Relatório do Trabalho de Estrutura de Dados Avançada
Dicionários de Frequência

Alana Maria Sousa Augusto[1]

[1]Universidade Federal do Ceará (UFC)
Av. José de Freitas Queiroz, 5003 – Quixadá – CE – Brazil

alana.augusto@alu.ufc.br

Abstract. *This report presents the final project for the Advanced Data Structures course, taught by Professor Atílio Gomes Luiz. The objective is to implement frequency dictionaries to count word occurrences in .txt files, using four data structures: AVL Tree, Red-Black Tree, Hash Table with separate chaining, and open addressing. Metrics such as key comparison counters were included to evaluate performance. The implementation was done in C++, using object-oriented programming and abstract data types.*

Resumo. *Este relatório apresenta o trabalho final da disciplina de Estrutura de Dados Avançada, ministrada pelo professor Atílio Gomes Luiz. O objetivo é implementar dicionários de frequência para contar ocorrências de palavras em arquivos .txt, utilizando quatro estruturas: Árvore AVL, Árvore Rubro-Negra, Tabela Hash com encadeamento exterior e com endereçamento aberto. Foram incluídas métricas como contadores de comparações de chaves para avaliar o desempenho. A implementação foi feita em C++ com uso de orientação a objetos e tipos abstratos de dados.*

1. Descrição do Projeto

O projeto "Dicionários" consiste na implementação de dicionários de ocorrência baseados em quatro estruturas de dados específicas: árvore binária AVL, árvore binária rubro-negra, tabela hash por encadeamento exterior e tabela hash por endereçamento aberto. O propósito principal desse projeto é receber um arquivo .txt, percorrê-lo, contar a incidência de cada palavra dentro do texto, ignorando sinais de pontuação, e gerar um novo arquivo contendo cada palavra e seu respectivo número referente à quantidade de vezes que ela apareceu no texto.

1.1. Implementação dos dicionários

Ademais, como mencionado anteriormente, tal implementação é feita com quatro EDs diferentes, ou seja, é construído um dicionário para cada estrutura, porém todos possuem as mesmas funcionalidades e finalidades. Além disso, todas as estruturas garantem as seguintes operações:

1. **Criação** (cria um novo dicionário vazio ou com valores pré-definidos em um vetor de pares)
2. **Inserção** (insere um novo par<chave, valor> no dicionário)
3. **Atualização** (atualiza um valor associado a uma chave no dicionário)

4. **Remoção** (remove um par<chave, valor> do dicionário)
5. **Acesso** (acessa um valor associado a uma chave)
6. **Verificar existência** (verifica se uma chave existe no dicionário)
7. **Verificar tamanho** (verifica quantas palavras existem no dicionário)
8. **Mostrar dicionário** (mostra todas as palavras do dicionário e seus valores associados)
9. **Limpeza** (apaga todas as palavras do dicionário)
10. **Destruição** (destrói o dicionário apagando todos seus valores)

Todas essas funcionalidades foram feitas em implementações específicas dos dicionários do projeto. Cada estrutura de dados possui um dicionário próprio responsável por chamar as funções implementadas nas EDs.

Além disso, também é válido ressaltar que, para garantir a generecidade das estruturas, foram utilizados templates para chave (key) e valor (value), e os nós (no caso das árvores) ou os dados (no caso das tabelas hash) são instanciados com o tipo $pair < key, value >$, o que permite que as EDs sejam inicializadas com quaisquer pares de tipos de variável (string, int, float, entre outros). A única característica que quebra a ideia das classes serem genéricas são as métricas, tópico discutido na seção 2.

Abaixo, está o diagrama de classes dos dicionários e suas respectivas estruturas:

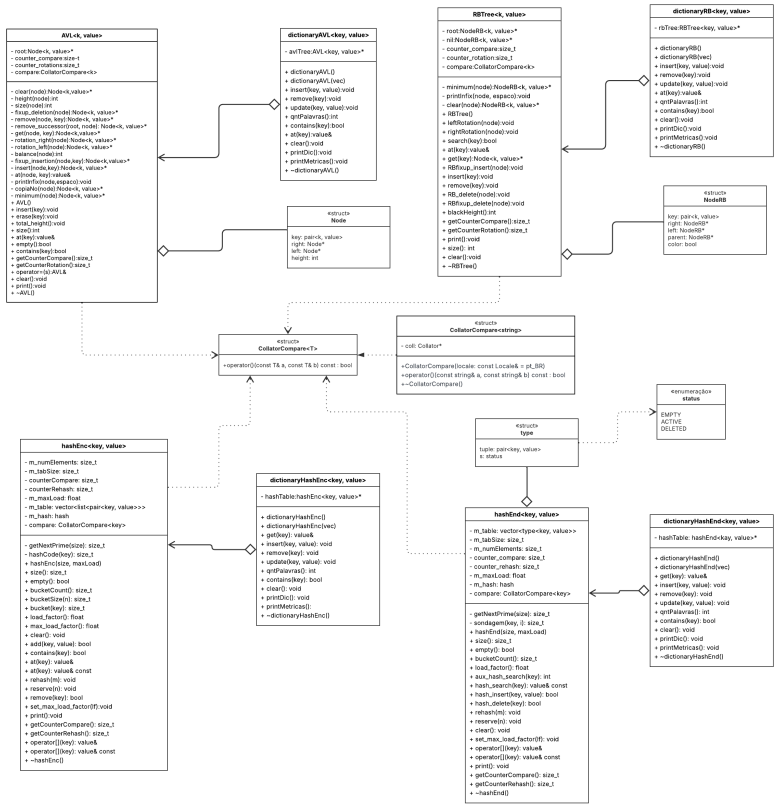

Figura 1. Diagrama de classes dos dicionários

Por último, a seguir é descrito como cada ED foi implementada dentro do projeto:

1.1.1. Árvore AVL

A árvore binária AVL corresponde a uma árvore de busca balanceada, ou seja, ela é construída visando manter a diferença de altura entre as subárvores de cada nó em no máximo 1, garantindo um desempenho eficiente nas operações de inserção, remoção e busca, todas com complexidade $O(log_2 n)$.

No trabalho, a árvore AVL foi implementada com a definição de uma classe genérica parametrizada pelos templates chave e valor. Cada nó da árvore contém um par chave-valor, ponteiros para os filhos esquerdo e direito, além do altura, que serve para calcular o fator de balanceamento de cada nó. A inserção e remoção seguem o mesmo princípio de uma árvore binária de busca, mas após cada operação é verificado o balanceamento da árvore. Caso a diferença de altura entre as subárvores ultrapasse 1 ou seja menor que -1, são aplicadas rotações simples ou duplas (esquerda-direita ou direita-esquerda), conforme necessário.

1.1.2. Árvore Rubro-Negra

Assim como a AVL, a árvore rubro-negra também é uma estrutura de dados do tipo árvore binária de busca balanceada, porém ela mantém o equilíbrio por meio de restrições baseadas em cores (vermelho e preto) associadas a cada nó. Essas restrições garantem que o caminho mais longo da raiz até uma folha não seja mais do que o dobro do caminho mais curto, assegurando que as operações de inserção, remoção e busca sejam realizadas em tempo $O(log_2 n)$.

No trabalho, a árvore Rubro-Negra foi implementada em C++ utilizando uma classe genérica parametrizada pelos templates chave e valor. Cada nó da árvore armazena um par chave-valor, ponteiros para os filhos, para o pai, e uma informação de cor. A inserção é feita de forma semelhante à de uma árvore binária de busca, seguida de uma etapa de correção motivada pelo tio do nó que foi inserido para restaurar as propriedades da árvore Rubro-Negra , por meio de rotações e recolorimentos. A remoção também segue o mesmo padrão de uma árvore binária de busca, entretanto as regras de recoloração e rotação são baseadas no irmão do nó a ser removido fisicamente e no conceito de duplo-x.

1.1.3. Tabela Hash por Encadeamento Exterior

A Tabela Hash por Encadeamento Exterior é uma estrutura de dados que utiliza uma função de hashing para mapear chaves a posições em um vetor, onde cada posição (ou slot) contém uma lista encadeada com os pares chave-valor que colidirem naquele índice. Essa abordagem lida com colisões de forma eficiente, permitindo que múltiplos elementos compartilhem o mesmo índice sem perda de dados.

No trabalho, a estrutura foi implementada em C++ por meio de uma classe genérica com listas encadeadas em cada posição de um vetor principal que representa a tabela em si, além de possuir variáveis que definem informações importantes sobre a estrutura, como o nº de elementos, fator máximo de carga, tamanho da tabela e um objeto da classe hash do próprio C++ que auxilia nas operações da tabela. Para cada operação

de inserção, remoção ou busca, a chave é processada pela função de hash, e a operação é realizada na lista correspondente ao índice calculado. Além disso, a fim de garantir que o número de elementos inseridos não ultrapasse o tamanho da tabela, o máximo fator de carga é definido como 1 por padrão, entretanto o usuário pode passar outro valor caso deseje, desde que ele não ultrapasse 1.

1.1.4. Tabela Hash por Endereçamento Aberto

A Tabela Hash por Endereçamento Aberto resolve colisões armazenando todos os elementos diretamente no vetor principal. Quando ocorre uma colisão, uma nova posição é buscada a partir de uma sequência de sondagem, como linear ou dupla. No trabalho, foi utilizada a sondagem dupla, que utiliza duas funções de hash para determinar os passos de sondagem, reduzindo agrupamentos primários e melhorando a distribuição.

A implementação foi feita em C++ com uma classe genérica que define o vetor de pares chave-valor, além de um campo de status (ativo, vazio ou removido) em cada slot e informações relevantes sobre a implementação, tais como o tamanho da tabela, o número de elementos e um objeto da classe hash do próprio C++. Assim como na hash por encadeamento exterior, o máximo fator de carga também é definido por padrão, entretanto, nessa tabela em questão, é interessante que a tabela nunca fique cheia por completa, então o padrão ideal definido é $\alpha = 0.75$, ou qualquer outro valor tal que $0 < \alpha < 0.75$. Durante as operações de inserção, remoção e busca, as sondagens por hashing duplo são executadas até encontrar uma posição apropriada.

1.2. Comparação de strings com acentos gráficos

Pensando que os dicionários, dentro da aplicação, seriam utilizados para fins de contagem de ocorrências de palavras em textos quaisquer, uma preocupação relevante foi garantir que as palavras com acentos gráficos tiradas de textos escritos em idiomas específicos, como o próprio português, fossem comparadas de maneira correta para a construção das estruturas. Como o C++, por padrão, não possui nenhum suporte específico para tratamento de strings com sinais e, devido a isso, os operadores de comparação padrões $(>, <, ==, \leq, \geq)$ não funcionam corretamente nesses casos, foi preciso utilizar uma biblioteca à parte do C++.

A biblioteca escolhida para isso foi a ICU (International Components for Unicode). Ela é responsável por facilitar o desenvolvimento de aplicações que lidam com texto em diferentes idiomas e culturas, oferecendo recursos como formatação de datas, números, ordenação, conversão de codificações e manipulação de strings Unicode[1]. O ICU foi utilizado, dentro do projeto, especialmente para realizar comparações entre chaves que são strings unicode. Para fazer isso, foi criado um arquivo, cujo nome é compare.hpp, na mesma pasta onde estão as estruturas.

O arquivo compare.hpp utiliza e inclui os seguintes componentes da biblioteca unicode:

- **unistr.h** $\rightarrow$ fornece a classe *UnicodeString* que manipula strings Unicode.

[1] "Unicode é um padrão de codificação que atribui um valor numérico único a cada caractere, independentemente da plataforma, programa ou idioma. " [Lenovo, 2024]

- **coll.h** → fornece a classe *Collator*, que compara string com regras de ordenação linguísticas.
- **locid.h** → fornece a classe *Locale*, que especifica o idioma e país

A principal função desse arquivo é a criação de structs denominados como "CollatorCompare", que sobrecarregam o operador parênteses e fazem com que ele receba duas chaves (a, b) como parâmetros do tipo template e retorne *true*, caso 'a' anteceda 'b' ou *false*, caso contrário. Como os dicionários implementados são genéricos, as chaves que serão comparadas podem ser de qualquer tipo, logo, essa sobrecarga é feita de duas maneiras:

- **Chave do tipo string:** a comparação é feita utilizando um objeto da classe *Collator* e também através da conversão das strings 'a' e 'b' em objetos da classe *UnicodeString*. Após isso, ocorre a verificação através da chamada da função 'compare'[2].
- **Chave de qualquer outro tipo:** a comparação é feita utilizando os operadores padrões do C++.

1.3. Arquivo main.cpp

A main foi implementada de modo que o usuário possa executar o projeto pelo próprio terminal, seguindo a parametrização explicada na seção 3. O arquivo main.cpp foi construído limitando a passagem de apenas 4 argumentos no terminal ao executar o arquivo executável criado previamente. Além disso, também é verificado se o usuário passou apenas nomes de estruturas de dados válidos nos parâmetros. Se for o caso, é criado um dicionário de acordo com a estrutura selecionada e a função *executeDic* (ler seção 1.4) é executada para construir o dicionário.

1.4. Arquivo dic_utils.hpp

O arquivo dic_utils.cpp contém as principais funções responsáveis pela construção completa dos dicionários, com base no arquivo .txt fornecido como entrada no terminal durante a execução do projeto. A seguir, são descritas as funções implementadas neste arquivo:

- ***void executeDic(string file_in, Dictionary& dic, string file_out, string structure)***
  A função executeDic é uma função template que processa um arquivo de texto para construir um dicionário de palavras e registrar a frequência de ocorrência de cada uma. Ao ser chamada, a função tenta abrir o arquivo de entrada e, se ele for aberto com sucesso, a função lê palavra por palavra do arquivo. Em seguida, verifica se a palavra já está presente no dicionário: se estiver, incrementa seu contador; caso contrário, insere a palavra no dicionário com a contagem inicial igual a 1. Além disso, com o auxílio da biblioteca *chrono*, essa função também conta o tempo de execução de toda a construção do dicionário em nanossegundos e o armazena em uma variável específica.

[2]Função da classe *Collator* que retorna um valor UCollationResult; se valor < 0, 'a' vem antes de 'b', se valor == 0, 'a' == 'b', se valor > 0, 'a' vem depois de 'b'.

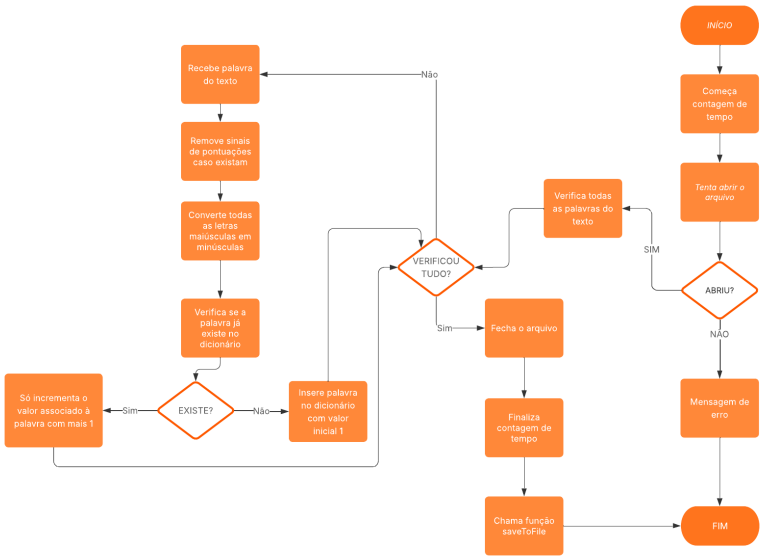

Figura 2. Fluxograma da função executeDic

- ***void saveToFile(Dictionary& dic, const string& file_out, const string& file_in, const string& structure, nanoseconds time)***
  A função saveToFile é responsável por receber o dicionário já povoado com as palavras e as suas respectivas ocorrências no texto e gerar um arquivo de saída formatado contendo uma tabela organizada dessas palavras com suas frequências. Além disso, a função imprime informações gerais sobre o arquivo de entrada, a estrutura de dados utilizada, estatísticas internas da estrutura e o tempo gasto na construção do dicionário convertido em segundos. Essa função faz tudo isso através de métodos específicos da biblioteca *streambuf* que redirecionam temporariamente a saída padrão (cout) para um arquivo, escrevem todos esses dados formatados, e ao final restauram a saída para o console.

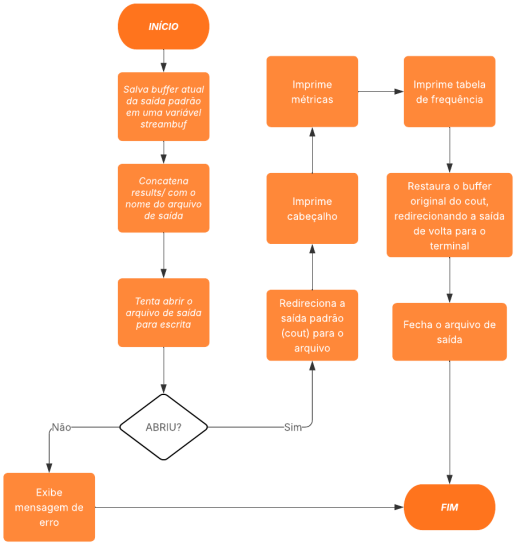

Figura 3. Fluxograma da função saveToFile

- ***string removePunctuationICU(const string& input)***
  Consiste em uma função auxiliar para tratamento de strings que é responsável por remover sinais de pontuação que possam estar associados às palavras lidas no arquivo .txt. Isso é feito com o auxílio de funções específicas do ICU, como *u_isalpha* e *u_isdigit*. A única exceção considereda aqui são palavras que possuem hífen em sua composição, para isso, é feita uma verificação se o hífen está entre duas palavras ou entre dois dígitos, se estiver, o hífen é considerado como parte da palavra.
- ***string toLowerICU(const string& input)***
  Também é uma função auxiliar para tratamento de strings, esta, porém, é responsável apenas por converter todas as letras de cada palavra do arquivo .txt lido em minúsculas seguindo os padrões unicode para o idioma português brasileiro através da função *toLower* do ICU.
- ***void how_to_use()***
  Função que exibe uma mensagem no terminal explicando a sintaxe do comando de execução do projeto caso o usuário digite alguma coisa errada.

2. Métricas utilizadas

A fim de uma melhor observação de como cada estrutura de dado se comporta durante a execução do projeto para um mesmo arquivo .txt, foram adicionadas algumas métricas numéricas na composição de todas as EDs. A principal delas, e que está presente em cada uma das estruturas, é a métrica que conta quantas comparações de chaves foram feitas no decorrer da construção do dicionário. Esse contador é incrementado sempre que alguma comparação entre duas chaves é feita no decorrer da execução. Além disso, foram adicionadas outras duas métricas específicas para cada dupla de estruturas: um contador de rotações para as árvores AVL e Rubro-Negra, e um contador de rehashs para as duas tabelas hash.

2.1. Árvores balanceadas (contador de rotações)

Como as árvores AVL e Rubro-Negra são ambas árvores balanceadas, é de senso comum assumir que eventualmente irão ocorrer rotações na estrutura das árvores, sejam elas à direita, à esquerda, ou dupla à direita ou à esquerda, a fim de mantê-las balanceadas de acordo com suas propriedades individuais. Pensando nisso, foi inserido um contador de rotações para medir quantas rotações cada uma das árvores faz durante a execução de seus dicionários para um mesmo texto, e, assim, comparar o desempenho de ambas nesse quesito. Além disso, é importante mencionar que quando acontece uma rotação dupla, seja ela à direita ou à esquerda, o contador é incrementado duas vezes, visto que fisicamente ocorrem duas rotações na árvore.

2.2. Tabelas hash (contador de rehashs)

Já em relação aos dois tipos de tabela hash (por encadeamento exterior e por endereçamento aberto), foi pensado que seria interessante analisar quantas vezes a função de rehash é chamada na execução de cada um dos dicionários referente às tabelas. É importante explicar que a função de rehash serve para redimensionar a tabela, aumentando seu tamanho quando o fator de carga ($\alpha = \frac{n}{M}$) ultrapassa seu valor máximo pré-definido no construtor da tabela, e, logo em seguida, realocar na tabela os pares de chave e valor seguindo a função que define para que slot cada um deles vai na tabela. O contador

de rehash é interessante justamente porque as duas tabelas tratam as colisões de chave de maneiras diferentes e possuem um fator de carga máximo diferente, o que pode impactar diretamente na frequência com que a realocação se torna necessária, refletindo no desempenho geral da estrutura.

3. Como executar

Para executar este projeto, é necessário ter instalada, na máquina, a biblioteca ICU, seguindo as especificações apontadas na seção 4.
Após garantir isso, abra seu terminal na raiz do projeto e navegue até o diretório /src do projeto:

```
cd \Dicionario\src
```

Então, gere um arquivo executável pelo terminal:
**Caso esteja compilando em uma máquina UNIX:*

```
g++ -Wall -Wextra main.cpp -std=gnu++17 -licuuc -licui18n -o
programa
```

**Caso esteja compilando em uma máquina Windows:*

```
g++ -Wall -Wextra main.cpp -std=gnu++17 -IC:/icu/include
-LC:/icu/lib -licuuc -licuin -o programa
```

Note que os campos -IC:/icu/include e -LC:/icu/lib são caminhos para determinados diretórios, então lembre-se de verificar em que local os arquivos do ICU foram instalados em sua máquina (-I busca a pasta onde estão os headers[3] e -L busca a pasta onde estão as bibliotecas[4]).
Por último, execute o arquivo 'programa', passando os seguintes parâmetros no terminal:

```
.\programa <structure_type> <file_in> <file_out>
```

Observações:
<structure_type>: nome da estrutura a ser utilizada (dicAVL, dicRB, dicChainedHash ou dicOpenHash)
<file_in>: nome do arquivo que será contada a frequência das palvras[5]
<file_out>: nome do arquivo em que a contagem será salva

Exemplo de execução:

```
.\programa dicAVL files/kjv-bible.txt  bible-avl.txt
```

Isso irá gerar um arquivo .txt na pasta 'results' com o nome 'bible-avl' contendo a contagem de frequência de cada palavra do arquivo 'kjv-bible', além de informações sobre as métricas implementadas no dicionário da AVL (ler seção 2.1), e a apresentação de quanto tempo (em segundos) a construção do dicionário em questão levou.

[3] arquivos .h
[4] arquivos .a, .lib, .dll.a
[5] Lembre-se de garantir que o arquivo seja do tipo .txt e que esteja presente no diretório do projeto

4. Especificações

A seguir estão listadas todas as configurações da máquina em que o projeto foi elaborado, feito, compilado e testado.

- **Processador:** Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz
- **Memória RAM:** 8,00 GB
- **Sistema Operacional:** Windows 11 Home Single Language
- **Arquitetura:** 64 bits
- **Memória:** SSD de 238GB
- **Versão do C++:** C++17
- **Versão do MinGW:** MinGW 14.2.0
- **Versão do ICU:** 77.1
- **Versão do Unicode:** 16.0

5. Dificuldades encontradas

5.1. Dificuldades relacionadas à implementação dos dicionários genéricos

- **Elaboração da tabela hash por endereçamento aberto**
  Em relação à implementação da Tabela Hash por Endereçamento Aberto, um dos maiores desafios foi garantir que o método de sondagem dupla estivesse correto e eficiente, especialmente no tratamento de colisões e no cálculo adequado da segunda função de hash. Também foi necessário ter atenção redobrada ao processo de rehash, assegurando que todos os elementos fossem realocados corretamente, sem perda de dados ou falhas de acesso.
- **Implementação dos contadores de comparações de chave.**
  Muitas vezes, havia uma confusão sobre em que momento ele seria incrementado, principalmente em funções que comparavam chaves com vários "if" isolados. A solução mais ideal encontrada foi transformar esses "if" em um conjunto de "if ( ) {} else if ( ) {} else" e contar as comparações baseado em que condição o programa entrou (por exemplo, se entrou no if, incrementa 1 vez, se entrou no else if, incrementa 2 vezes e, por último, se entrou no else, incrementa também 2 vezes).
- **Comparações na árvore AVL**
  Durante os testes, foi percebido que a árvore AVL estava registrando um número excessivo de comparações de chaves em relação às outras estruturas, mesmo com os mesmos dados de entrada. Ao analisar o código, identificou-se que o problema estava na função fixup_insert, que realiza o rebalanceamento após inserções. Nela, o contador era incrementado em várias comparações, inclusive quando nenhuma rotação era necessária. Para resolver isso, foi feita uma distinção clara entre os casos de balanceamento maior que 1 e menor que -1, evitando comparações desnecessárias. Com a correção, o número de comparações caiu significativamente, alinhando o desempenho da AVL com as demais EDs.

5.2. Dificuldades relacionados às funcionalidades do dicionário de ocorrências

- **Tratamento de Strings com ICU**
  Provavelmente uma das principais dificuldades durante a implementação do projeto foi conseguir manipular a biblioteca ICU, visto que, como explicado na seção

1.2, os comparadores padrões do C++ não apresentam suporte para caracteres especiais nas strings, então, foi necessário remodelar os dicionários em relação às comparações de chaves, dado que eventualmente chaves strings acentuadas poderiam aparecer na leitura dos arquivos. Ademais, as implementações de funções específicas de tratamento de strings, como a de remoção de sinais de pontuação no arquivo dic_utils.hpp (ler seção 1.4), também foram bem difíceis de elaborar, entretanto as bibliotecas do ICU facilitaram o processo em grande parte, tornando o código bem estruturado.

- **Gerar arquivos contendo as tabelas de frequência**
  Outro impasse encontrado durante a construção do projeto foi a elaboração de uma forma ideal para gerar os arquivos com os dicionários construídos. Após uma gama de testes com várias opções diferentes para isso, a decisão tomada foi utilizar a biblioteca *streambuf* do C++, através do armazenamento do buffer atual e redirecionamento da saída padrão (cout) do terminal para um arquivo *ofstream*. Essa abordagem permitiu reutilizar diretamente as funções de impressão do dicionário, sem a necessidade de duplicar código ou adaptar chamadas para diferentes fluxos de saída. Após a escrita no arquivo, o buffer original é restaurado, garantindo que o fluxo padrão do terminal continue funcionando normalmente. Essa solução se mostrou simples e eficiente, sendo aplicada de forma uniforme a todas as estruturas de dados implementadas.

6. Listagem de testes

6.1. Testes inciais

Foi feito um arquivo teste.cpp a fim de testar todas as funções implementadas em cada dicionário. Os seguintes testes foram feitos:

- Criação de um dicionário vazio
- Criação de um dicionário a partir de um vetor de pares de chave e valor
- Inserção de elementos
- Remoção de elementos
- Atualização de um valor associado a uma chave
- Acesso de um valor associado a uma chave
- Verificação de existência de chave
- Mostrar quantidade de palavras
- Mostrar o resultado das métricas
- Mostrar dicionário
- Limpeza do dicionário

6.2. Testes finais

A seguir estão listados alguns testes realizados após a conclusão do projeto, para avaliar como cada dicionário da implementação se comporta para diferentes situações específicas e seus respectivos resultados:
Observação: é importante ressaltar que os testes foram realizados em um ambiente controlado, sem a execução de outras atividades simultâneas na máquina, de modo a evitar interferências nos resultados de desempenho.

1. **Teste:** *Execução do arquivo kjv-bible.txt (Bíblia) em cada dicionário*

   Esse teste visa a análise do comportamento de cada estrutura de dados para um arquivo bastante grande. A seguir está a tabela com as estatísticas obtidas dos resultados:

Estatísticas (kjv-bible.txt)				
	AVL	**Rubro-Negra**	**Hash encadeada**	**Hash aberta**
Tempo (em segundos)	4,79643s	3,95721s	1,09622s	1,16059s
Nº comparações	38.279.599	32.040.994	2.225.771	2.414.610
Nº rotações	14.255	12.684	N\A	N\A
Nº rehashs	N\A	N\A	11	11

Figura 4. Estatísticas do arquivo kjv-bible.txt

   É possível notar que o número de comparações é bem discrepante entre as árvores balanceadas e as tabelas hash. Já em relação ao número de rotações das árvores, a árvore AVL tende a executar mais rotações que a rubro-negra. Por fim, em relação aos rehashs, ambas as tabelas hash realizam esse procedimento de modo igualitário nesse caso.

2. **Teste:** *Execução do arquivo os-miseraveis.txt em cada dicionário*

   O arquivo os-miseraveis.txt também foi testado nos 4 dicionários, a seguir está a tabela com os resultados obtidos das estatísticas:

Estatísticas (os-miseraveis.txt)				
	AVL	**Rubro-Negra**	**Hash encadeada**	**Hash aberta**
Tempo (em segundos)	3,32411s	2,4891s	0,71872s	0,768706
Nº comparações	27.421.200	18.445.327	1.345.332	1.422.549
Nº rotações	28.017	23.363	N\A	N\A
Nº rehashs	N\A	N\A	12	13

Figura 5. Estatísticas do arquivo os-miseraveis.txt

   O mesmo padrão observado no arquivo kjv-bible.txt é percebido também para o arquivo os-miseraveis.txt, sendo a única diferença notada o número de rehash que difere em 1 entre as duas tabelas hash.

3. **Teste:** *Execução do arquivo dom-carmurro.txt em cada dicionário*

   Já esse teste foi realizado para analisar o desempenho das estruturas para um arquivo relativamente menor.

Estatísticas (dom-casmurro.txt)				
	AVL	**Rubro-Negra**	**Hash encadeada**	**Hash aberta**
Tempo (em segundos)	0,457523s	0,274152s	0,110505s	0,123778
Nº comparações	3.229.005	2.227.170	170.997	198.649
Nº rotações	7.437	6.293	N\A	N\A
Nº rehashs	N\A	N\A	10	11

Figura 6. Estatísticas do arquivo dom-casmurro.txt

Nota-se que o tempo de construção de cada dicionário diminuiu em cerca de 90% se comparado com o teste realizado com os arquivos kjv-bible.txt e os-miseraveis.txt. Ademais, ainda é percebida a grande discrepância do número de comparações de chaves entre as árvores e as tabelas de dispersão (hash), além da diferença significativa da quantidade de rotações entre as árvores balanceadas. Entretanto, nesse caso, a tabela hash encadeada apresentou um número menor de rehashs quando comparada com a tabela hash por endereçamento aberto. Isso ocorre por causa da diferença de fator de carga das duas estruturas (ler seções 1.1.3 e 1.1.4).

4. **Teste:** *Execução do arquivo pega-terno-rei.txt em cada dicionário*
   O arquivo pega-terno-rei.txt consiste na letra de uma música. Como o arquivo é bem menor quando comparado aos outros dois testados anteriormente, ele foi testado para avaliar o desempenho das estruturas em um tempo bem mais reduzido. A seguir, a tabela com os resultados é apresentada:

Estatísticas (pega-terno-rei.txt)				
	AVL	**Rubro-Negra**	**Hash encadeada**	**Hash aberta**
Tempo (em segundos)	0,002996s	0,0013004s	0,0008535s	0,0011007s
Nº comparações	1.830	1.407	294	396
Nº rotações	16	14	N\A	N\A
Nº rehashs	N\A	N\A	2	2

Figura 7. Estatísticas do arquivo pega-terno-rei.txt

Nota-se, então, que o padrão de diferença de tempo ainda é mantido entre as estruturas, sendo a AVL sempre a que demora mais tempo para executar. As árvores balanceadas continuam tendo bem mais comparações de chaves do que as tabelas, e a árvore AVL executa mais rotações que a rubro-negra.

5. **Teste:** *Execução dos arquivos kjv-bible.txt (Bíblia) e os-miseraveis.txt 100 vezes em cada dicionário*
   Para esse teste, foram criadas main auxiliares na pasta testes que inserem um mesmo arquivo 100 vezes em cada um dos 4 dicionários passando a bíblia e os miseráveis como os arquivos a serem lido. O objetivo principal desse teste é analisar, com maior precisão, o tempo médio gasto, em segundos, para criar e povoar cada dicionário a partir de arquivos com muitas palavras. A seguir, estão as tabelas seguidas de gráficos ilustrados com os resultados obtidos:

Tempo Médio de Execução (kjv-bible.txt) - 100 execuções				
	AVL	**Rubro-Negra**	**Hash encadeada**	**Hash aberta**
Tempo médio (em segundos)	5,07528	4,44077	1,2253	1,27652

Tempo Médio de Execução (os-miseraveis.txt) - 100 execuções				
	AVL	**Rubro-Negra**	**Hash encadeada**	**Hash aberta**
Tempo médio (em segundos)	3,79617	2,99689	0,755941	0,761136

Figura 8. Tempo médio de execução dos arquivos kjv-bible.txt e os-miseraveis.txt

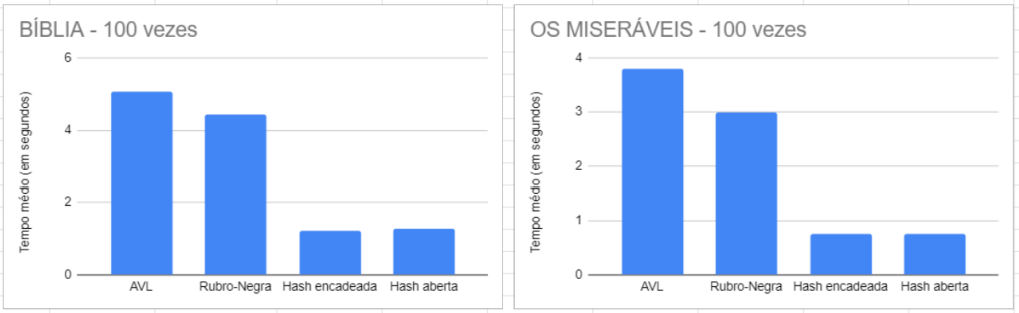

Figura 9. Tempo médio de execução dos arquivos kjv-bible.txt e os-miseraiveis.txt

Percebe-se, portanto, que a árvore AVL é a estrutura que, em média, demora mais tempo para executar. A árvore rubro-negra difere pouco menos de 1 segundo da AVL, enquanto as duas tabelas Hash demoram bem menos e possuem tempos médios bem similares

6. **Teste:** *Execução dos arquivo dom-casmurro.txt e pega-terno-rei.txt 100 vezes em cada dicionário*
   O mesmo teste feito com a bíblia e os miseráveis anteriormente foi feito com as arquivos dom-casmurro.txt e pega-terno-rei.txt, visto que são arquivos bem menores e é desejável verificar se o padrão de tempo médio se mantém nesses casos.

Tempo Médio de Execução (dom-casmurro.txt) - 100 execuções				
	AVL	**Rubro-Negra**	**Hash encadeada**	**Hash aberta**
Tempo médio (em segundos)	0,405137	0,300137	0,0929914	0,0960984

Tempo Médio de Execução (pega-terno-rei.txt) - 100 execuções				
	AVL	**Rubro-Negra**	**Hash encadeada**	**Hash aberta**
Tempo médio (em segundos)	0,00144643	0,000883453	0,000794025	0,00102172

Figura 10. Tempo médio de execução dos arquivos dom-casmurro.txt e pega-terno-rei.txt

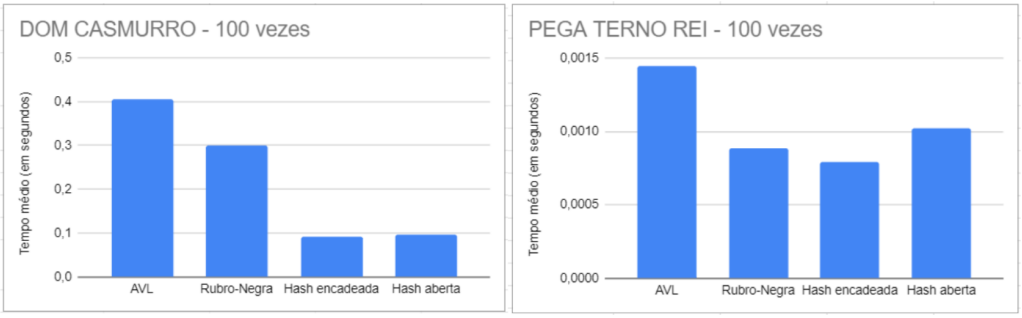

Figura 11. Tempo médio de execução dos arquivos dom-casmurro.txt e pega-terno-rei.txt

Percebe-se, assim, que o padrão de tempo médio observado com a execução da bíblia no teste anterior é sim mantido para esses dois arquivos menores. A AVL continua sendo a estrutura que demanda mais tempo, enquanto a hash por encadeamento exterior é sempre a que leva bem menos tempo.

7. **Teste:** *Execução de arquivos .txt vazios*

   Esse teste busca verificar se o projeto também executa caso sejam passados arquivos .txt sem nenhum texto excrito.

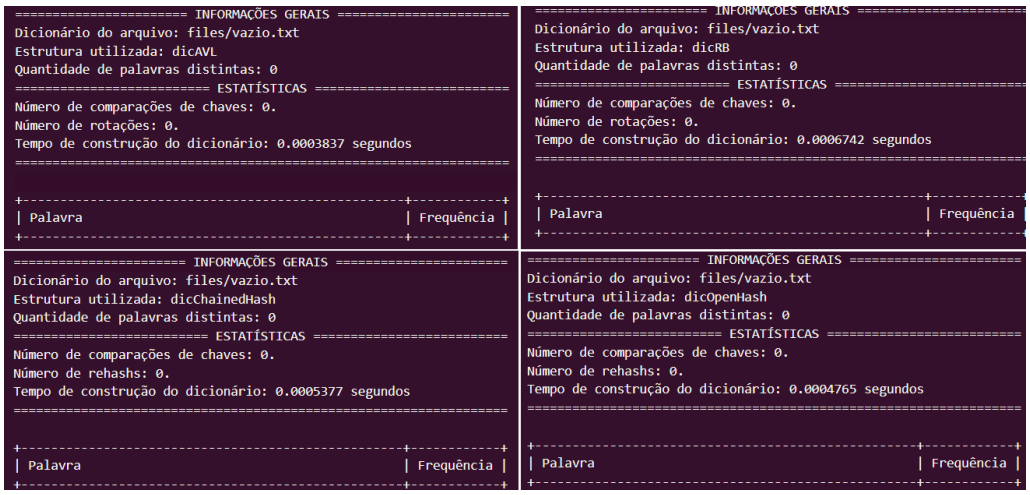

Figura 12. Resultado da construção de dicionários para arquivos vazios

A conclusão é que o projeto é sim executado, gerando apenas um arquivo .txt com as informações do arquivo, sem a contagem de ocorrência de nenhuma palavra.

8. **Teste:** *Execução de arquivo que possui predominantemente sinais de pontuação*

   Esse teste possui como objetivo verificar se o ICU está realizando o tratamento de strings de maneira correta, desconsiderando os sinais de pontuação. O seguinte arquivo foi passado como parâmetro para esse teste:

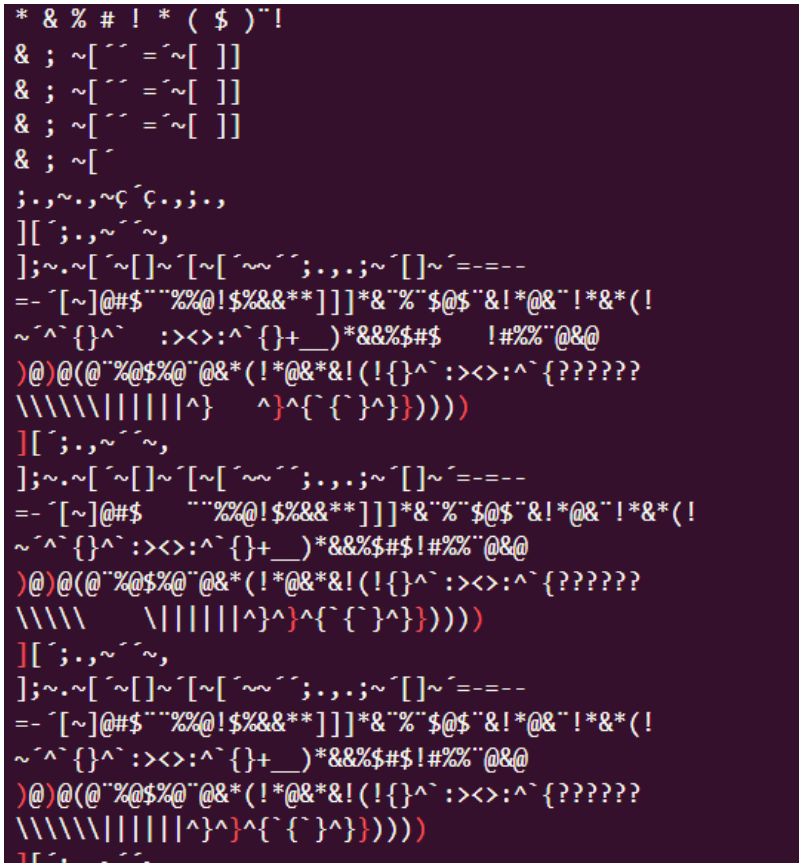

Figura 13. Trecho do arquivo passado como parâmetro no teste

A seguir, estão os resultados obtidos para cada dicionário:

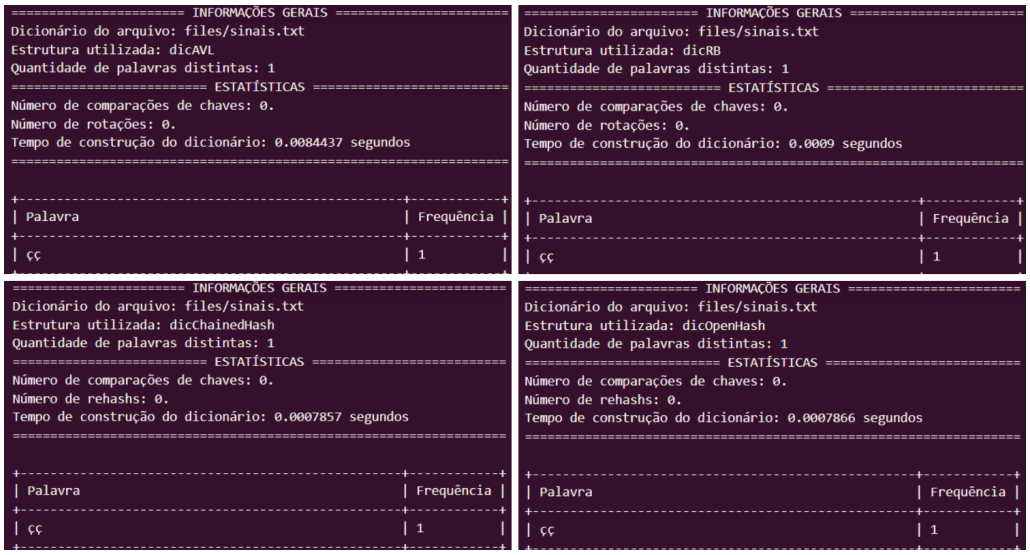

Figura 14. Resultados do teste de sinais de pontuação

Assim, é possível concluir que todas os dicionários estão satisfatoriamente desconsiderando os sinais de pontuação no momento de ler um arquivo.

7. Conclusão

Por fim, conclui-se que a implementação dos dicionários de frequência utilizando diferentes estruturas de dados possibilitou uma análise prática e comparativa entre os métodos abordados durante a disciplina, visto que cada estrutura comportou-se de maneira diferente.

Além disso, a inclusão de métricas como contadores de comparações, rotações e rehashs permite uma avaliação mais precisa do desempenho de cada estrutura, oferecendo uma visão mais concreta dos impactos das decisões de implementação. A utilização de templates e técnicas de programação genérica em C++ também contribue para a flexibilidade do código.

Analisando os resultados obtidos nos testes finais (seção 6.2) feitos com os arquivos kjv-bile.txt, os-miseraveis.txt, pega-terno-rei.txt e dom-casmurro.txt, é possível afirmar que a árvore AVL é sempre a estrutura que demanda mais tempo de processamento para construir seu dicionário, enquanto a tabela hash por encadeamento exterior é a que processa tudo de maneira mais rápida.

Já em relação às métricas de contagem aplicadas e analisadas nos testes individuais para cada arquivo, é possível concluir que o número de comparações de chaves é sempre bem maior para as árvores balanceadas do que para as tabelas hash, sendo a árvore AVL a que apresenta sempre o maior número de comaprações; a árvore AVL normalmente executa mais rotações do que a árvore rubro-negra; e quando há diferença no número de rehashs nas tabelas de dispersão, a tabela hash por encadeamento exterior é a que apresenta um número menor referente a essa métrica.

Através dessas análises, é possível concluir, então, que a tabela hash por encadeamento exterior é a estrutura que apresenta um melhor desempenho para o objetivo da aplicação, enquanto a árvore AVL é a estrutura menos proveitosa. Já a árvore rubro-negra possui um desempenho similar com a árvore AVL, sendo a primeira ainda um pouco mais eficiente. E, por sua vez, a tabela hash por endereçamento aberto é também bem parecida com a tabela por encadeamento exterior, porém, esta última consegue ser um pouco melhor em todos os aspectos.

Referências

[1] cplusplus.com. *std::map - C++ Reference*. Acesso em 20 jun. 2025. 2024. URL: https://cplusplus.com/reference/map/map/?kw=map.

[2] cplusplus.com. *std::unordered_map - C++ Reference*. Acesso em 20 jun. 2025. 2024. URL: https://cplusplus.com/reference/unordered_map/unordered_map/?kw=unordered_map.

[3] ICU4X. *icu_collator - Rust crate documentation*. Acesso em: 9 jul. 2025. 2025. URL: https://docs.rs/icu_collator/latest/icu_collator/.

[4] Lenovo. *What is Unicode?* Acessado em: 9 jul. 2025. 2024. URL: https://www.lenovo.com/us/en/glossary/what-is-unicode/.

[5] Unicode Consortium. *International Components for Unicode (ICU)*. https://icu.unicode.org. Acesso em: 11 jul. 2025. 2025.