Relatório do projeto de contador de frequências utilizando Estruturas de Dados Avançadas

Ana Beatriz Martins Santiago[1]

[1]Aluna da Universidade Federal do Ceará (UFC), Campus Quixadá
Quixadá – CE – Brazil

anabsantiago0@gmail.com

Abstract. *This report describes the implementation of a frequency counter program, discussing the observed issues and the program's development process, including the data structures implemented and used, the main functions and their purposes, the tests performed, and more. Furthermore, it also addresses how to use the developed program, detailing data input and output. Moreover, a brief analysis of the performance of each data structure used is found here, along with a comparison between them, discussing how they performed when applying certain metrics, and their results.*

Resumo. *Este relatório descreve a implementação de um programa contador de frequências, discorrendo a respeito da problemática observada e do processo de construção do programa, tal como quais estruturas de dados foram implementadas e utilizadas, as principais funções e seus propósitos, os testes realizados, entre outros. Além disso, aborda-se também a respeito de como utilizar o programa desenvolvido, detalhando entrada e saída de dados. Ademais, uma breve análise a respeito do desempenho de cada estrutura de dados utilizada é encontrada aqui, juntamente da comparação entre as mesmas, discorrendo sobre como se comportaram ao aplicar determinadas métricas, e seus resultados.*

1. Introdução

A necessidade de um programa capaz de contar as ocorrências de palavras em um texto de tamanho considerável foi observada e apresentada como problema, cuja solução explorada implica desenvolver um dicionário, que são estruturas de dados que armazenam pares chave-valor, na qual cada chave é única e, diferentemente dos vetores comuns onde valores são acessados a partir do índice, aqui acessamos os valores buscando pela chave associada à esses. As vantagens dos dicionários são diversas, por exemplo, a versatilidade para os tipos de dados das chaves, podendo ser strings, objetos, inteiros, etc. Além disso, com buscas, inserções e remoções rápidas e eficientes, com complexidades média logaritmíca ou até mesmo constante, os dicionários possuêm aplicações reais e importantes, tais como bancos de dados, caches, DNS, jogos, etc. Visto isso, foram implementados 4 dicionários, cada um usando uma estrutura de dados diferente como base, que serão detalhadas posteriormente neste artigo, juntamente da comparação dos seus desempenhos.

2. Implementação

Esta seção descreve a implementação dos dicionários propostos, detalhando as estruturas de dados utilizadas e as medidas escolhidas para cada uma delas, a fim de analizar e

comparar seus potenciais desempenhos. Foram escolhidas duas árvores binárias de busca, sendo elas a árvore AVL e a Rubro-Negra, e duas tabelas de dispersão (tabela hash), uma com tratamento de colisão por encadeamento exterior (chained hash table), e a outra com tratamento de colisão por endereçamento aberto (open addressing hash table).

Vale ressaltar que cada estrutura recebe um par de chave-valor genéricos, e também que foram escolhidas duas métricas para cada estrutura, sendo que uma foi aplicada para todas elas: o número de comparações de chaves necessárias para construir a tabela de frequências (saída final do programa). A segunda métrica escolhida para as árvores foi contabilizar o número de rotações realizadas, enquanto que para as tabelas foi decidido calcular o fator de carga médio, que se dá pelo somatório dos fatores de carga de cada inserção dividido pelo número de elementos da tabela. Enfim, segue os detalhes de cada estrutura.

2.1. Árvores

As árvores escolhidas buscam, em suma, prover um balanceamento dos nós de modo que uma altura muito grande seja evitada, uma vez que sua complexidade de tempo das operações (busca, inserção e remoção) é diretamente relacionada com sua altura, sendo essa $O(log\ n)$ no pior caso, para n igual ao número de nós. Essa eficácia é prometida devido ao balanceamento próprio dessas árvores, o qual descrevo singularmente a seguir:

A árvore AVL possui um balanceamento mais rígido se comparado à Rubro-Negra, pois exige que o balanço de cada nó seja 1, 0 ou -1. Por balanço de um nó x qualquer, entendemos que se deve ao cálculo da diferença entre a altura da árvore direita do nó x e da altura da árvore esquerda de x. Logo, quando x possui um balanço igual a 0, o número de nós à esquerda é igual ao da direita. Sendo o balanço negativo, então o peso da árvore enraizada em x possui peso (mais nós) em sua porção esquerda, sendo o contrário quando o balanceamento for positivo. Os nós da AVL possuem os seguintes atributos: filho direito, filho esquerdo, chave e altura. Além disso, as AVL's possuem quatro tipos de rotações: rotação simples à esquerda, rotação simples à direita, rotação dupla à esquerda e rotação dupla à direita.

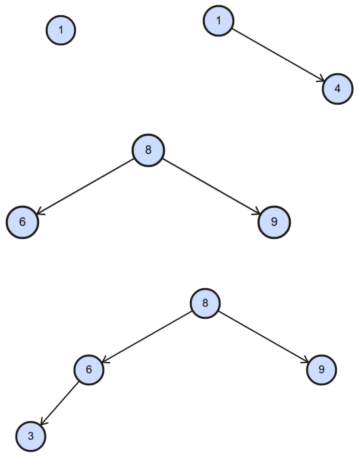

Figure 1. Exemplo de árvores AVL

A árvore Rubro-Negra por sua vez, permite uma diferença de altura maior que a

AVL, tendo seu balanceamento baseado nas cores dos nós da árvore. De modo geral, essa árvore tem 5 propriedades: todo nó é vermelho ou preto, a raiz da árvore é preta, toda folha (aqui chamada de nó NIL) é preta, se um nó é vermelho então seus filhos devem ser pretos e, para todo nó, todos os caminhos dele até as folhas devem ter o mesmo número de nós pretos. Além disso, cada nó possui os seguintes atributos: pai, filho esquerdo, filho direito, chave e cor. O atributo pai ajuda nos rebalanceamento e colorações da árvore, sendo que o nó raiz possui como pai o nó NIL. Esse tipo de árvore possui apenas dois tipos de rotação, sendo essas a rotação simples à esquerda e a rotação simples à direita.

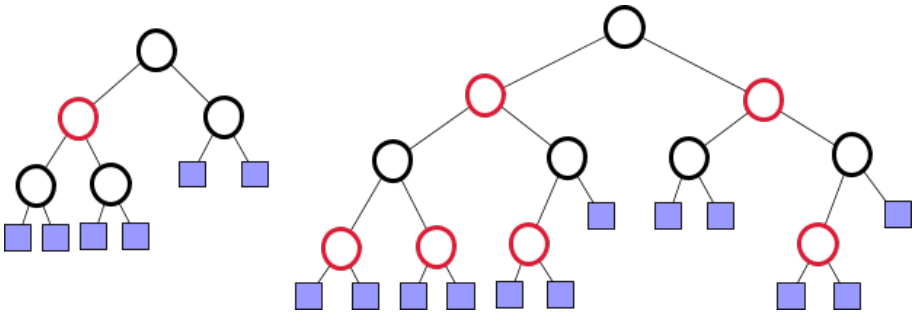

Figure 2. Exemplo de árvores Rubro-Negras

Apesar das diferenças, as duas árvores possuem mesma complexidade de pior caso, sendo essa $O(log\ n)$, como já dito. Entretanto, por ser mais restrita, a AVL acaba sendo uma árvore mais compacta, com no máximo a diferença de alturas igual a 1. Logo, suas buscas são mais rápidas se comparadas à Rubro-Negra, que por sua vez tem uma estrutura mais liberal no quesito altura, mas devido a isso, suas inserções e remoções são mais rápidas que na AVL, além de realizar menos rotações. Em suma, a árvore AVL lida bem com buscas, além de ter um balanceamento equilibrado e constante, porém possui operações de inserção e remoção mais complicadas, onde mais rotações são executadas para manter o equilíbrio restrito, carecendo de bastante processamento. A Rubro-Negra então, também possui um balanceamento equilibrado, havendo dois métodos de rebalanceamento (coloração e rotação), executando menos rotações ao inserir e remover, mas a coloração pode acabar sendo vista como um balanceamento preguiçoso, além de acabar com um desempenho mais fraco que a AVL ao fazer buscas.

2.2. Tabelas

As tabelas hash escolhidas, em suma, visam prover um acesso eficiente aos dados por meio da função de espalhamento, sendo sua principal vantagem o tempo médio constante para as operações de inserção, busca e remoção, ou seja, $O(1)$. Porém, o comportamento real dessas estruturas depende do tratamento dado às colisões, isto é, quando duas ou mais chaves distintas são mapeadas para a mesma posição da tabela. Os dois métodos analisados a seguir (endereçamento aberto com hashing duplo e encadeamento com listas ligadas (chaining)) tratam essas colisões de formas distintas, cada uma com vantagens e desvantagens características.

No endereçamento aberto com hashing duplo, ao ocorrer uma colisão, a tabela não armazena múltiplos elementos por posição, mas sim procura por outra posição livre utilizando uma segunda função hash. Essa função é aplicada repetidamente em conjunto com a função hash primária, conforme a seguinte fórmula geral, onde k é a chave, i é o

número da tentativa, $hash1$ e $hash2$ são funções de hash distintas e m é o número de slots da tabela:

```
hash(k, i) = (hash1(k) + i * hash2(k)) mod m
```

A escolha pela função hash2 exige que seu retorno nunca seja 0 e que seja primo em relação a m, dessa forma a evitar ciclos e garantir que toda a tabela possa ser explorada. Essa forma de tratamento de colisão exige que a tabela esteja parcialmente vazia, geralmente com fator de carga máximo menor que 0.7 (na implementação foi utilizado 0.5), para manter o desempenho eficiente. Quando bem dimensionada, a busca é rápida e o acesso direto à memória favorece a performance. Entretanto, deletar elementos exige tratamento especial (com marcadores de exclusão), e o desempenho diminui rapidamente com o aumento da ocupação.

Por outro lado, o encadeamento com listas ligadas (chaining) lida com colisões por meio da alocação dinâmica de listas em cada posição da tabela. Assim, várias chaves podem coexistir numa mesma posição, ligadas por uma lista. Este método permite que o fator de carga ultrapasse 1 sem muitos problemas (foi utilizado 0.75 na implementação), pois não há limite estrito de elementos por posição. A estrutura é mais flexível, e a remoção de elementos é mais direta. Mas o desempenho de busca depende do comprimento das listas, que tende a crescer se a função hash não distribuir bem os elementos.

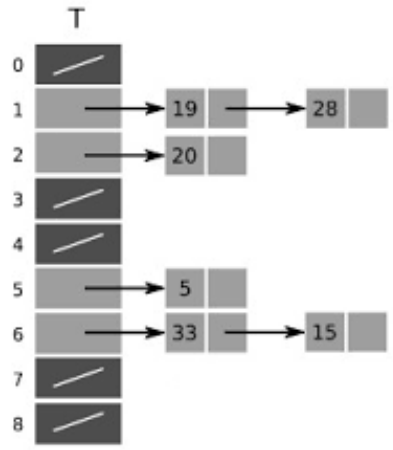

Figure 3. Exemplo de tabela hash com chaining

Em suma, apesar de suas diferenças, ambas buscam eficiência na média $O(1)$, mas divergem no tratamento das colisões e no comportamento sob altas cargas. O método de hashing duplo oferece melhor uso da memória e evita alocação dinâmica, além de ser mais eficiente em buscas, sendo indicado quando o espaço é limitado e a tabela pode ser redimensionada para manter-se esparsa, porém carece de atenção na escolha das funções de hash e no gerenciamento da exclusão. Já o encadeamento com listas é mais tolerante a sobrecarga, mais robusto, mais fácil de implementar, e facilita operações como remoção e inserção, embora possa sofrer mais com listas longas se a função hash for mal escolhida.

2.3. Arquivo main, utilização do programa e demais detalhes

Todas as estruturas de dados descritas tem como principais funções públicas: criação, inserção, remoção, verificação de existência, verificação de estado vazio, número de

comparações realizadas e sobrescrita do operador de indexação. Cabe ressaltar que nessa última, caso um elemento não existente seja buscado (pelo index), ele será inserido. Ademais, as árvores possuem uma função extra que retorna o número de rotações ocorridas, enquanto as tabelas também tem uma função extra que retorna o fator de carga médio.

O arquivo main do programa recebe dois parâmetros de linha de comando, um indicando qual o dicionário escolhido (AVL, Rubro-Negra, Chained Hash Table ou Open Addressing Hash Table) e o outro com o nome do arquivo .txt escolhido para contar as frequências. O padrão utilizado foi para essas entradas foi:

Estrutura	Exemplo de chamada
AVL	./main AVL texto.txt
Rubro-Negra	./main RB texto.txt
Chained Hash Table	./main HashC texto.txt
Open Addressing Hash Table	./main HashO texto.txt

Figure 4. Exemplo de chamadas de cada dicionário

Quanto a compilação do programa, é importante que sejam utilizadas as flags -licuuc e -licuio (g++ main.cpp -licuuc -licuio -o main, por exemplo), pois foi utilizada a biblioteca ICU, para que Strings Unicode sejam processadas. Isso é, o programa consegue ler caracteres acentuados, que era a principal intenção, mas as Strings Unicode são bem mais amplas, incluindo símbolos diversos, mas que não precisaram ser explorados aqui, visto que o foco são caracteres alfabéticos (e o hífen, muitas vezes incluído em palavras do nosso português). Ademais, o resultado de cada dicionário é registrado na pasta “saida”, em um arquivo .txt, nomeado como “resultado” seguido do nome do dicionário escolhido. Dessa forma, é possível comparar mais facilmente os resultados finais. No começo de cada arquivo de resultado, há o cabeçalho com “Tabela de frequências”, seguido na linha abaixo com “Chave : Frequencia”, indicando o modelo que as chaves e valores estão dispostos nesses arquivos. Já no final de cada arquivo de resultado, há um texto dizendo quanto tempo foi necessário para criar a tabela, além de mostrar também as contagens das métricas, isso é, nos resultados das árvores, há registrado o número de rotações e o número de comparações realizadas, enquanto nos resultados das tabelas, haverá valor do fator de carga médio no lugar do número de rotações.

Quanto às entradas, há uma pasta nomeada “entrada”, onde o arquivo .txt é buscado primeiramente nela e, se o mesmo não for encontrado nessa pasta, ele será então buscado da maneira literal como o usuário escreveu na linha de comando, logo, o usuário pode criar sua própria entrada .txt e incluí-la na pasta de entradas (e digitar apenas o nome do arquivo .txt na passagem de parâmetros), ou passar corretamente o caminho até

o arquivo desejado.

3. Testes executados e resultados

Diversos testes foram executados ao longo do desenvolvimento do projeto, dentre os principais cito aqui os da primeira fase, onde o foco era preparar e deixar as estruturas de dados funcionais e prontas para uso. Nessa etapa, foram criados arquivos de testes para cada estrutura, que testavam suas principais funções, das quais menciono criação, inserção, remoção, verificação de existência, verificação de estado vazio, operador colchetes, quantidades de rotações (caso das árvores), fator de carga médio (caso das tabelas) e quantidade de comparações operadas.

As entradas desses arquivos testes eram as mesmas para todas as estruturas, para fins de comparações. Os resultados observados nessa primeira parte ainda não contavam com a contagem de tempo, porém foi notável uma quantidade menor de rotações e de comparações na árvore Rubro-Negra, se comparada à AVL. Quanto às tabelas, a Chained Hash Table apresentou um número de comparações menor, enquanto a Open Addressing Hash Table mostrou um fator de carga médio menor que a primeira, além de finalizar com um tamanho maior da tabela.

Na fase 2 e final do projeto, as estruturas foram utilizadas pela main em prol de construir a tabela de frequências tão esperada, então os testes aqui foram executados utilizando arquivos .txt. Cabe ressaltar que na construção da tabela, as funções mais utilizadas são o operador colchetes, a inserção e a busca. Logo, as estruturas com mais versatilidade em inserções e buscas obtiveram melhores resultados.

Abaixo, segue uma tabela com o desempenho observado de cada estrutura para a entrada sendo a Bíblia do Rei Jaime na sua versão em inglês, com aproximadamente 770.440 palavras.

A	B	C	D	E
	AVL	Rubro-Negra	C. Hash Table	O. A. Hash Table
Número de comparações	158841	159955	3054	10718
Tempo de construção (milissegundos)	2028	1746	815	1321
Número de rotações	9640	8114	-	-
Fator de carga médio	-	-	0,24165	0,57572

Figure 5. Desempenho a partir da leitura da Bíblia

Por outro lado, abaixo apresento a mesma tabela mas com a entrada sendo o manifesto comunista, também em sua versão em inglês, sendo um livro menor, possuindo em torno de 13.000 palavras.

Ademais, o formato de saída dos dados no arquivo de resultado foi explorado de maneira a ficar mais visualmente agradável, com espaçamentos e linhas (separadores), porém isso aumentou um tanto que consideravelmente o tempo de execução. Portanto, foi decidido imprimir de maneira simples, compreensível e rápida, seguindo o modelo de

A	B	C	D	E
	AVL	Rubro-Negra	C. Hash Table	O. A. Hash Table
Número de comparações	27038	27320	786	2052
Tempo de construção (milissegundos)	62	53	42	78
Número de rotações	1934	1588	-	-
Fator de carga médio	-	-	0,28746	0,49919

Figure 6. Desempenho a partir da leitura do Manifesto comunista

“Chave : Frequência”, já que o propósito principal do projeto é comparar os desempenhos e métricas estabelecidas das estruturas.

4. Conclusão

Esta seção aborda as conclusões tiradas a partir dos dados apresentados no capítulo anterior e das noções das vantagens e desvantagens de cada estrutura.

Inicialmente, é perceptível o ótimo desempenho da Chained Hash Table em comparação às demais estruturas, sendo ela a mais rápida e com menor número de comparações em ambos os testes apresentados. Porém, apesar da Open Addressing Hash Table ter apresentado um número maior de comparações, ela terminou com um fator de carga médio superior ao da outra tabela. Isso ocorre porque, ao contrário da estrutura encadeada que armazena múltiplos elementos em listas na mesma posição da tabela, a tabela com endereçamento aberto precisa de mais espaço livre para funcionar eficientemente, e mesmo assim os elementos acabam ocupando posições distintas. Isso resulta em uma maior utilização do espaço disponível. Logo, embora o desempenho da Chained Hash Table tenha se destacado nos testes, a Open Addressing Hash Table demonstrou uma melhor ocupação da tabela como um todo, o que pode ser vantajoso em contextos onde o uso de memória compacta é desejado.

Já dentre as árvores, a Rubro-Negra também possuiu um desempenho agradável, sendo mais rápida e fazendo menos rotações que a árvore AVL, que por sua vez, conseguiu um número menor de comparações, provavelmente devido à sua vantagem em buscas.

Com isso, fica evidente que a escolha da estrutura de dados ideal depende diretamente das prioridades e restrições do cenário de uso. A tabela com listas encadeada se destaca em performance bruta quando há muitas inserções e buscas, mas em situações onde o uso eficiente da memória é um fator crítico, a tabela com endereçamento aberto se torna uma opção interessante, mesmo com um custo maior em comparações. Já no caso das árvores, a Rubro-Negra equilibra bem desempenho e complexidade, sendo vantajosa em aplicações que exigem um comportamento mais estável, enquanto a AVL pode ser preferida em casos onde a eficiência de busca é prioritária. Por fim, o projeto permitiu não só uma análise comparativa prática entre essas estruturas, como também reforçou a importância de considerar as características internas de cada uma para tomar decisões informadas no desenvolvimento de sistemas eficientes.

5. Relatório em Latex

Este relatório foi escrito utilizando Latex, no Overleaf. Segue o link para o projeto: https://pt.overleaf.com/read/rpxnrtzchssm#617891

References

Bauer, R. (2017). Algoritmo concreta: Árvore rubro-negra. https://algoritmoconcreta.blogspot.com/2014/05/arvore-rubro-negra.html.

Cormen, T. H. (2012). *Algoritmos: teoria e prática.* Elsevier.

Doug, T. (2011). What is the advantage of using map data structure? https://softwareengineering.stackexchange.com/questions/117513/what-is-the-advantage-of-using-map-datastructure.

GeeksforGeeks (2024). Introduction to map data structure. https://www.geeksforgeeks.org/dsa/introduction-to-map-data-structure/.

Pessoa, G. (2025). Por dentro da estrutura de dados dicionário (hash map) - dev community. https://dev.to/gabrieljbpessoa_96/por-dentro-da-estrutura-de-dados-dicionario-hash-map-2kl0#:~:text=Em%20programa%C3%A7%C3%A3o%2C%20um%20dicion%C3%A1rio%20(tamb%C3%A9m,em%20vez%20de%20%C3%ADndices%20num%C3%A9ricos.

Villalta, H. (2023). Red black — avl tree — tree algorithms — medium. https://medium.com/@humberto521336/red-black-avl-trees-overview-d2f1af7886eee.

Wikipedia (2025). Hash table. https://en.wikipedia.org/wiki/Hash_table.