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

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.

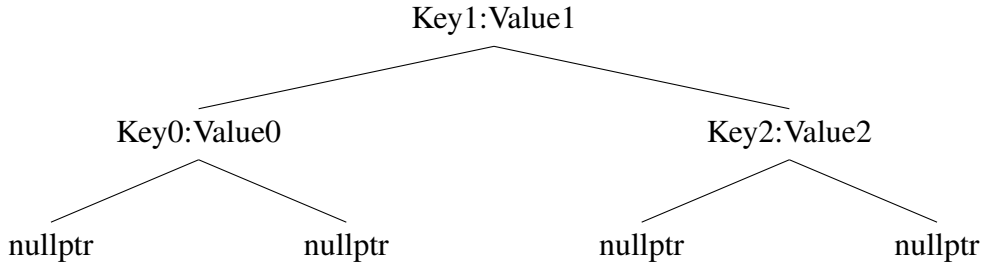

Figura 1. Diagrama ilustrativo da AVL

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.

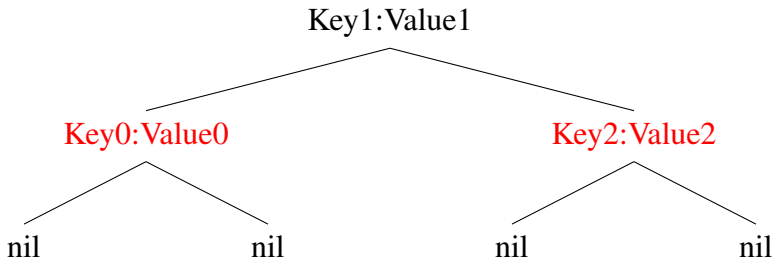

Figura 2. Diagrama ilustrativo da árvore Rubro-Negra

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).

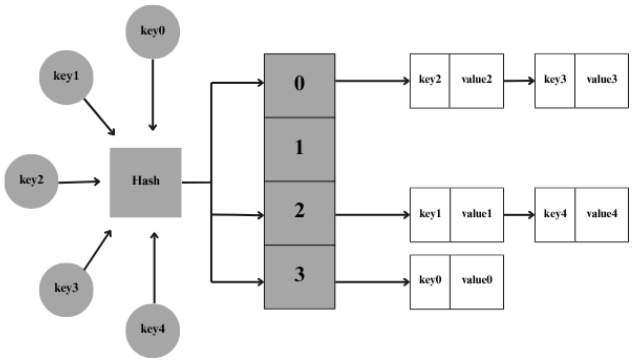

Figura 3. Diagrama ilustrativo da hash com encadeamento exterior

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.

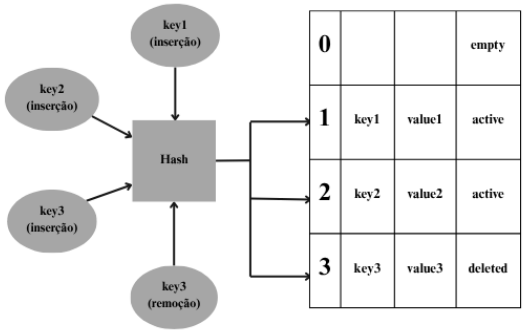

Figura 4. Diagrama ilustrativo da hash com endereçamento aberto

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. 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 6.3.0
- **Padrão da linguagem:** C++11 e C++14

5. Conclusão

Por fim, essa primeira parte do projeto, dedicada exclusivamente a implementação das estruturas de dados que serão futuramente utilizadas na construção de dicionários genéricos, consolidou os conteúdos teóricos vistos em sala e permitiu observar na prática as principais diferenças entre as abordagens. A utilização de métricas de contagem também contribuiu para uma análise comparativa do desempenho entre as estruturas. Esse processo serviu como uma base para a próxima etapa do trabalho, onde essas implementações serão integradas a uma aplicação concreta, permitindo explorar suas vantagens e desvantagens.

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::map - C++ Reference*. 2024. URL: https://cplusplus.com/reference/map/map/.

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

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