Lexicon: Análise de Estruturas de Dados para Contagem de Frequências de Palavra

Francisco Djalma Pereira da Silva Júnior[1]

[1]Universidade Federal do Ceará - Campus Quixadá (UFC)
Av. José de Freitas Queiroz, 5003 - Cedro Novo - CEP 63900-000 - Quixadá - CE

Resumo. *Este trabalho apresenta a aplicação Lexicon, uma ferramenta desenvolvida para a contagem de frequência de palavras em arquivos de texto, utilizando diferentes estruturas de dados: árvore AVL, árvore Rubro-Negra e duas implementações de tabelas hash (com encadeamento exterior e endereçamento aberto). O objetivo principal é comparar o desempenho dessas estruturas em termos de tempo de execução, número de comparações e atribuições realizadas durante a inserção e busca das palavras. A aplicação lê arquivos no formato .txt, desconsidera espaços em branco e pontuações, e realiza a análise das palavras em ordem alfabética. Os resultados obtidos permitiram avaliar a eficiência das estruturas de dados e compreender as vantagens e desvantagens de cada uma em diferentes cenários de uso.*

Abstract. *This paper presents Lexicon, a tool developed for word frequency counting in text files, utilizing various data structures: AVL tree, Red-Black tree, and two implementations of hash tables (with separate chaining and open addressing). The main objective is to compare the performance of these structures in terms of runtime, number of key comparisons, and assignments during word insertion and lookup. The application reads .txt files, disregards spaces and punctuation, and analyzes the words in alphabetical order. The results obtained allowed us to evaluate the efficiency of each data structure and understand their advantages and disadvantages in different usage scenarios.*

1. Introdução

1.1. Descrição do Problema

A crescente quantidade de dados digitais gerados diariamente demanda ferramentas eficientes para a análise e manipulação dessas informações. Um problema comum em processamento de texto é a contagem de palavras e a análise de frequência. Em muitos cenários, é necessário identificar quais palavras aparecem com maior frequência e organizar essas informações de maneira ordenada para análise ou apresentação. O desafio é implementar uma solução que seja eficiente em termos de tempo e espaço, especialmente quando lidamos com grandes volumes de texto.

1.2. Objetivo do Projeto

O objetivo deste projeto é desenvolver uma aplicação capaz de ler um arquivo de texto (.txt), processar seu conteúdo para identificar e contar a frequência de cada palavra, e apresentar essas informações de forma ordenada.

2. Implementação

A implementação do sistema de análise de frequência de palavras utiliza diferentes estruturas de dados para armazenar e processar palavras. A seguir, apresentamos uma visão geral do método utilizado, das estruturas de dados empregadas e das principais funções da classe responsável pela execução do programa.

2.1. Método Usado para Implementação

O sistema é implementado utilizando C++ e emprega uma abordagem orientada a objetos. A classe principal `Lexicon` é responsável por gerenciar o processamento das palavras, desde a leitura do arquivo de entrada até a geração do arquivo de saída. Diferentes estruturas de dados são utilizadas para armazenar e manipular as palavras, permitindo comparar o desempenho de árvores AVL, árvores Rubro-Negra e tabelas hash com encadeamento e endereçamento aberto.

2.2. Explicação Simples das Estruturas de Dados

Para o armazenamento e análise das palavras, foram utilizadas quatro estruturas de dados principais:

- **Árvore AVL**: É uma árvore binária de busca balanceada onde as diferenças de altura entre subárvores não podem ser maiores que um. Essa estrutura garante operações de inserção, exclusão e busca com complexidade $\mathcal{O}(\log n)$, o que proporciona um desempenho eficiente mesmo em grandes volumes de dados.
- **Árvore Rubro-Negra**: Outra árvore binária de busca balanceada, que mantém o equilíbrio das operações por meio de regras específicas que garantem um tempo de execução $\mathcal{O}(\log n)$ para inserções, exclusões e buscas. É uma alternativa à árvore AVL, com características que favorecem operações mais rápidas em alguns cenários.
- **Tabela Hash com Encadeamento**: Utiliza uma função de hash para distribuir as palavras em diferentes buckets. Em caso de colisões, as palavras são armazenadas em listas ligadas dentro de cada bucket. Essa estrutura é eficiente para operações de busca e inserção, com complexidade média de $\mathcal{O}(1)$ para operações básicas, embora possa se degradar em situações de muitas colisões.
- **Tabela Hash com Endereçamento Aberto**: Também usa uma função de hash, mas resolve colisões ao procurar o próximo espaço livre na tabela. Esse método pode ser mais eficiente do que o encadeamento em termos de uso de memória, porém pode sofrer com a degradação do desempenho em tabelas muito cheias.

2.3. Funções Principais do Projeto

2.3.1. Funções da Classe `Lexicon`

A classe `Lexicon` é responsável por gerenciar o fluxo completo do processamento das palavras, desde a leitura do arquivo de entrada até a geração do arquivo de saída. As principais funções dessa classe são:

- `Lexicon(std::string inp_file, std::string dictionary_type)`: Construtor da classe. Inicializa o tipo de dicionário a ser utilizado com base no parâmetro fornecido e define os nomes dos arquivos

de entrada e saída. Aceita o nome do arquivo de entrada e o tipo de dicionário (”avl”, ”rb”, ”htc”, ”hto”) como parâmetros. Lança uma exceção se o tipo de dicionário fornecido for inválido. O atributo `dictionary` é um ponteiro para um `Dictionary` abstrato, que pode ser uma instância de qualquer uma das quatro implementações: árvore AVL, árvore Rubro-Negra, tabela hash com encadeamento ou tabela hash com endereçamento aberto.

- `void open_files()`: Abre o arquivo de entrada para leitura. Se o arquivo não puder ser aberto, lança uma exceção. Esta função é essencial para preparar o sistema para o processamento dos dados.
- `void run()`: Executa o processamento das palavras. Lê palavras do arquivo de entrada, processa cada palavra e insere-a no dicionário apropriado. Mede o tempo total de execução do processamento, excluindo o tempo gasto na escrita do arquivo de saída e na ordenação (para tabelas hash). Utiliza a função `processWord` para limpar e converter as palavras para o formato Unicode adequado antes de inseri-las no dicionário.
- `void write_output()`: Gera o arquivo de saída com os resultados do processamento. Cria um arquivo que contém o número total de palavras, o tempo de execução, a quantidade de comparações realizadas e, se aplicável, o número de rotações realizadas. Também grava as palavras ordenadas no arquivo de saída. Lança uma exceção se o arquivo de saída não puder ser gerado.
- `size_t elapsed_time()`: Retorna o tempo total de execução do processamento em milissegundos. Esta função fornece uma métrica para avaliar o desempenho do sistema.

Essas funções permitem que a classe `Lexicon` gerencie eficientemente o fluxo de dados e controle o processamento das palavras, oferecendo uma visão completa dos resultados obtidos com as diferentes estruturas de dados.

2.3.2. Classe Abstrata `Dictionary` e Suas Instâncias

A classe abstrata `Dictionary` define uma interface comum para diferentes implementações de dicionários. Ela estabelece os métodos que todas as implementações devem fornecer, garantindo que qualquer estrutura de dados que derive de `Dictionary` possa ser utilizada de forma uniforme. Os principais métodos da classe `Dictionary` são:

- `insert`: Insere uma palavra no dicionário.
- `getOrdenedDictionary`: Obtém uma representação em string do dicionário ordenado.
- `show`: Exibe o dicionário na saída padrão.
- `size`: Obtém o número total de palavras no dicionário.
- `getRotations`: Obtém o número total de rotações realizadas. (Aplicável apenas para estruturas de dados que utilizam rotações, como árvores balanceadas.)
- `getComparisons`: Obtém o número total de comparações realizadas.

As instâncias concretas da classe `Dictionary` são:

- **AVL_Dictionary**: Implementa um dicionário utilizando uma árvore AVL. Esta classe utiliza a `AVLTree` para armazenar palavras e suas frequências.

- **RB_Dictionary**: Utiliza uma árvore Rubro-Negra para implementar o dicionário. Esta classe emprega a `RBTree` para armazenar as palavras e suas frequências.
- **HashTableC_Dictionary**: Implementa um dicionário utilizando uma tabela hash com tratamento de colisões por encadeamento. A classe usa a `HashTable_Chaining` para armazenar palavras e suas frequências.
- **HashTableOA_Dictionary**: Utiliza uma tabela hash com tratamento de colisões por endereçamento aberto. Esta classe utiliza a `HashTable_OpenAddressing` para armazenar palavras e suas frequências.

Cada instância da classe `Dictionary` deve fornecer uma implementação específica para os métodos descritos, adaptada às características da estrutura de dados que representa.

3. Diagrama UML

O diagrama UML a seguir ilustra a estrutura e as relações entre as principais classes do projeto. Ele fornece uma visão geral das classes e como elas interagem, ajudando a entender a arquitetura do sistema.

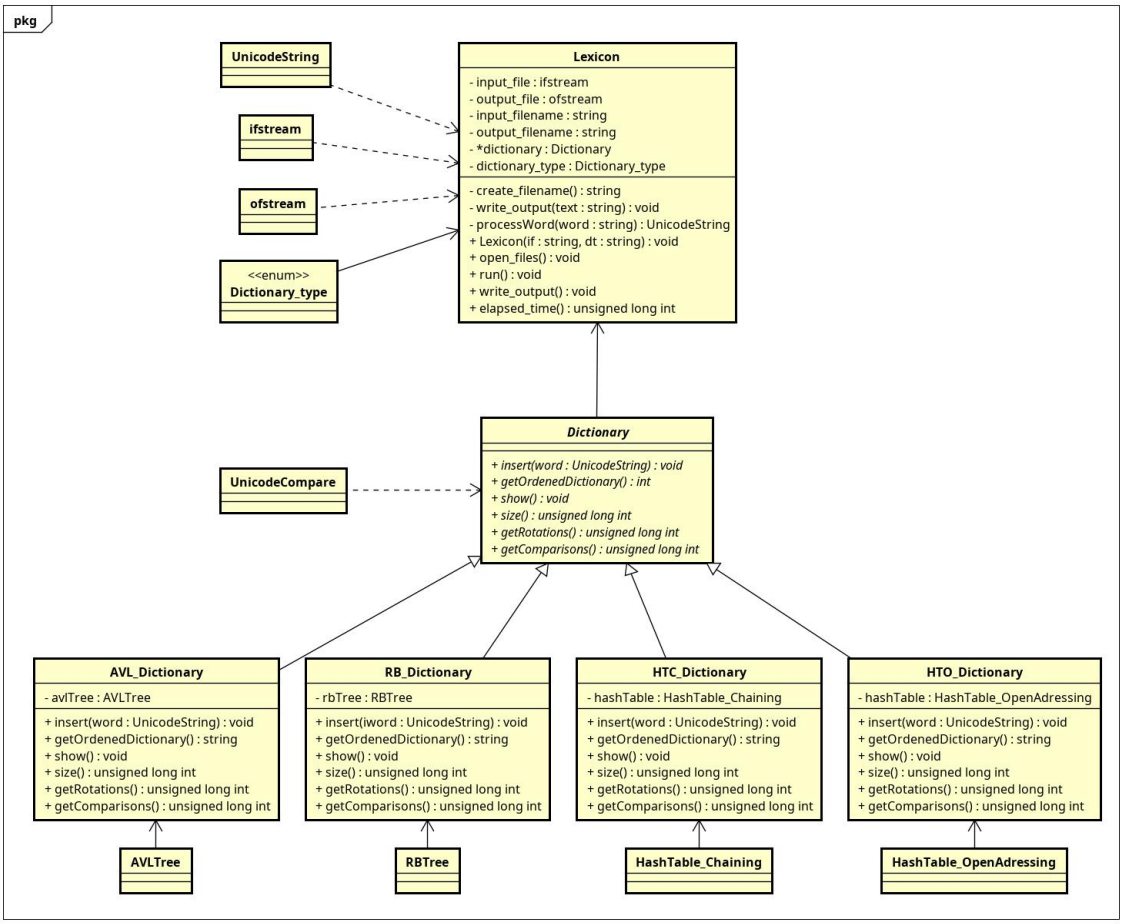

Figura 1. Diagrama UML do projeto `Lexicon`

4. Instruções de Execução

O projeto Lexicon é um programa em C++ que lê um arquivo de texto e gera um dicionário de palavras e suas frequências para fins de comparação de desempenho.

Para compilar e executar o código, siga as instruções abaixo:

4.1. Instalação das Bibliotecas ICU

Para compilar e executar o código, é necessário instalar as bibliotecas ICU (International Components for Unicode) no seu sistema. Para sistemas baseados em Ubuntu/Debian, você pode instalar as bibliotecas ICU usando o seguinte comando:

```
sudo apt-get install libicu-dev
```

Esse comando instalará as bibliotecas ICU necessárias, permitindo que você compile e execute o código.

4.2. Compilação

Para compilar o código usando o compilador g++ e vincular contra as bibliotecas ICU, utilize o seguinte comando:

```
g++ main.cpp -o main -licuuc -licui18n
```

Este comando compilará o arquivo `main.cpp` e gerará um executável chamado `main`. As flags `-licuuc` e `-licui18n` são usadas para vincular contra as bibliotecas ICU. Certifique-se de que as bibliotecas ICU necessárias estão instaladas no seu sistema antes de executar este comando.

4.3. Execução

Para executar o executável `main`, você precisa fornecer dois argumentos de linha de comando: o nome do arquivo de entrada e o tipo de dicionário. Veja um exemplo de como rodar o programa com diferentes tipos de dicionários:

```
./main <nome_arquivo> <tipo_dicionario>
```

4.3.1. Tipos de Dicionário

- Dicionário AVL: `avl`
- Dicionário Rubro-Negro: `rb`
- Tabela Hash com Encadeamento: `htc`
- Tabela Hash com Endereçamento Aberto: `hto`

4.4. Arquivos de Entrada e Saída

Os arquivos de entrada devem ser colocados no diretório `data/in/`, e o programa gerará arquivos de saída no diretório `data/out/`.

Substitua `<nomearquivo>` pelo nome real do seu arquivo de entrada e escolha o tipo de dicionário apropriado.

Lembre-se de instalar as bibliotecas ICU necessárias e compilar o arquivo `main.cpp` antes de executar esses comandos.

5. Resultados dos Testes

Nesta seção, apresentamos os resultados dos testes realizados para comparar o desempenho dos diferentes tipos de dicionários implementados. Cada tipo de dicionário foi testado 50 vezes usando o arquivo *biblia_sagrada_english.txt*, e os seguintes dados foram coletados: tempo médio de execução, desvio padrão do tempo, quantidade de comparações e quantidade de rotações (para árvores AVL e Rubro-Negra).

5.1. Descrição dos Testes

Os testes foram realizados em um ambiente com as seguintes especificações:

- **Processador**: Intel® Core™ i5-1035G1
- **Memória RAM**: 12 GB
- **Sistema Operacional**: Ubuntu 24.04
- **Compilador**: g++ 13.2.0
- **Bibliotecas Utilizadas**: ICU (International Components for Unicode)

O procedimento de teste envolveu a execução do programa 50 vezes para cada tipo de dicionário, com coleta de dados sobre o tempo de execução, comparações e rotações.

5.2. Cálculo das Estatísticas

Para cada tipo de dicionário, foi calculado o tempo médio de execução, o desvio padrão do tempo, e as métricas de comparações e rotações. O tempo médio e o desvio padrão foram calculados utilizando as fórmulas padrão de estatística:

$$\text{Tempo Médio} = \frac{1}{N} \sum_{i=1}^{N} t_i \tag{1}$$

$$\text{Desvio Padrão} = \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} (t_i - \text{Tempo Médio})^2} \tag{2}$$

onde t_i é o tempo de execução da i-ésima execução e N é o número total de execuções (50 neste caso).

5.3. Resultados dos Testes

Os resultados obtidos são apresentados a seguir:

Tabela 1. Resultados dos Testes dos Diferentes Tipos de Dicionários

Tipo de Dicionário	Tempo Médio (ms)	Desvio Padrão (ms)	Comparações	Rotações
Árvore AVL	893	36,633	27.643.743	12.216
Árvore RB	831	14,091	30.092.314	10.310
HT Chaining	474	9,088	975.536	-
HT OpenAdressing	497	4,381	2.402.495	-

5.4. Análise dos Resultados

- **Árvore AVL**: Apresentou um tempo médio de execução de 893 ms e um desvio padrão de 36.633 ms. Apesar de não ser o mais alto, o tempo de execução é considerável, refletindo o custo de manutenção do balanceamento contínuo. O desvio padrão relativamente alto indica uma variação significativa no tempo de execução, possivelmente devido ao impacto das rotações e reequilíbrios. O número de comparações (27,643,743) e o número de rotações (12,216) destacam o custo adicional das operações de balanceamento.
- **Árvore Rubro-Negra**: Mostrou um tempo médio de execução de 831 ms e um desvio padrão de 14.091 ms. Este desempenho é ligeiramente melhor que o da Árvore AVL, com menor variação no tempo de execução. A eficiência em termos de rotações (10,310) é superior à da Árvore AVL, mas o número de comparações (30,092,314) é elevado. Isso pode indicar que, embora a estrutura seja mais eficiente em termos de balanceamento, o custo das comparações pode ser um fator limitante.
- **Tabela Hash com Encadeamento**: Teve o menor tempo médio de execução (474 ms) e um desvio padrão de 9.088 ms. A tabela hash demonstrou a maior eficiência geral em termos de tempo de execução. No entanto, o número de comparações (975,536) é relativamente alto, o que pode ser atribuído ao custo de busca dentro das listas encadeadas em caso de colisão. A eficiência na execução é compensada por um número significativo de comparações, o que pode afetar a performance em cenários com alta carga de dados.
- **Tabela Hash com Endereçamento Aberto**: Apresentou um tempo médio de execução de 497 ms e o menor desvio padrão (4,381 ms). Esta estrutura indicou uma performance geral superior com menor variação no tempo de execução comparada às outras estruturas. O número de comparações (2.402.495) é maior do que na Tabela Hash com Encadeamento, mas o desempenho geral é superior devido à menor variação no tempo médio de execução, refletindo uma maior eficiência nas operações de busca.

6. Conclusão

Durante o desenvolvimento do projeto, algumas dificuldades importantes foram encontradas, como o tratamento de palavras, que envolveu a remoção de caracteres especiais e a utilização da biblioteca ICU, aumentando a complexidade do processamento de texto.

No que diz respeito às estruturas de dados, as Árvores AVL e Rubro-Negra apresentaram uma performance mais estável para operações que requerem a manutenção de uma ordem dos elementos. A diferença principal entre as duas está no nível de balanceamento. A Árvore AVL é uma estrutura mais estritamente balanceada, o que significa que ela realiza mais rotações para garantir que a diferença de altura entre as subárvores de qualquer nó nunca seja maior do que 1. Isso resulta em um maior número de rotações (12.216) comparado à Árvore Rubro-Negra, que é menos estritamente balanceada e permite uma diferença maior de altura entre as subárvores, realizando menos rotações (10.310) para manter o balanceamento. Devido a essa diferença, a Árvore Rubro-Negra apresenta um tempo de execução um pouco melhor (831 ms) em relação à AVL (893 ms), já que a manutenção do balanceamento da AVL demanda um custo adicional de rotações.

As Tabelas Hash, tanto com encadeamento quanto com endereçamento aberto, mostraram um desempenho significativamente superior em termos de tempo de criação das estruturas. No entanto, o tempo adicional de 120ms para a ordenação das tabelas deve ser levado em consideração em aplicações onde a ordenação dos elementos é necessária. Embora esse tempo extra possa impactar o desempenho, ele não chega a inviabilizar o uso das Tabelas Hash.

Concluindo, a escolha da estrutura de dados ideal depende das necessidades específicas da aplicação. Para operações rápidas de inserção e busca, as Tabelas Hash são as mais indicadas, mas, se for necessário manter a ordem dos elementos de forma eficiente, as Árvores AVL e Rubro-Negra são mais apropriadas, com a AVL oferecendo uma estrutura mais balanceada, apesar de um custo adicional de rotações.

Referências

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*, Fourth Edition. MIT Press, 2022.

[2] Jayme Luiz Szwarcfiter, Lilian Markenzon. *Estruturas de Dados e Seus Algoritmos*, 3ª Edição. LTC, 2010.