terem sido introduzidas por Adel'son-Vel'skii e Landis em 1962. Em linhas gerais, uma árvore AVL é uma árvore de busca binária balanceada que utiliza, como critério de balanceamento, a regra de que a diferença de altura entre os filhos de cada nó interno não pode ser maior que 1. Dessa forma, garante-se que a altura da árvore com n nós seja da ordem de $O(\lg n)$ [Goodrich and Tamassia 2001].

A implementação da árvore AVL deste trabalho foi baseada no algoritmo proposto por [Weiss 2013].

Um exemplo de uma árvore AVL válida está ilustrado na Figura 1.

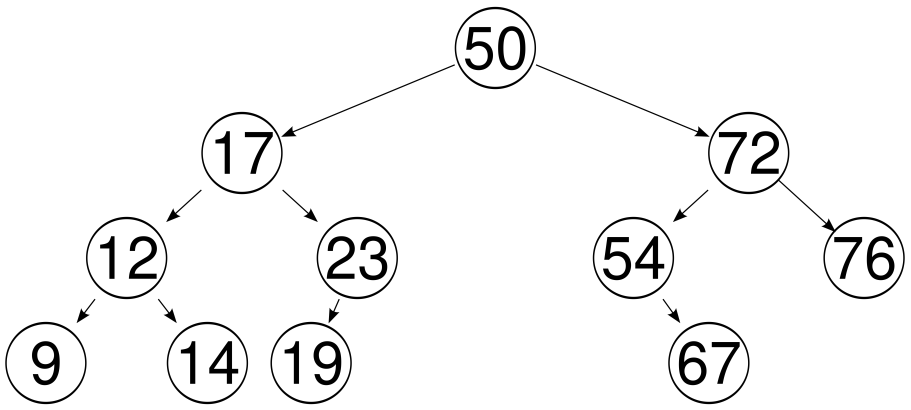

Figure 1. Exemplo de árvore AVL válida
Fonte: Wikipédia (AVL Tree)

2.2.2. Árvore Rubro-Negra

As árvores Rubro-Negras são uma estrutura de dados criada em 1972 por Rudolf Bayer. De forma simplificada, uma árvore rubro-negra é uma árvore de busca com um atributo extra em cada nó para guardar sua cor (vermelho ou preto). A ideia por trás dessa coloração é garantir que nenhum caminho, desde a raiz até uma folha, possua o dobro do comprimento de qualquer outro. Um exemplo de uma árvore rubro-negra válida está ilustrado na Figura 2.

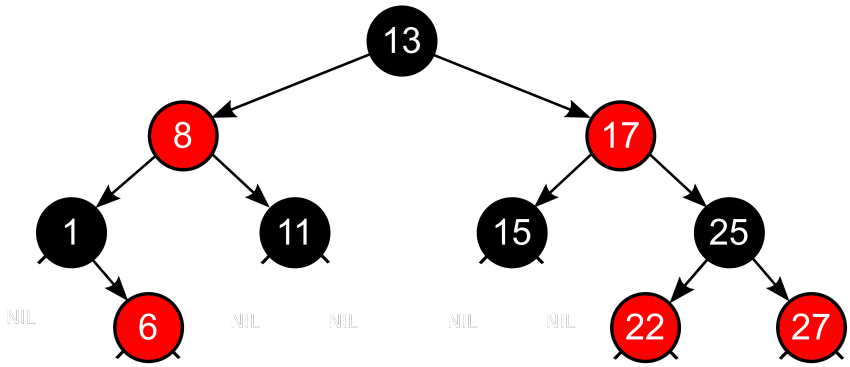

Figure 2. Exemplo de árvore rubro-negra válida
Fonte: Wikipédia (Red Black Tree)

As propriedades que definem uma árvore rubro-negra são as seguintes:

1. Todo nó é vermelho ou preto.
2. A raiz é preta.
3. Todas as folhas (NIL) são pretas.
4. Ambos os filhos de todos os nós vermelhos são pretos.
5. Todo caminho de um dado nó até qualquer uma de suas folhas descendentes contém o mesmo número de nós pretos.

A implementação da árvore rubro-negra utilizada neste trabalho se baseou no conteúdo disponibilizado pelo professor Atílio e na implementação proposta por [GeeksforGeeks 2024].

2.2.3. Tabelas *Hash*

As tabelas *hash* são estruturas de dados projetadas para que as operações de inserção, busca e remoção apresentem, idealmente, complexidade próxima de $O(1)$. Para isso, utilizam uma função *hash*, responsável por distribuir os elementos de forma aparentemente aleatória ao longo de um vetor [Cormen et al. 2022].

No entanto, é importante compreender que uma função *hash* perfeita — isto é, que nunca mapeia duas chaves distintas para a mesma posição — não é viável na maioria dos contextos práticos. Assim, a estrutura deve estar preparada para lidar com colisões, ou seja, situações em que diferentes chaves são mapeadas para o mesmo slot.

No caso das tabelas *hash* que utilizam tratamento de colisões por encadeamento externo, a função *hash* mapeia os valores para um vetor de listas. Quando ocorre uma

colisão, o novo valor é simplesmente adicionado à lista correspondente ao slot calculado. Um exemplo do tratamento de colisões por encadeamento exterior está ilustrado na Figura 3.

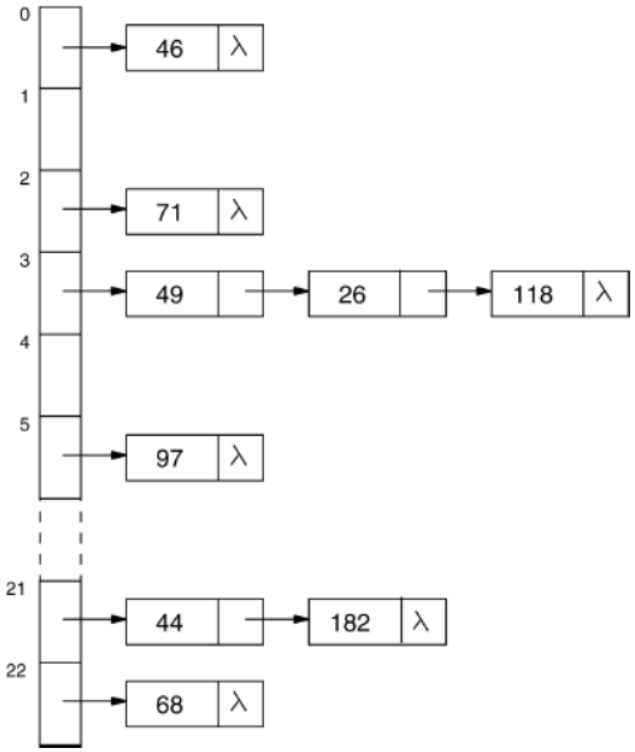

Figure 3. Tratamento de colisões por encadeamento exterior.
Fonte: FIGURA 10.5 [Szwarcfiter and Markenzon 2010] Fonte: Wikipédia (Red Black Tree)

Analogamente, nas tabelas *hash* que utilizam tratamento de colisões por endereçamento aberto, quando ocorre uma colisão, a função *hash* procura sequencialmente (ou de acordo com uma estratégia definida) por outra posição livre no vetor.

Neste trabalho, a implementação das tabelas *hash* baseou-se no conteúdo e nos códigos disponibilizados pelo professor Atílio, utilizados como referência para consulta e revisão.

2.3. Dicionário

Para encapsular a lógica de mapeamento chave-valor, foi implementada a classe `Dictionary`. Essa classe utiliza um atributo do tipo `IDataStruct`, o que permite a utilização de diferentes estruturas de dados por meio do polimorfismo. A escolha da estrutura é feita a partir de um valor do enumerador `Type`, passado como parâmetro ao construtor. Com isso, é possível instanciar dinamicamente uma das implementações filhas de `IDataStruct`, como `RB_Tree`, `AVL_Tree`, `OpenHashTable` ou `ChainedHashTable`.

Além da construção flexível, a classe implementa dois métodos principais: `insert` e `getIterator`. O método `insert` é responsável pela inserção de palavras.

Caso a palavra já exista na estrutura, seu contador é incrementado; caso contrário, a palavra é adicionada com contador igual a 1 (para isso, utiliza os método implementados por cada estrutura: `exists`, `update` e `insert`). Já o método `getIterator` retorna um iterador apropriado para a estrutura de dados utilizada, permitindo a iteração sobre os elementos armazenados independentemente da implementação escolhida.

3. Compilação e Execução

O código para execução está disponível no seguinte repositório.

Para compilar o projeto, é necessário ter instalados o `CMake` (versão 3.28 ou superior), um compilador C++ com suporte à versão C++17 (como `g++` ou `clang++`), e a biblioteca ICU (*International Components for Unicode*). Em sistemas baseados em Debian, os seguintes pacotes são recomendados:

```
sudo apt install build-essential cmake pkg-config libicu-dev
```

3.1. Compilação do Projeto

O projeto utiliza o CMake para gerenciar o processo de compilação. Abaixo estão os passos para gerar o executável:

1. Acesse a pasta raiz do projeto.
2. Crie um diretório para os arquivos de build:
   ```
   mkdir build
   cd build
   ```
3. Execute o CMake para configurar o projeto:
   ```
   cmake ..
   ```
4. Compile o projeto:
   ```
   make
   ```

O comando acima criará o executável `TRABALHO_FINAL` dentro da pasta `build/`.

3.2. Execução do Programa

O programa recebe três argumentos:

- `<estrutura>`: o tipo de estrutura de dados a ser utilizada (`dictionary_avl`, `dictionary_rb`, `dictionary_cht`, ou `dictionary_oht`).
- `<arquivo_entrada>`: caminho para o arquivo de texto a ser processado.
- `<arquivo_saida>`: nome do arquivo onde o resultado será gravado.

Exemplo de uso

```
./TRABALHO_FINAL dictionary_avl ../biblia.txt biblia_avl.txt
```

Nesse exemplo, as palavras contidas no arquivo `biblia.txt` serão processadas utilizando uma Árvore AVL, e o resultado será salvo no arquivo `biblia_avl.txt`.

Mensagens de erro

Caso os argumentos estejam incorretos, o programa exibirá mensagens como:

```
Uso: ./TRABALHO_FINAL <estrutura> <arquivo_entrada> <arquivo_saida>
```

Ou, no caso de uma estrutura inválida:

```
Erro: estrutura de dados "dict_hash" não reconhecida.
```

4. Resultados

4.1. Entradas

As entradas utilizadas neste trabalho foram fornecidas pelo professor e consistem em arquivos `.txt` contendo os seguintes livros: A Riqueza das Nações, Dom Casmurro, Bíblia Sagrada – King James, O Manifesto Comunista, Sherlock Holmes e O Jardim Secreto.

Com o intuito de garantir maior precisão na medição do tempo de execução, foi calculada a média com base em mil execuções para cada livro.

4.2. Ambiente de Execução

As especificações do ambiente de execução utilizados estão descritas na Tabela 1.

Table 1. Especificações do ambiente de execução

Hardware	**Especificações**
Notebook	Acer Aspire A315-23G
Processador	AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx, 2.10 GHz
RAM instalada	12,0 GB (utilizável: 9,94 GB)
Tipo de sistema	Sistema operacional de 64 bits, processador baseado em x64
Armazenamento	SSD NVME 256 GB
Sistema Operacional	**Informações**
Edição	Kubuntu
Versão	24.04 LTS
Kernel	6.8.0-49-generic (64-bit)
Ambiente	**Detalhes**
Programa em execução	Editor de código CLion aberto
Modo de desempenho	”Equilibrado”
Fonte de energia	Ligado à tomada

4.3. Árvore AVL

A partir da execução para árvore AVL, obtemos os valores descritos na Tabela 2.

4.4. Árvore Rubro-Negra

A partir da execução para árvore rubro-negra, obtemos os valores descritos na Tabela 3.

Table 2. Resultados da execução — AVL

Entrada	Tempo (ms)	Comparações	Rotações
A Riqueza das Nações	442.776	269 980	8 288
Dom Casmurro	98.082	248 912	7 543
Bíblia	935.835	435 682	14 227
Manifesto Comunista	15.077	55 424	1 972
Sherlock Holmes	119.690	208 278	6 281
The Secret Garden	85.342	134 548	4 342

Table 3. Resultados da execução — Rubro-Negra

Entrada	Tempo (ms)	Comparações	Rotações
A Riqueza das Nações	172.871	205 563	7 009
Dom Casmurro	40.074	187 383	6 294
Bíblia	403.055	378 458	12 638
Manifesto Comunista	6.023	42 091	1 623
Sherlock Holmes	46.732	157 803	5 298
The Secret Garden	32.945	100 718	3 538

Table 4. Resultados da execução — colisão por encadeamento externo

Entrada	Tempo (ms)	Comparações	Colisões
A Riqueza das Nações	77.420	471 603	8 068
Dom Casmurro	22.092	85 540	7 592
Bíblia	160.281	1 084 727	12 934
Manifesto Comunista	4.002	17 663	1 991
Sherlock Holmes	24.768	135 259	6 703
The Secret Garden	19.145	108 485	5 312

4.5. Tabela *Hash* com tratamento de colisão por encadeamento externo

A partir da execução para Tabela *Hash* com tratamento de colisão por encadeamento externo obtemos os valores descritos na Tabela 4.

4.6. Tabela Hash com tratamento de colisão por endereçamento aberto

A partir da execução para Tabela *Hash* com tratamento de colisão por endereçamento aberto obtemos os valores descritos na Tabela 5.

Table 5. Resultados da execução — tratamento de colisão por endereçamento aberto

Entrada	Tempo (ms)	Comparações	Colisões
A Riqueza das Nações	73.364	909 000	70 390
Dom Casmurro	17.034	175 990	18 523
Bíblia	147.732	1 991 778	131 334
Manifesto Comunista	3.000	37 320	4 156
Sherlock Holmes	20.580	261 954	21 229
The Secret Garden	15.561	220 050	21 328

4.7. Comparação dos Resultados

De modo geral, observou-se melhor desempenho das Tabelas *Hash*, conforme exemplificado na Figura 4, especialmente ao utilizar fator de carga igual a 0,5. No caso da tabela *hash* com tratamento por encadeamento aberto, empregou-se a técnica de *double hashing* com o objetivo de otimizar o espalhamento. Analogamente, no que diz respeito ao tempo de processamento nas árvores, a árvore rubro-negra apresentou desempenho superior, possivelmente devido a um número menor de rotações, como ilustrado na Figura 5.

Quanto às comparações de chaves, representadas na Figura 6, observou-se tendência a um maior número de comparações na tabela *hash* com tratamento por endereçamento aberto. Isso decorre da forma de implementação dessa estrutura, na qual cada chave deve ser verificada sequencialmente até se encontrar um slot válido. Já na implementação com encadeamento externo, esse custo é reduzido graças ao uso do `vector` para armazenar chaves e valores, dispensando múltiplas comparações indesejadas: um novo elemento é simplesmente adicionado via `push_back`. Esse comportamento também é evidenciado pelo número de colisões registrado em cada tabela, mostrado na Figura 7.

5. Conclusão

Com base nos dados coletados com a análise da inserção e contagem de palavras nos livros utilizados, foi possível observar uma série de comportamentos que trazem informações relevantes.

No caso da árvore AVL, observou-se um número elevado de comparações, especialmente em textos longos. Isso é esperado, já que a estrutura precisa manter um balanceamento rigoroso a cada inserção, o que exige verificações contínuas ao longo do caminho percorrido na árvore. Embora mantenha altura logarítmica, o custo adicional

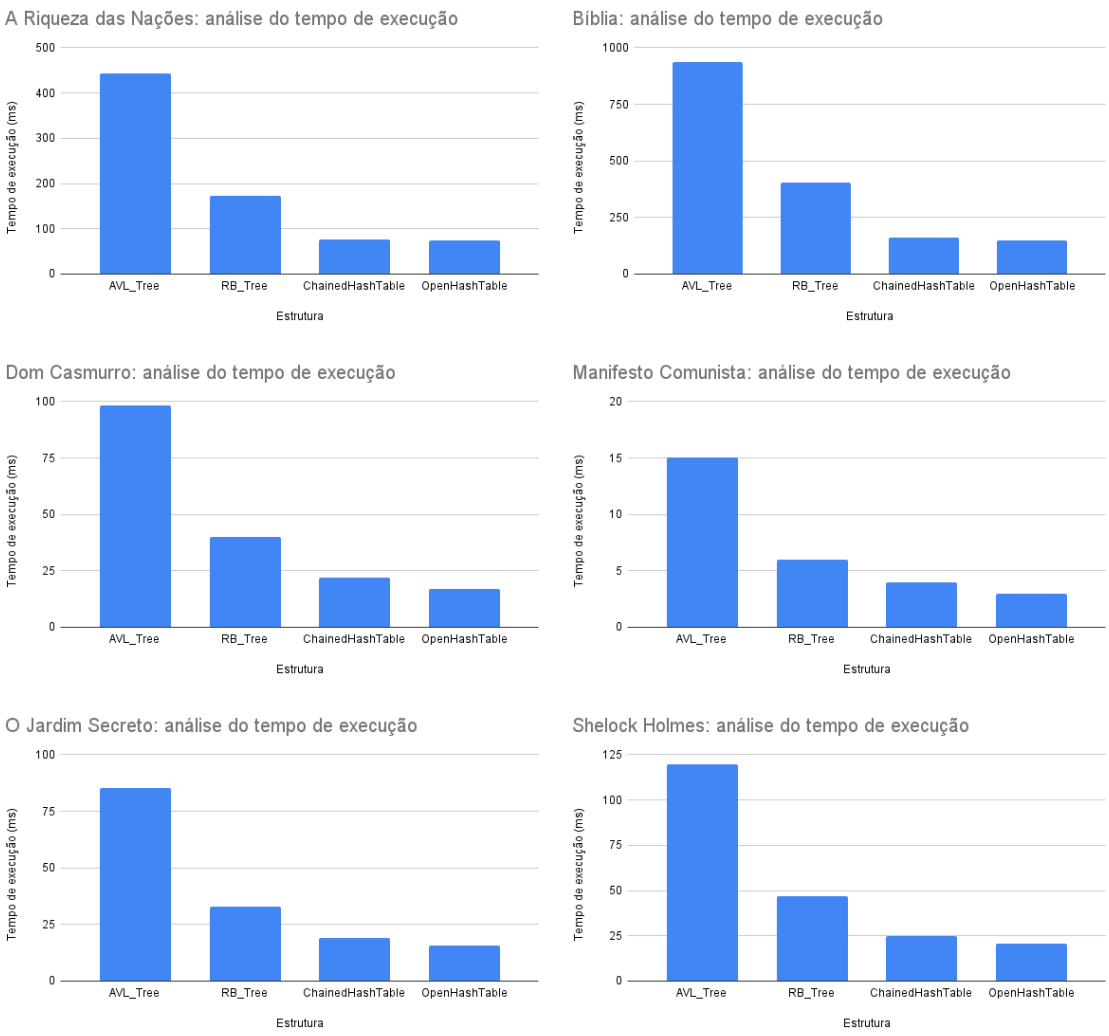

Figure 4. Comparação do tempo médio de execução para cada estrutura

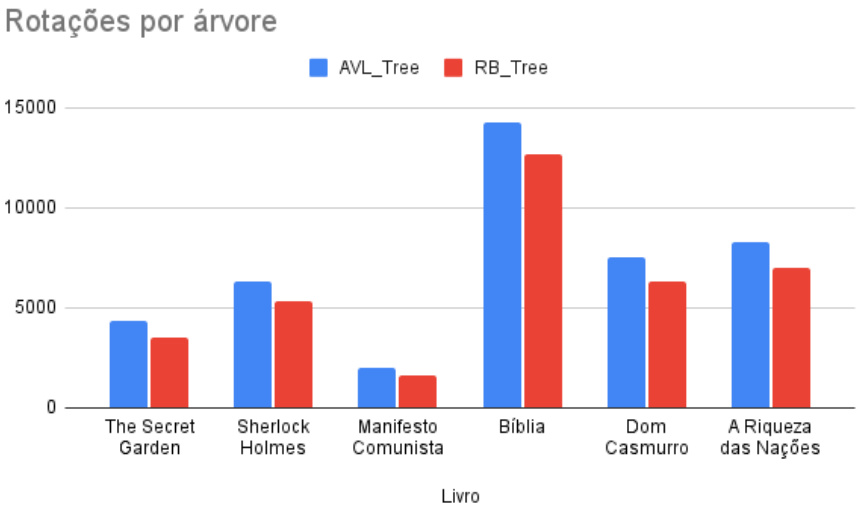

Figure 5. Número de rotações por árvore
Fonte: elaborado pelo autor

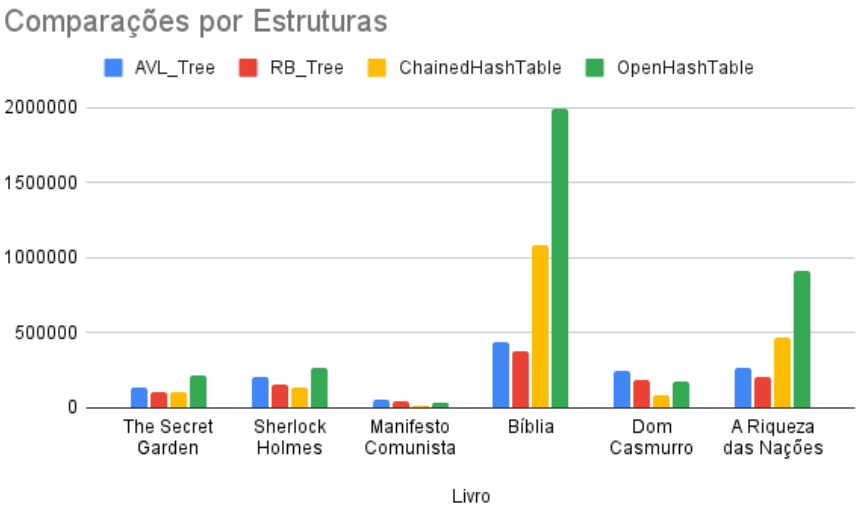

Figure 6. Comparações de chaves por estrutura
Fonte: elaborado pelo autor

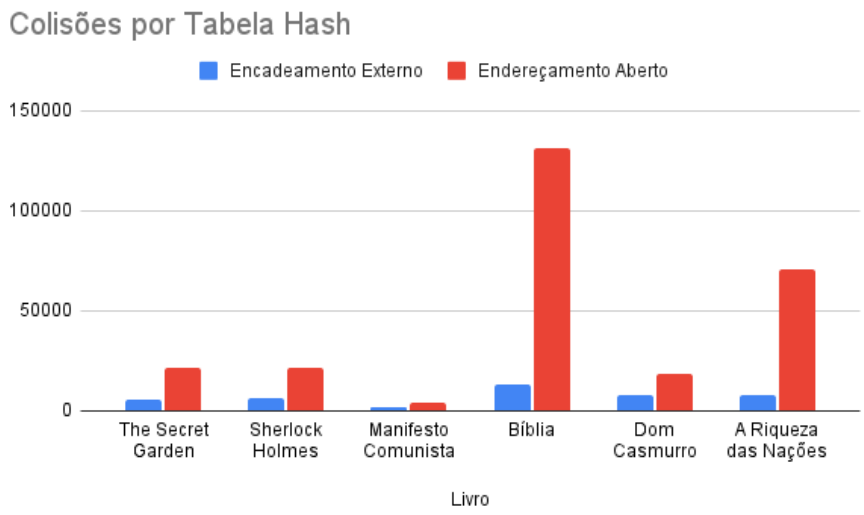

Figure 7. Número de colisões nas tabelas *hash*
Fonte: elaborado pelo autor

de reequilíbrio pode aumentar o número de comparações em relação a outras estruturas, sobretudo em entradas extensas.

Entre as árvores, a rubro-negra demonstrou um desempenho superior, principalmente em relação ao tempo de execução e ao número de comparações. Isso reforça o interesse no uso de um balanceamento mais ”relaxado”, característico dessa estrutura.

A Tabela Hash com encadeamento externo apresentou números significativamente mais baixos de comparações na maioria dos livros, especialmente em textos menores como Manifesto Comunista. Entretanto, em obras mais extensas, como a Bíblia, esse número cresce consideravelmente, o que indica que, mesmo com uma boa distribuição da função hash, há um aumento natural nas colisões e, consequentemente, na necessidade de percorrer listas encadeadas nos buckets.

Por fim, no caso da Tabela Hash com endereçamento aberto, foi possível obter um desempenho superior em termos de tempo em comparação com as demais estruturas. Destaca-se, nesse contexto, a importância de utilizar um fator de carga baixo (0,5 neste trabalho) e uma técnica de hashing que promova maior espalhamento — neste caso, o double hashing.

References

[Cormen et al. 2022] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2022). *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 4 edition. Includes new chapters on matchings, online algorithms, machine learning; 140exercises, 22problems.

[GeeksforGeeks 2024] GeeksforGeeks (2024). Red black tree (rb-tree) using c++.

[Goodrich and Tamassia 2001] Goodrich, M. T. and Tamassia, R. (2001). *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley & Sons, New York. Disponível em: `https://dl.acm.org/doi/book/10.5555/1538644`.

[Szwarcfiter and Markenzon 2010] Szwarcfiter, J. L. and Markenzon, L. (2010). *Estruturas de Dados e Seus Algoritmos*. LTC, Rio de Janeiro, RJ, Brasil, 3 edition. 3ª edição, revisada e expandida com novos capítulos sobre ordenação e listas de prioridades avançadas.

[Unicode Consortium 2025] Unicode Consortium (2025). International Components for Unicode (ICU). `https://icu.unicode.org/`. Acesso em: 29 jun. 2025.

[Weiss 2013] Weiss, M. A. (2013). *Data Structures and Algorithm Analysis in C++*. Pearson, Upper Saddle River, NJ, 4 edition. Disponível em: `https://www.amazon.com/Data-Structures-Algorithm-Analysis-C/dp/013284737X`.