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

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. 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 iniciais do que serão os dicionários do projeto em si. 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 que será discutido mais a frente.

Abaixo, está o diagrama de classes do projeto:

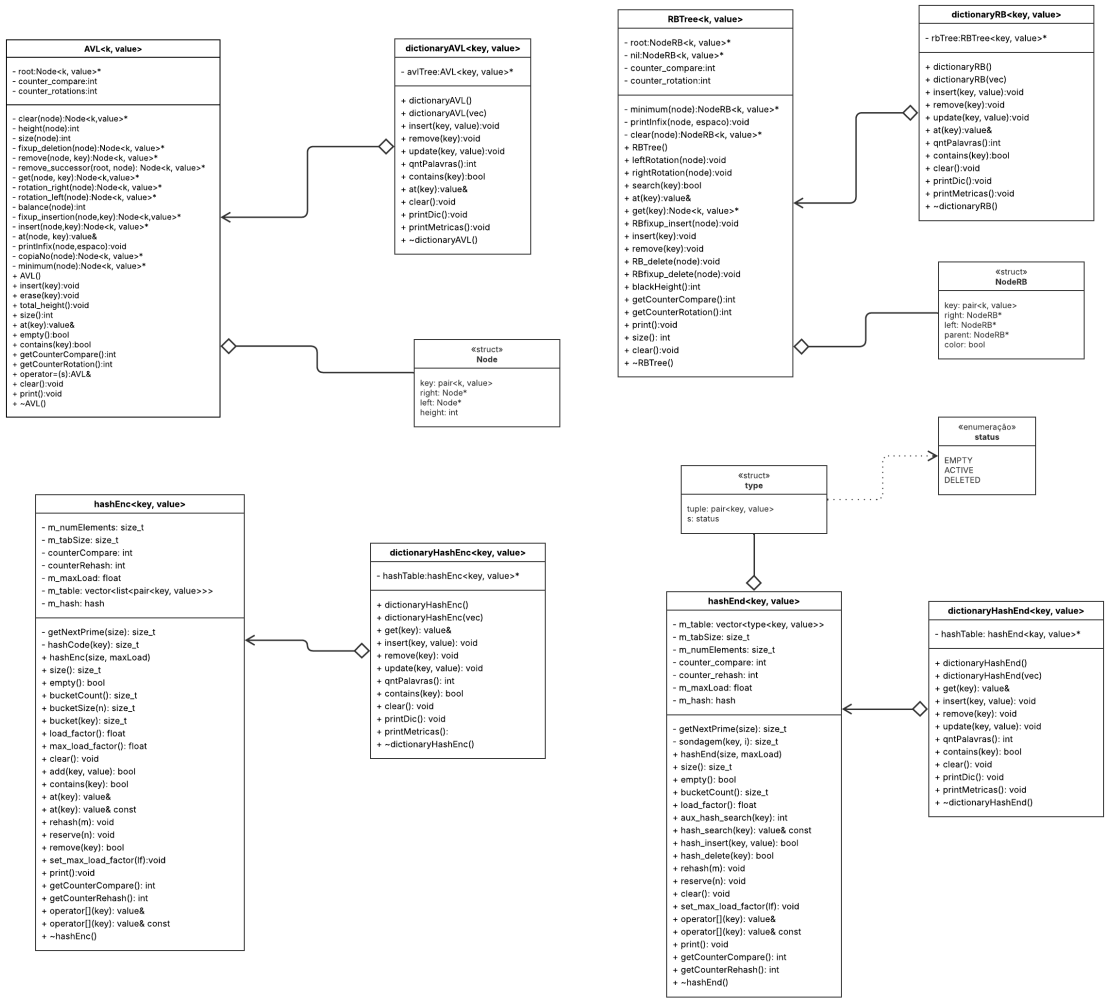

Figura 1. Diagrama de classes do projeto

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

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, são aplicadas rotações simples ou duplas (esquerda-direita ou direita-esquerda), conforme necessário.

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.

1.3. 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.

1.4. 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.

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. 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

4. Dificuldades encontradas

Nessa primeira parte do trabalho, as principais dificuldades estiveram relacionadas 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.

Além disso, a adição dos contadores de comparações de chave também foi algo um pouco complicado de estabelecer dentro do projeto, pois, 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).

Por fim, 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 por 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. Arquivo de teste

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

1. Criação de um dicionário vazio
2. Criação de um dicionário a partir de um vetor de pares de chave e valor

3. Inserção de elementos
4. Remoção de elementos
5. Atualização de um valor associado a uma chave
6. Acesso de um valor associado a uma chave
7. Verificação de existência de chave
8. Mostrar quantidade de palavras
9. Mostrar o resultado das métricas
10. Mostrar dicionário
11. Limpeza do dicionário

6. 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. Cada estrutura apresentou vantagens e desafios específicos.

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.

Essa primeira etapa do projeto proporcionou um aprendizado sólido sobre as diferentes abordagens de implementação de dicionários e servirá como base para análises futuras mais aprofundadas, especialmente quando for realizada a leitura com tratamento de strings e contagem real das palavras em arquivos de texto, objetivo final da segunda parte do trabalho.

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.