Relatório do Trabalho de Estrutura de Dados Avançada Dicionários - Parte I

Clara Cruz Alves[1]

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

claracruz.fac12@alu.ufc.br

Abstract. *This project implements an associative container of the map type using four different approaches, each based on a distinct generic data structure: AVL Trees, Red-Black Trees, Hash Tables with separate chaining, and Hash Tables with open addressing. The application reads a .txt file, processes all words, and builds a sorted frequency table saved to a new file for analysis. Additionally, metrics such as the number of key comparisons, rotations, and collisions are collected to evaluate the performance of each data structure.*

Resumo. *Neste trabalho será desenvolvido um contêiner associativo do tipo dicionário (map), implementado de quatro formas distintas com o uso de diferentes estruturas de dados genéricas: Árvores Binárias AVL e Rubro-Negra, e Tabelas Hash com tratamento de colisões por encadeamento exterior e endereçamento aberto. A aplicação lê um arquivo .txt, processa todas as palavras e constrói uma tabela de frequência ordenada em um novo arquivo para análise. Além disso, métricas como número de comparações, quantidade de rotações e colisões também são coletadas para a avaliação do desempenho das estruturas.*

1. Introdução

Este projeto tem como objetivo desenvolver uma aplicação em C++ capaz de ler um arquivo no formato '.txt' e gerar um novo arquivo contendo uma tabela das palavras encontradas, junto com sua frequência de aparições, ordenadas em ordem alfabética, o programa também deve realizar o devido tratamento das strings, desconsiderando pontuações, acentos e distinções entre letra maiúsculas e minúsculas.

Para armazenar as palavras e suas respectivas frequências, será necessário implementar dicionários, com quatro estruturas de dados distintas: árvore AVL, árvore Rubro-Negra, Tabela Hash com tratamento de colisões por encadeamento exterior e Tabela Hash com tratamento de colisões por endereçamento aberto. Cada uma delas deve ser desenvolvida de forma genéricas, por meio de templates e seguindo os princípios da programação orientada a objetos, com o uso de classes, métodos públicos e privados.

Tanto os dicionários quanto suas respectivas estruturas implementam funções fundamentais, como criação, inserção, atualização, acesso, remoção, verificação de existência, obtenção do tamanho e limpeza, além das funções específicas de cada uma que garantem seu funcionamento interno.

A única exceção que foge à regra da generalidade, são as métricas de contagem, como os contadores de comparação de chave, rotações, colisões, dentre outro específicos que possuem o objetivo de comparar e analisar o desempenho entre as estruturas.

Dessa forma, alinhado com os requisitos propostos, todas as estruturas foram implementadas separadamente como classes genéricas com encapsulamento dos seus dados internos. Além disso, por decisão de projeto, optei por separar a interface pública da implementação de cada classe, organizando o código em arquivos de cabeçalho (`.hpp`) contendo as assinaturas dos métodos e comentários de funcionamento, e arquivos de implementação (`.tpp`) que possuem a lógica correspondente. Essa decisão foi tomada visando melhorar a organização, facilitando a manutenção e a legibilidade do código. A seguir, será descrita como cada uma das estruturas foram feitas.

2. Implementação das Estruturas de Dados

2.1. AVL

A estrutura AVL garante que a diferença de altura entre as subárvores de qualquer nó seja, no máximo, 1. Isso assegura que a árvore permaneça balanceada e evita casos onde, dependendo da ordem de inserção, leve essa estrutura a ter uma complexidade de tempo linear de $O(n)$. Assim, é assegurado que ao realizar qualquer operação básica, como inserção, remoção e busca, dentro dessa estrutura mantêm no pior caso uma complexidade de tempo $O(lg\ n)$.

Desse modo, a estrutura interna da árvore desenvolvida é composta por nós (node) que armazenam, um par de chave e valor, do tipo genérico Key e Value, representado por std::pair; um inteiro height para guardar a altura do nó, necessário para o cálculo do fator de balanceamento; e dois ponteiros (left, right) para os filhos esquerdo e direito, respectivamente. Referente a classe principal a árvore possui um ponteiro para raiz (root), além de contadores auxiliares escolhidos para métrica, são eles o count_comp para contabilizar as comparação de chaves e count_rotation para contabilizar rotações.

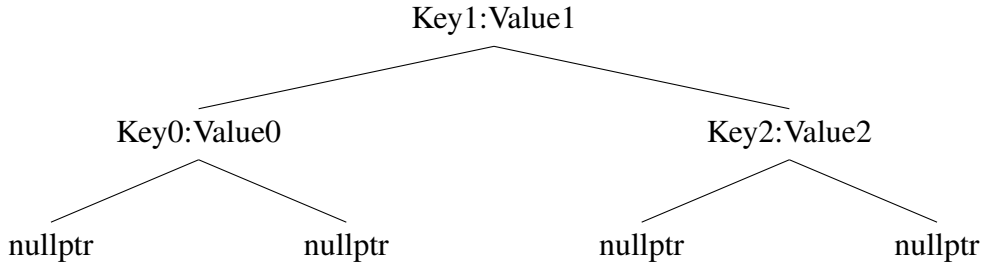

Figura 1. Diagrama ilustrativo da AVL

2.1.1. Funções principais

- **Inserção(insert):** feita chamando à função privada recursiva _insert, que insere um novo nó caso a chave passada já não esteja inserida, se já estiver, faz a atualização do valor associado a chave. Para ficar em conformidade com as propriedades da AVL, já que a inserção de um nó aumenta altura da árvore, é

necessário uma verificação, e um conserto se necessário, para isso é chamado a função fixup_insertion, ao fim de cada inserção, para aplicar as rotações necessárias para manter o balanceamento da estrutura.

- **Remoção(erase):** antes de remover um elemento verifica sua existência chamando a função contains, se a chave estiver na árvore chama a função privada recursiva _erase, esta faz a busca pelo nó onde a chave se encontra, e avalia dois casos: se o nó possui os dois filhos, onde é necessário achar o sucessor do nó; ou se possui um ou nenhum filho onde ele vai receber o filho existente ou o vazio. Assim como na inserção, pode haver um desbalanço na remoção, por isso é usada a função fixup_erase para calcular o fato de balanceamento pós remoção e realizar as rotações necessárias para árvore ficar no seu estado ideal.

- **Busca(at e contains):** ambas realizam a busca por uma chave na árvore. A função contains retorna um valor booleano indicando a presença da chave, enquanto a função at retorna o valor associado à chave ou lança uma exceção se a chave não for encontrada.

- **Rotações(leftRotation e rightRotation):** responsáveis pela reorganização dos ponteiros dos nós para corrigir desequilíbrios. São utilizadas tanto na fixup_insertion quanto na fixup_erase.

2.1.2. Auxiliares

- **height:** função privada que calcula e retorna a altura do nó.
- **balance:** função privada que calcula e retorna o fator de balanceamento da árvore.
- **minimum:** função privada que retorna o valor mínimo de uma subárvore, usada na função erase para encontrar o sucessor do nó excluído.
- **clear:** feita chamando a função privada _empty que verifica se um árvore é vazia ou não a partir do ponteiro do nó da raiz.
- **size:** função que retorna o total de nós armazenados.
- **empty:** verifica se árvore está vazia.
- **operator[]:** acessa ou insere valores de forma simplificada utilizando o operador colchete.
- **show:** função que demonstra visualmente a estrutura da árvore.

2.2. RBT

A estrutura RBT (Red_Black Tree) ou Árvore Rubro-Negra é uma árvore binária de busca balanceada que mantém o equilíbrio da árvore por meio de regras relacionadas a coloração dos nós. Isso garante que operações como inserção, remoção e busca sejam executadas em tempo $O(lg\ n)$ no pior caso. Diferente da AVL, que utiliza rotações baseadas em altura, a árvore Rubro-Negra utiliza como propriedade o balanceamento pela cor, permitindo que seja mais eficiente em cenários com muitas inserções e remoções consecutivas.

Internamente, cada nó da árvore contém um par chave e valor do tipo genérico Key e Value, representado por std::pair; um ponteiro para o pai (parent); ponteiros para os filhos esquerdo e direito (left, right); e uma flag indicando a cor do nó (RED ou BLACK).

Existe ainda um nó sentinela chamado nil, que representa os ponteiros nulos da árvore. A classe principal mantém um ponteiro para a raiz (root) e três contadores utilizados para fins métricos: count_comp para contar comparações de chave, count_rotation para contar rotações realizadas e count_recolor para contabilizar recolorações.

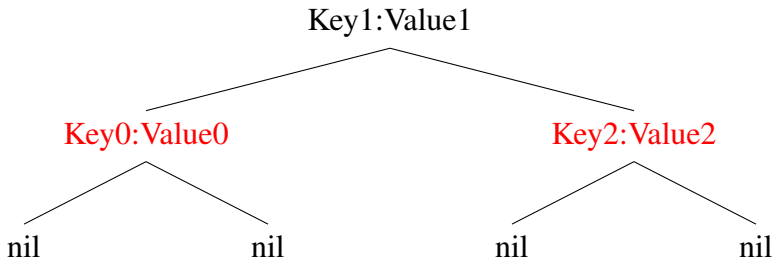

Figura 2. Diagrama ilustrativo da árvore Rubro-Negra

2.2.1. Funções principais

- **Inserção (insert):** a inserção é feita por meio da função privada _insert, que localiza um local adequado na árvore para inserir a nova chave. Se a chave já existir, apenas atualiza seu valor. Caso contrário, cria um novo nó vermelho e chama a função fixup_insertion, que reequilibra a árvore com base nas propriedades da RBT, realizando recolorações e rotações conforme necessário.

- **Remoção (erase):** feita com auxílio da função privada _erase, que busca o nó correspondente a chave e, se encontrado, realiza a remoção por meio da função delete_RB. Esta, trata os diversos casos possíveis de um ou dois filhos, e se necessário, invoca fixup_erase para restaurar as propriedades da árvore Rubro-Negra, utilizando rotações e recolorações conforme o caso.

- **Busca (at e contains):** a função at retorna o valor associado a chave informada, lançando exceção caso não esteja presente. Já contains verifica apenas a existência da chave na árvore, retornando true, se existir e false, caso contrário. Ambas realizam a navegação pela árvore com base nas comparações da chave.

- **Rotações (leftRotation e rightRotation):** funções utilizadas nas etapas de balanceamento (fixup_insert e fixup_erase), realizam o ajuste na árvore mudando a posição relativa entre pai e filho para a esquerda ou direita.

2.2.2. Funções auxiliares

- **minimum:** retorna o nó com a menor chave da subárvore usada pela função delete_RB.
- **clear:** remove todos os nós da árvore recursivamente.
- **size:** retorna o total de nós armazenados.
- **empty:** verifica se a árvore está vazia.

- **operator[]:** acessa ou insere valores de forma simplificada utilizando o operador colchete.
- **show:** função que demonstra visualmente a estrutura da árvore.

2.3. CHT

A estrutura CHT (Chained Hash Table) é uma Tabela Hash que trata colisões por meio de encadeamento exterior. Nessa tabela quando há colisões referentes ao índice do vetor, os pares são colocados no fim de uma listas que tem a função de armazenar múltiplos elementos em um mesmo slot. Essa abordagem evita a necessidade de reestruturações da tabela quando ocorrem colisões, mantendo uma boa performance para inserção, busca e remoção, com tempo médio esperado de $O(1)$.

A tabela é composta por um vetor de listas de pares chave-valor do tipo genérico Key e Value. Cada posição do vetor representa um slot, onde, em caso de colisões, os pares são colocados em uma lista. A classe também possui uma referencia para função de codificação chamada hashing, que transforma a chave em um valor numérico, e também possui um função de compressão (compress) que mapeia esse valor para um índice válido na tabela.

A estrutura também possui variáveis auxiliares para o controle do funcionamento e métricas:

- **numElem:** número total de elementos na tabela.
- **tableSize:** quantidade de slots (tamanho do vetor).
- **maxLoadFactor:** fator de carga máximo permitido antes do rehash.
- **count_comp:** número de comparações de chaves realizadas.
- **count_collisions:** número de colisões detectadas (inserções em slots já ocupados).
- **count_rehash:** número de vezes que a tabela foi redimensionada (rehash).

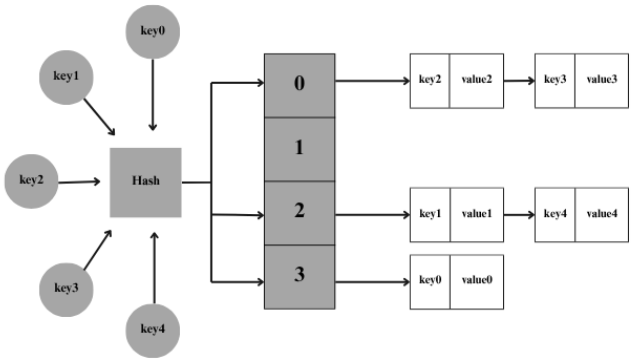

Figura 3. Diagrama ilustrativo da hash com encadeamento exterior

2.3.1. Funções principais

- **Inserção (insert):** antes de inserir um novo elemento, verifica se o fator de carga atual ultrapassou o limite definido. Se sim, realiza um rehash, dobrando o ta-

manho da tabela para o próximo número primo. Em seguida, aplica a função de compressão para determinar o slot e insere o par (k,v)ao fim da lista, caso ele já não tenha sido inserido, caso contrário atualiza.

- **Busca (at e contains):** a função at percorre a lista do slot correspondente a chave e retorna o valor associado. Caso a chave não exista, lança uma exceção. A função contains verifica apenas a existência da chave e retorna um valor booleano.
- **Remoção (erase):** percorre a lista correspondente ao slot e remove o par chave-valor caso encontre a chave. Diminui o número total de elementos da tabela.
- **Redimensionamento (reserve e rehash):** o redimensionamento da tabela ocorre automaticamente quando o fator de carga excede o limite estipulado. Isso é feito por meio da função rehash, que dobra o tamanho tabela, ajustado para o próximo número primo, e copia os seus elementos guardados em uma tabela temporária. No código desenvolvido, o rehash foi implementada como uma função privada, para evitar que o usuário force redimensionamentos desnecessário. Para o caso do usuário precisar fazer um redimensionamento seguro terá a função reserve, que ao receber uma estimativa de quantidade de elementos, verifica se o redimensionamento é necessário e, se for o caso, delega a tarefa a rehash. Essa abordagem evita o redimensionamentos indevidos.
- **Compressão (compress):** transforma o valor de hash gerado pela função hashing(k) em um índice válido, por meio da operação mod tableSize.

2.3.2. Funções auxiliares

- **get_next_prime:** retorna o próximo valor primo do valor passado, é utilizada para mexer no tamanho da tabela.
- **clear:** remove todos os elementos da tabela.
- **empty:** retorna true se a tabela estiver vazia.
- **size:** retorna o número total de elementos armazenados.
- **num_slot:** retorna a quantidade de slots na tabela.
- **slot_size:** retorna a quantidade de elementos em um slot específico.
- **load_factor:** retorna o fator de carga atual.
- **max_load_factor:** retorna o valor do fator de carga máximo permitido.
- **set_max_load_factor:** muda o valor do fator de carga e chama a função reserve para averiguar se será necessário um redimensionamento.
- **operator[]:** acessa ou insere valores de forma simplificada utilizando o operador colchete.
- **show:** demonstra visualmente a tabela no terminal.

2.4. OHT

A estrutura OHT (Open Addressing Hash Table) é uma Tabela Hash que trata colisões utilizando o método de endereçamento aberto. Nesse método, todas as chaves são armazenadas diretamente no vetor da tabela, e quando ocorre uma colisão, a estrutura aplica uma nova tentativa de inserção baseada em uma função de sondagem.

Neste projeto, é utilizada a sondagem dupla (double hashing), onde duas funções de hash distintas determinam os saltos de sondagem. A primeira função define a posição

inicial e a segunda define o deslocamento. A posição final do elemento é então determinada pela fórmula:

$$\mathit{slot} = (h_1(k) + i \cdot h_2(k)) \bmod \mathit{tableSize}$$

A estrutura interna é composta por um vetor de nós, estes possuem em sua estrutura um par de chave e valor do tipo genérico Key e Value, e um campo de status que indica o estado do nó: se está vazio, ativo ou deletado. A classe também possui variáveis auxiliares para manipulação da tabela e métricas de contagem:

- **numElem:** número de elementos atualmente armazenados.
- **tableSize:** tamanho atual da tabela.
- **maxLoadFactor:** fator de carga máximo antes de redimensionar.
- **count_comp:** número de comparações de chave realizadas.
- **count_collisions:** número acumulado de colisões ocorridas.
- **count_rehash:** quantidade de rehashes realizados.

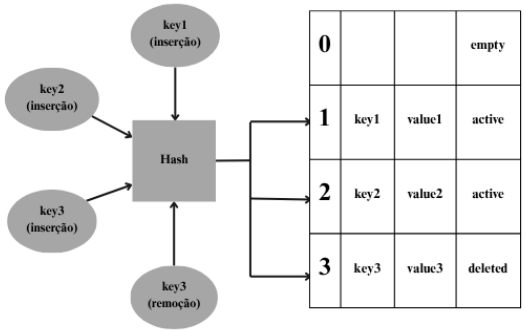

Figura 4. Diagrama ilustrativo da hash com endereçamento aberto

2.4.1. Funções principais

- **Inserção (insert)**: antes de inserir um novo elemento, é verificado se o fator de carga ultrapassou o limite permitido. Se sim, é realizado um redimensionamento da tabela (rehash). Caso a chave já exista, seu valor é atualizado. Caso contrário, a estrutura aplica a sondagem dupla até encontrar uma posição livre.

- **Busca (at e contains)**: a busca percorre as posições geradas pela função de compressão com sondagem, até encontrar o elemento correspondente a chave ou um slot vazio. A função at retorna o valor associado a chave, e contains verifica apenas sua existência retornando um booleano.

- **Remoção (erase)**: semelhante a busca, mas ao encontrar a chave, o slot é marcado como deletado logicamente, preservando a coerência da sequência de sondagem para futuras operações.

- **Redimensionamento(rehash e reserve)**: caso o fator de carga ultrapasse o valor limite, a tabela é redimensionada para o próximo número primo maior que o dobro do tamanho atual, e todos os elementos ativos são reinseridos utilizando a nova tabela. Assim como na implementação com encadeamento (CHT), a função rehash é mantida como privada para evitar que o usuário provoque redimensionamentos não controlados. A função pública equivalente é a reserve, que recebe o novo tamanho desejado e verifica com base no fator de carga se é necessário redimensionar, garantindo maior controle e segurança sobre essa operação.

- **Compressão (compress)**: utiliza duas funções de hash. A hash1 define a posição inicial de busca na tabela, enquanto a hash2 determina o tamanho do salto a cada tentativa. A função compress então calcula o índice final aplicando a técnica de sondagem dupla para tratar as colisões.

2.4.2. Funções auxiliares

- **get_next_prime:** retorna o próximo valor primo do valor passado, é utilizada para mexer no tamanho da tabela.
- **clear:** remove todos os elementos da tabela.
- **empty:** verifica se a tabela está vazia.
- **size:** retorna o número de elementos armazenados.
- **num_slot:** retorna o número total de slots.
- **load_factor:** calcula o fator de carga atual.
- **max_load_factor:** retorna o fator de carga máximo permitido.
- **set_max_load_factor:** muda o valor do fator de carga e chama a função reserve para averiguar se será necessário um redimensionamento.
- **operator[]:** operador de acesso e inserção de forma simplificada.
- **show:** demonstra visualmente a tabela no terminal.

3. Métricas utilizadas

Cada estrutura desenvolvida possui métricas específicas para análise de desempenho, além de contadores gerais, como o número de comparações de chaves realizadas. Algumas dessas métricas são comuns a todas as estruturas, enquanto outras são particulares de acordo com suas características.

No caso das árvores balanceadas, a métrica utilizada em ambas (AVL e Rubro-Negra) é a contagem de rotações, visto que essa é uma característica comum em ambas para o seus balanceamentos. A árvore Rubro-Negra, além disso, também possui um contador adicional que registra o número de recolorações efetuadas, o que é característico desse tipo de estrutura.

Para as tabelas hash, as métricas escolhidas foram a contagem de colisões durante as inserção e a quantidade de redimensionamentos executados, ambas essenciais para avaliar o comportamento dessas estruturas cm diferentes fatores de carga.

Além disso, cada estrutura implementa funções específicas que retornam os valores desses contadores, permitindo uma análise e comparação de desempenho entre as implementações.

4. Implementação dos Dicionários

Dicionários (Map) são contêineres associativos, que podem ser ordenados, com os elementos sendo inseridos em uma ordem predefinida, no caso dos dicionários que usam árvores binárias de busca como estrutura de base, ou podem ser ser desordenados no caso dos que usam tabela hash como estrutura de base. No caso do trabalho em questão temos os dois tipos.

Durante a implementação, todos os dicionários possuem templates, o que os tornam genéricos assim como as estruturas. Os dicionários possuem templates que recebem o tipo da chave, do valor e comparadores como ”menor que”e ”igual a”, que já são inicializados para ser os padrões do C++, porém, se necessário o usuário pode passar seus próprios comparadores na instância do objeto dicionário. Nos templates dos dicionários que usam tabela hash como base, também possuem um tipo hash sendo passado no template que fica no padrão C++ mas pode ser passado outro pelo usuário.

Todos os dicionários possuem a mesma estrutura de implementação, possuem uma instância privada da respectiva estrutura que utilizam e funções públicas iguais, sendo elas:

- **insert:** Insere um novo par no dicionário, antes da inserção é verificado se a chave já está na estrutura, caso já esteja nada é inserido.
- **update:** Atualiza um par de elementos já existentes na estrutura, antes de atualizar verifica se a chave já se encontra na estrutura.
- **at/const at:** Função que retorna o valor associado a chave.
- **erase:** Função que remove um par do dicionário se este não for vazio e se o par estiver na estrutura.
- **contains:** Verifica se um par de elementos está na estrutura, baseado na chave.
- **size:** Retorna o total de elementos dentro do dicionário
- **clear:** Limpa o dicionário, retirando todos os elementos contidos dentro dele.
- **empty:** Verifica se o dicionário está vazio.
- **show:** Mostra como os elementos se encontram dentro estrutura de acordo com o tipo dela (AVL, Rubro-Negra, Chained Hash, Open Hash).
- **operator[]/const operator[]:** Sobrecarga do operador de indexação, retorna o valor associado a chave caso a ela esteja no dicionário, caso contrário, adiciona um valor padrão na estrutura, também pode atualizar o valor relacionado a chave. A versão constante apenas retorna o valor associado.
- **toVector:** Transforma o dicionário em um vetor.
- **metric:** Retorna as métricas referentes a estrutura usada, em forma de string.

5. Tratamento de strings com Unicode

Um problema proposto para esse projeto foi o tratamento de string. Ao receber um texto, é necessário ler apenas as palavras, ignorando espaços em branco e sinais de pontuação,

com exceção do hífen entre palavras. O meio encontrado para solucionar isso foi a utilização da biblioteca ICU (International Components for Unicode) que permite o processamento adequado de textos e facilita principalmente o tratamento de palavras com acentuações, que não foi proposto para solução de forma explícita mas se tornou necessário no decorrer do projeto.

Inicialmente, ao criar uma instância de dicionário, era necessário apenas passar o tipo da chave e do valor, e as comparações de chaves feitas dentro das estruturas usavam os operadores padrões da linguagem ($==$, $!=$, $<$, $>$, $<=$ e $>=$). Isso funciona se for passado tipos nativos do C++ e com o unicodeString do ICU parecia ser o mesmo caso, pelo menos de forma aparente, porque a biblioteca dá suporte para isso, porém as comparações de chaves da biblioteca ocorrem com comparações de bit por bit e poderia ocasionar erro na interpretação correta das palavras com acentuação, por exemplo, a letra ”á”ser maior que ”z”, o que não estaria correto quando fosse feito a ordenação em ordem lexicográfica.

Para resolver esse problema, necessário modificar as estruturas do dicionário para utilizar comparadores personalizados compatíveis. Nos templates dessas estruturas, é possível fornecer comparadores Compare e Equals, que por padrão estão definidos como ”std::less”e ”std::equals_to”, respectivamente, mas também podem ser sobrescritos conforme a necessidade. Assim, ao instanciar o dicionário com a chave unicodeString, são fornecido os comparadores que dão suporte a esse tipo.

5.1. UnicodeComparator

Por isso, foi criado o arquivo UnicodeComparator com a classe uniStringKey, esse arquivo encapsula uma unicodeString e define estruturas (structs) com os comparadores que tratam o objeto dessa classe da forma correta com o uso de um objeto Collator, que vai tratar a comparação das strings com a lógica linguística e não bit por bit. Existem vários níveis de comparação que o collator utiliza, para essa implementação foi escolhido o secundário que é sensível a palavras acentuadas mas não a letras maiúsculas e minúsculas.

Como já dito, o arquivo UnicodeComparator possui as structs dos comparadores, no caso um para igualdade (uniStringEquals) e um para o menor que (uniStringLess). Além desses, também possui outras structs que auxiliam outros tratamentos, como por exemplo no caso da tabela hash possui um tipo hash que passa a função de codificação do tipo da chave, como a chave nesse caso é do tipo unicodeString também é necessário sobrescrever o tipo hash.

5.2. DictIO

Outra proposta do trabalho era que as palavras a serem contabilizadas viessem de um arquivo de texto. A medida que o arquivo fosse lido, as palavras contidas nele deveriam ser contadas. No entanto, antes de serem inseridas no dicionário, essas palavras precisavam passar por um pré-processamento, conforme as exigências definidas. Com os dicionários oferecendo suporte adequado para comparações, bastava preparar a leitura do arquivo para fornecer os dados no formato desejado.

Para isso, foi criada a classe DictIO, responsável por realizar a leitura dos arquivos, processar e inserir as palavras no dicionário, por meio da função read, além de gerar um

arquivo de saída contendo as tabelas com as palavras e suas respectivas frequências por meio da função write.

Na função de leitura, também utilizou-se da biblioteca ICU para tratar as strings. De modo resumido, as etapas são as seguintes:

1. Lê uma linha, normaliza para tratar acentuação de forma padrão (porque o ICU tem uma inconsistência onde uma mesma letra com acento pode ser tratada de forma diferente mesmo sendo igual).
2. Transforma tudo em minúscula.
3. Começa a construir as palavras para a inserção caractere por caractere, também aceitando palavras com hífen verificando se este está entre duas letras, quando há um espaço em branco insere a palavra construída para o dicionário.

A função de escrita chama a função toVector do dicionário para obter os dados, ordená-los e gravá-los em um arquivo. Para tornar o arquivo mais legível são utilizadas funções da biblioteca <iomanip> para formatar o conteúdo de forma organizada e visualmente clara.

6. Funcionamento do Programa

O projeto está organizado em diferentes pastas, com o objetivo de facilitar a localização e a manutenção dos arquivos:

- **includes:** Arquivos cabeçalhos (.hpp) do programa. Possui as subpastas dicionarios e estruturas que guardam os cabeçalhos correspondentes destes sem a implementação. Além disso, esta pasta também inclui os arquivos DictIO.hpp e o UnicodeComparator.hpp.
- **src:** Guarda a implementação (.tpp) dos arquivos cabeçalho que não foram implementados diretamente no include.
- **tables_texts:** Pasta para onde são direcionado os arquivos com as tabelas de frequência.
- **tests_dictionary:** Pasta que possui uma main de teste que realiza testes de inserção, busca e remoção nos dicionários, também possui arquivos .txt com as informações obtidas pós teste.
- **tests_structure:** Pasta com mains de testes individuais das estruturas de dados AVL, Rubro-Negra, Chained Hash Table e Open Hash Table.
- **texts:** Pasta onde estão armazenados os arquivos de texto para leitura com os dicionários.

A compilação do programa é feita através de um Makefile que gera um executável da main. O Makefile é compatível tanto com sistemas Windows quanto Linux. Para compilar o projeto basta estar do terminal/prompt de comando, dentro da pasta do projeto e executar um dos comandos disponíveis `make all` para criação do executável ou `make clean` que apaga o executável. Após a compilação, o executável estará disponível na pasta principal do projeto. Para executá-lo, utilize o seguinte formato no terminal:

```
./nome_do_executavel [estrutura]
[arquivo_de_entrada].txt [arquivo_de_saida].txt
```

Substitua `./nome_do_executavel` pelo nome gerado (por padrão `./freq`), e os arquivos conforme desejado. Por exemplo:

```
./freq map_avl livro.txt saida.txt
```

Onde a estrutura pode ter os seguintes valores:

- **map_avl:** dicionário com árvore AVL.
- **map_rbt:** dicionário com árvore Rubro-Negra.
- **map_cht:** dicionário com tabela hash com encadeamento externo.
- **map_oht:** dicionário com tabela hash com endereçamento aberto.

O programa já possui alguns arquivos de entrada salvos que se encontram na pasta `/texts`, caso queira testar novos arquivos, os coloque nessa pasta para evitar erros. Já o arquivo de saída pode possuir o nome que você desejar, para encontrá-los basta procurar a pasta `/tables_texts`.

7. Listagem de Testes e Resultados

Para os testes de desempenho, foi feito uma pasta com uma main de testes (`main_test`), responsável por executar as operações de inserção de palavras por um arquivo de texto, busca de uma palavra existente e de uma inexistente e a remoção de uma chave do dicionário. Cada uma dessas atividades são realizadas 100 vezes em diferentes dicionários para garantir uma média estável do tempo de execução de cada operação.

Para o primeiro teste o arquivo de texto escolhido foi a versão em inglês da Bíblia (“kjv-bible.txt“), com aproximadamente 892.132 palavras. Após o processo de normalização, o dicionário resultante contém cerca de 12.754 pares de palavra e frequência .

Tabela 1. Tabela com tempo médio de execução - “kjv-bible.txt“

Operação	AVL	RBT	CHT	OHT
Inserção	3.62	3.11	**0.73**	1.27
Busca (existe)	0.00000089	**0.00000031**	0.00000104	0.00000180
Busca (inexist.)	0.00000356	0.00000250	**0.00000049**	0.00000069
Remoção	0.00000507	0.00000189	**0.00000183**	0.00000225

Tabela 2. Tabela com as métricas das estruturas - “kjv-bible.txt“

Métrica	AVL	RBT	CHT	OHT
Comparações de chaves	41.588.147	33.207.281	**1.771.513**	**1.771.513**
Rotações	9.735	**8.209**	–	–
Recolorações	–	43.879	–	–
Colisões	–	–	**8.333**	21.083
Rehashes	–	–	11	11

No segundo teste, foi usado o livro Dom Casmurro (dom-casmurro.txt). Ele tem por volta de 68.984 palavras no total, e depois de ser inserido nos dicionários, fica com 10.119 pares de elementos.

Tabela 3. Tabela com tempo médio de execução - "dom-casmurro.txt"

Operação	AVL	RBT	CHT	OHT
Inserção	0.49285986	0.41812341	**0.10326485**	0.18368395
Busca (existe)	0.00000400	0.00000187	**0.00000165**	0.00000283
Busca (inexistente)	0.00000401	0.00000345	**0.00000045**	0.00000066
Remoção	0.00001169	0.00000548	**0.00000211**	0.00000412

Tabela 4. Tabela com as métricas das estruturas - "dom-casmurro.txt"

Métrica	AVL	RBT	CHT	OHT
Comparações de chaves	3.745.135	2.995.592	**127.852**	**127.852**
Rotações	7.403	**6.233**	–	–
Recolorações	–	34.355	–	–
Colisões	–	–	7.361	18.965
Rehashes	–	–	11	11

Já o terceiro teste foi feito com o livro Sherlock Holmes ("sherlock_holmes.txt"), que tem aproximadamente 108.376 palavras no total e 8.438 pares de palavra e frequência nos dicionários após a inserção.

Tabela 5. Tabela com tempo médio de execução - "sherlock_holmes.txt"

Operação	AVL	RBT	CHT	OHT
Inserção	0.89851190	0.93174836	**0.13375111**	0.23654798
Busca (existe)	0.00000463	0.00000575	**0.00000121**	0.00000250
Busca (inexist.)	0.00000583	0.00000587	**0.00000049**	0.00000197
Remoção	0.00001282	0.00001591	**0.00000246**	0.00000376

Tabela 6. Tabela com as métricas das estruturas - "sherlock_holmes.txt"

Métrica	AVL	RBT	CHT	OHT
Comparações de chaves	5.310.015	4.265.072	**208.317**	**208.317**
Rotações	6.117	**5.094**	–	–
Recoloracoes	–	28.283	–	–
Colisões	–	–	**5.102**	13.646
Rehashes	–	–	10	10

7.1. Análise Geral dos Resultados

Com base nos testes realizados com três obras distintas (Bíblia, Dom Casmurro e Sherlock Holmes), é possível observar padrões no desempenho das estruturas. A tabela hash com endereçamento exterior (CHT) apresentou o melhor tempo de execução médio nas operações de inserção, busca e remoção, se sobressaindo nos três testes.

Apesar de a tabela hash de endereçamento aberto (OHT) não ser a mais rápida em todos os casos, ela chega tempos de execução próximos ao da CHT, com mais colisões e mesmo número de rehashes. Além disso, tanto a CHT quanto a OHT possuem

menos comparações de chave do que as árvores (AVL e RBT), o que explica seu melhor desempenho nas operações básicas.

No caso específico das buscas por chaves existentes, as árvores AVL e RBT ocasionalmente se destacaram. Isso pode ser explicado pela ordenação e balanceamento dessas estruturas, que permitem encontrar elementos com pouca busca quando a chave está presente. No entanto, essa vantagem não é a mesma nas buscas por elementos inexistentes, já que as árvores percorrem caminhos até as folhas, resultando em mais comparações, o que não ocorre nas tabelas hash, que lidam melhor com esse tipo de operação.

7.2. Análise entre estruturas baseadas em árvores

Ao comparar as estruturas baseadas em árvores, a Rubro-Negra possui um desempenho ligeiramente superior ao da AVL, tanto nas operações gerais quanto nas métricas de comparações de chaves e rotações realizadas. Isso se deve ao critério de balanceamento, que é baseado na coloração dos nós em vez do balanceamento baseado na altura das subárvores, como é o caso da AVL. Por depender da verificação das cores de seus nós e não da comparação de altura, a Rubro-Negra consegue ter um pouco mais de eficiência do que a AVL, já que não precisar realizar tantas rotações em seu conserto o que o torna menos custoso.

7.3. Análise entre estruturas baseadas em tabelas hash

Já na comparação das estruturas baseadas em Hash, a tabela com encadeamento exterior (CHT) demonstrou melhor desempenho em relação a tabela com endereçamento aberto (OHT). Isso se deve principalmente a forma como as colisões são tratadas. Enquanto a CHT resolve colisões por meio de listas encadeadas, a OHT utiliza sondagem quadrática, o que leva a mais acessos e maior número de colisões consecutivas.

Além disso, a CHT apresentou menos colisões e tempos médios mais baixos em todas as operações analisadas, como inserção, busca e remoção. A OHT, apesar de simples e eficiente em cenários com baixa taxa de ocupação, tem seu desempenho piorado a medida que a tabela se aproxima do fator de carga, exigindo rehashes mais frequentes e tornando o gerenciamento de colisões mais custoso.

8. Especificações

- **Processador:** Intel Core i5-1235U (12ª geração), 1.30 GHz, 10 núcleos, 12 threads
- **Memória RAM:** 8 GB
- **Armazenamento:** SSD de 512 GB
- **Arquitetura:** x86_64 (64 bits)
- **Sistema Operacional:** Windows 11 Home Single Language (64 bits)
- **Compilador:** g++ (MinGW) versão 15.1.0
- **Padrão da linguagem:** C++20

9. Conclusão

Conclusão Este projeto proporcionou uma análise aprofundada das estruturas de dados abordadas em sala de aula, permitindo avaliar suas vantagens e desvantagens na prática. A implementação de cada estrutura não só revelou seu funcionamento individual, mas

também permitiu uma percepção do todo como as diferenças entre elas. Com o uso de métricas e contadores de tempo, as diferenças de desempenho se tornaram bem mais perceptíveis, reforçando o entendimento de que cada estrutura possui seus pontos fortes e fracos, e que a escolha da mais adequada depende diretamente do contexto de sua implementação.

Ao longo do desenvolvimento, algumas dificuldades foram encontradas. Dentre elas, destaco a utilização da biblioteca ICU, tanto para sua configuração no Windows quanto para sua integração no código. O fato de suas comparações serem realizadas bit a bit exigiu a modificação de todas as estruturas de dados, que originalmente utilizavam os comparadores padrão do C++. Isso conduz para outro desafio no desenvolvimento que foi a complexidade de implementar o objeto Collator para realizar as comparações de forma mais abstrata e linguisticamente correta.

Em resumo, este projeto consolidou o conhecimento teórico em Estruturas de Dados, transformando-o em experiência prática. As dificuldades encontradas, especialmente com a manipulação de strings via biblioteca ICU e a necessidade de ajustar os comparadores, foram obstáculos importantes que aprofundaram a compreensão sobre manutenção do código. A análise de desempenho e comparação entre métricas mostrou a importância de escolher uma estrutura de dados sabendo de seus pontos fortes e fracos para uma boa otimização em determinados projetos.

Referências

[1] Thomas H. Cormen. *Algoritmos - Teoria e Prática*. GEN LTC, 2012.

[2] cplusplus.com. *std::hash - C++ Reference*. 2024. URL: https://cplusplus.com/reference/functional/hash/.

[3] cplusplus.com. *std::iomanip - C++ Reference*. 2024. URL: https://cplusplus.com/reference/iomanip/?kw=iomanip.

[4] cplusplus.com. *std::map - C++ Reference*. 2024. URL: https://cplusplus.com/reference/map/map/.

[5] cplusplus.com. *std::pair - C++ Reference*. 2024. URL: https://cplusplus.com/reference/utility/pair/.

[6] cplusplus.com. *std::unordered_map - C++ Reference*. 2024. URL: https://cplusplus.com/reference/unordered_map/unordered_map/.

[7] Jayme Luiz Szwarcfiter. *Estrutura de Dados e Seus Algoritmos*. GEN LTC, 2010.