Relatório de ED2

Autor: Eduardo Figueredo Pacheco

Data
19/05/2023

Sumário

1 Introdução

Este relatório tem como objetivo apresentar uma análise detalhada de diferentes estruturas de dados utilizadas na construção de tabelas de símbolos. Neste contexto, exploraremos as estruturas ABB (Árvore Binária de Busca), A23 (Árvore 2-3), ARN (Árvore Rubro-Negra), Vetor Ordenado e Treaps. Nosso foco principal será fornecer uma explicação detalhada do código implementado para cada estrutura, além de comparar seus tempos de execução por meio de tabelas e gráficos.

Cada uma dessas estruturas possui características distintas, apresentando vantagens e desvantagens em diferentes cenários de uso. Compreender o funcionamento interno de cada estrutura e analisar o desempenho delas em situações diversas nos permitirá tomar decisões informadas sobre qual estrutura escolher para determinadas aplicações.

Além disso, examinaremos as operações fundamentais de cada estrutura, como inserção e busca, com o intuito de avaliar a complexidade computacional de cada uma delas. Por meio de experimentos e medições, coletaremos dados relevantes para construir uma análise comparativa confiável, permitindo-nos visualizar como as estruturas se comportam sob diferentes cargas de trabalho e quantificar a eficiência de cada uma.

Ao final deste relatório, esperamos fornecer insights valiosos sobre as características e o desempenho associados a cada uma dessas estruturas de dados. Essas informações serão úteis para orientar a escolha da estrutura mais adequada a um determinado contexto, considerando fatores como a quantidade e o tipo de dados a serem armazenados, a frequência e o tipo de operações a serem executadas, bem como as restrições de tempo e espaço impostas pelo problema em questão.

2 Desenvolvimento

O código fornecido parece ser parte de um programa escrito em C++ que realiza operações em diferentes estruturas de dados, como árvores binárias de busca (ABB), árvores rubro-negras (ARB), árvores 2-3 (A23), árvores Treap (TR) e Vetores Ordenados (VO). O programa recebe um arquivo como entrada e realiza operações com as palavras contidas nele. Cada linha é dividida em palavras, que são armazenadas em um vetor chamado "palavras". Em seguida, o programa itera sobre as palavras e realiza várias operações com elas. Para cada palavra, são calculados diferentes atributos, como tamanho, número de vogais, número de vogais repetidas e número de repetições de caracteres.

Dependendo do valor de "E"(passado como argumento), a palavra e seus atributos são inseridos em uma das estruturas de dados disponíveis: A23, ABB, ARB, TR ou VO. A escolha da estrutura de dados é feita usando estruturas condicionais (if-else).

Após processar todas as palavras do arquivo, o programa continua com a exibição dos resultados. O comportamento do programa difere dependendo do valor de "E". Para cada valor possível de "E", são realizadas operações específicas, como imprimir palavras com a maior frequência, buscar ocorrências de uma palavra, imprimir palavras com o maior tamanho, calcular e imprimir valores SR (tamanho da maior palavra com pelo menos uma vogal repetida), ou calcular e imprimir valores VD (tamanho da maior palavra com mais vogais do que vogais repetidas).

A partir desse ponto, o código continua com o tratamento de cada caso específico, realizando as operações correspondentes em cada estrutura de dados.
Para compilar esses códigos no terminal pode ser usado o odigo em bash abaixo:

```
1 g++ -o EP -g EP.cpp ABB.cpp A23.cpp VO.cpp
     ARB.cpp TR.cpp
2 ./EP
3 E          // Estrutura a ser testada
4 N          // Numero de palavras
5 {texto}    // Texto qualquer com N palavras
6 Q          // Numero de consultas a serem feitas
7 {Q consultas...}
```

É importante salientar que o trabalho em questão depende de entradas específicas

2.1 Testes

Os testes foram realizados utilizando um bash script com o objetivo de facilitar e acelerar a análise. O script foi desenvolvido para executar os testes de forma automatizada, garantindo a consistência e reprodutibilidade dos resultados. Os testes foram realizados em tempos e momentos diferentes e em textos diferentes e em maquinas (computadores) diferentes, com o mesmo código, por isso variações podem aparecer nas tabelas, mas o mais importante é que as análises e as relações de proporcionalidade entre os testes coincidem com o esperado.

Cada teste foi repetido 10 vezes para obter uma média mais precisa e reduzir o impacto de variações temporárias no desempenho do sistema. Todos os valores apresentados nas tabelas e gráficos são referentes às médias das 10 execuções. Os testes estão manipulados em segundos; ex: 0.06040 = 0.0604 segundos.

2.1.1 Teste F

O teste F foi projetado para analisar o desempenho da funcionalidade de busca das palavras mais frequentes no texto. O objetivo era identificar as palavras que ocorrem com maior frequência em todo o texto, exigindo a análise completa de todas as partes.

Tabela 1: Tabela F

	Texto de 10000	Texto de 100000	Texto com 1000000
VO F	0.06220	1.32210	3.38450
ABB F	0.03710	0.16370	0.22750
TR F	0.03680	0.10450	0.27690
A23 F	0.03720	0.11770	0.25460
ARB F	0.03590	0.10850	0.23300

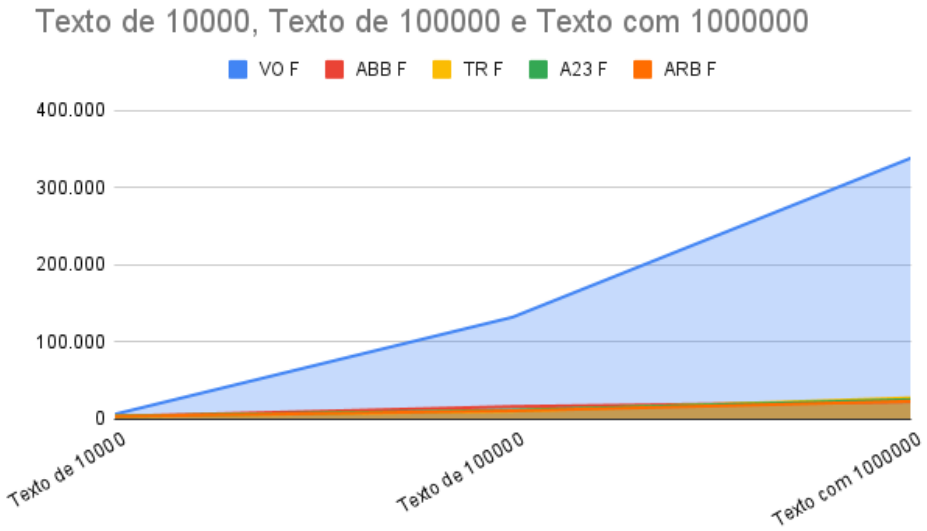

Figura 1: Gráfico Utilizando "F"

2.1.2 Teste L

Quanto ao teste L, ele se concentra em encontrar as maiores palavras na tabela de símbolos, exigindo a análise de todas as palavras.

A função tamanhoMaiorPalavraSR é responsável por encontrar o tamanho da maior palavra na estrutura de dados. Ela percorre a árvore, ou o vetor, de forma recursiva, verificando se cada nó atende aos critérios estabelecidos. No caso do Teste L, o parâmetro x é igual a -1 e o número de repetições do nó indifere para o resultado da função.A função compara o tamanho do nó atual com o tamanho da maior palavra encontrada até o momento e atualiza o valor caso o tamanho atual seja maior.

Também é chamada a função que imprime somente as palavras de tamanho máximo.

Tabela 2: Tabela L

	Texto de 10000	Texto de 100000	Texto com 1000000
VO L	0.06040	1.28560	3.40560
ABB L	0.03790	0.16470	0.29640
TR L	0.03680	0.10850	0.24120
A23 L	0.03640	0.09550	0.18240
ARB L	0.03570	0.13500	0.17430

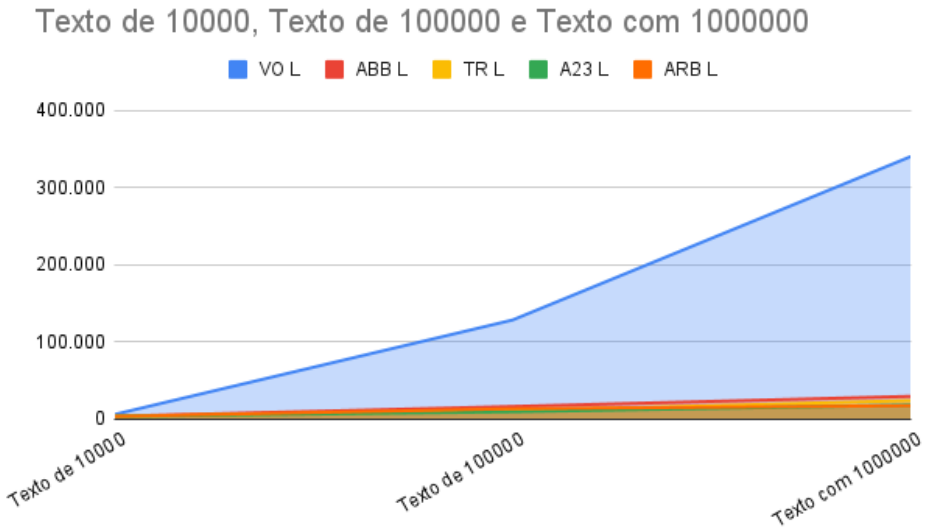

Figura 2: Gráfico Utilizando "L"

2.1.3 Teste SR

O teste SR visa identificar as maiores palavras na tabela de símbolos que não contenham letras repetidas. Assim como nos outros testes, é necessário analisar todas as palavras para determinar quais delas atendem a esse critério específico. Utilizando o bash script desenvolvido para esse propósito, o teste avalia a eficácia da funcionalidade de identificação das maiores palavras sem letras repetidas, registrando o tempo de execução e fornecendo a lista das palavras identificadas.

A função tamanhoMaiorPalavraSR é responsável por encontrar o tamanho da maior palavra na estrutura de dados que não possui letras repetidas. Ela percorre a árvore, ou o vetor, de forma recursiva, verificando se cada nó atende aos critérios estabelecidos. Se o parâmetro x for diferente de -1 e o número de repetições do nó for diferente de x, e o nó não tiver filhos (subárvores esquerda e direita), a função retorna 0. Caso contrário, ela compara o tamanho do nó atual com o tamanho da maior palavra encontrada até o momento e atualiza o valor caso o tamanho atual seja maior.

Quando x for igual a -1 é que essa função também é utilizada para o teste L, que busca a maior palavra sem nenhuma outra restrição, por isso existe essa parte da função, mas ela não interfere, pois ambos os casos são separados.

Já a função imprimePalavrasSR é responsável por imprimir as palavras na estrutura que possuem um determinado tamanho (TAM) e um número específico de repetições (X) [se X == -1 não irá interferir].

Tabela 3: Tabela SR

	Texto de 10000	Texto de 100000	Texto com 1000000
VO SR	0.06110	1.38220	3.40880
ABB SR	0.03830	0.12040	0.19500
TR SR	0.03720	0.15940	0.19040
A23 SR	0.03680	0.09630	0.16720
ARB SR	0.03630	0.13510	0.15450

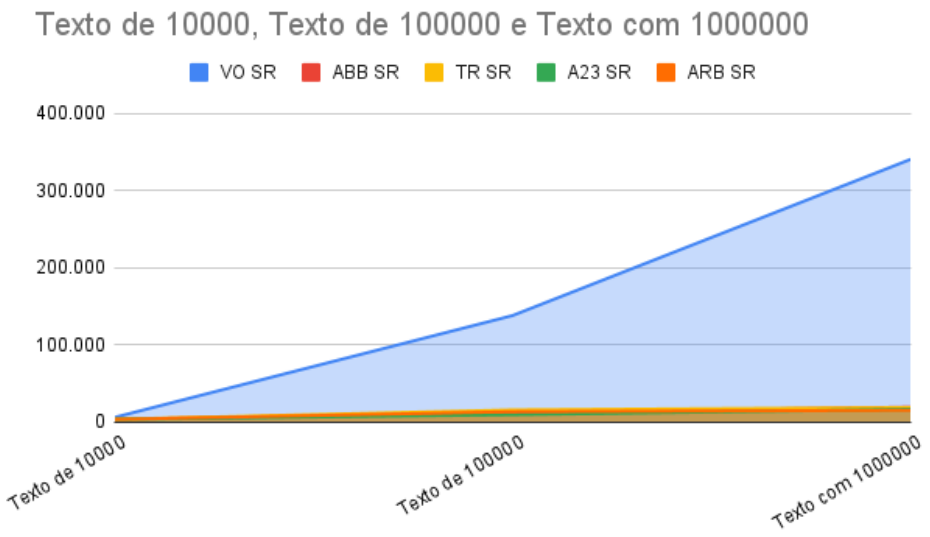

Figura 3: Gráfico Utilizando "SR"

2.1.4 Teste VD

este VD tem como objetivo identificar as menores palavras na tabela de símbolos que contenham o maior número de vogais sem repetição. Nesse teste, cada palavra é examinada para determinar quais delas possuem a maior quantidade de vogais únicas. Utilizando o bash script desenvolvido para esse propósito, o teste realiza a análise das palavras e registra o tempo de execução, fornecendo a lista das palavras identificadas. É importante ressaltar que, para ser considerada na lista, cada palavra precisa ter o maior número possível de vogais únicas, ou seja, vogais que não se repetem ao longo da palavra. Essa abordagem permite identificar e comparar as menores palavras que apresentam essa característica específica na tabela de símbolos.

A função menorVD implementa a lógica para encontrar a quantidade máxima de vogais sem repetição em uma estrutura de dados. A função percorre recursiva-

mente a árvore, ou o vetor, calculando o número de vogais não repetidas em cada nó e comparando com os valores encontrados nas subárvores esquerda e direita. Ela retorna o maior valor entre essas três opções.

Já a função menorTamanhoPalavraVD busca a menor palavra na ABB que atende ao critério da função anterior, de ter o maior número de vogais sem repetição. Ela também percorre a árvore de forma recursiva, comparando o número de vogais não repetidas em cada nó com o valor desejado (VD). Caso a palavra atenda ao critério, o tamanho da palavra é verificado e atualizado como o tamanho atual mínimo.

Por fim, a função imprimeVD é responsável por imprimir as palavras na ABB que possuem o maior número de vogais sem repetição (VD) e o menor tamanho tamanho específico (tam). Ela percorre a árvore de forma recursiva, verificando se cada palavra atende aos critérios estabelecidos e, em caso afirmativo, imprime a palavra na saída.

Tabela 4: Tabela VD

	Texto de 10000	Texto de 100000	Texto com 1000000
VO VD	0.06130	1.26140	3.44130
ABB VD	0.03770	0.10880	0.19560
TR VD	0.03780	0.16090	0.22870
A23 VD	0.03730	0.09660	0.20450
ARB VD	0.03630	0.11020	0.16360

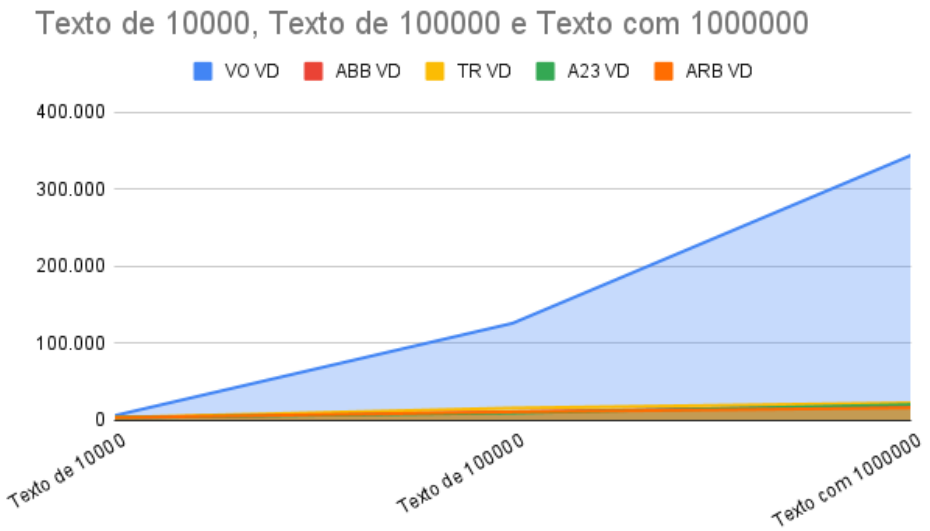

Figura 4: Gráfico Utilizando "VD"

2.2 Testes diferentes

2.2.1 Teste com muitas palavras

Nessa sessão foram feitos testes com um texto com muito mais palavras e analisado os resultados de relação entre as estruturas de dados. Nessa parte fica perceptível ainda que o vetor ordenado segue como o pior caso sempre, seguido da Treap, que necessita sempre de fazer muitas rotações, gerando mais tempo computacional;

Tabela 5: Tabela de texto com 1.000.000 palavras

	F	L	SR	VD
VO	0.130	0.123	0.121	0.117
ABB	0.038	0.038	0.039	0.039
TR	0.044	0.044	0.044	0.043
ARB	0.038	0.036	0.036	0.036
A23	0.039	0.038	0.038	0.038

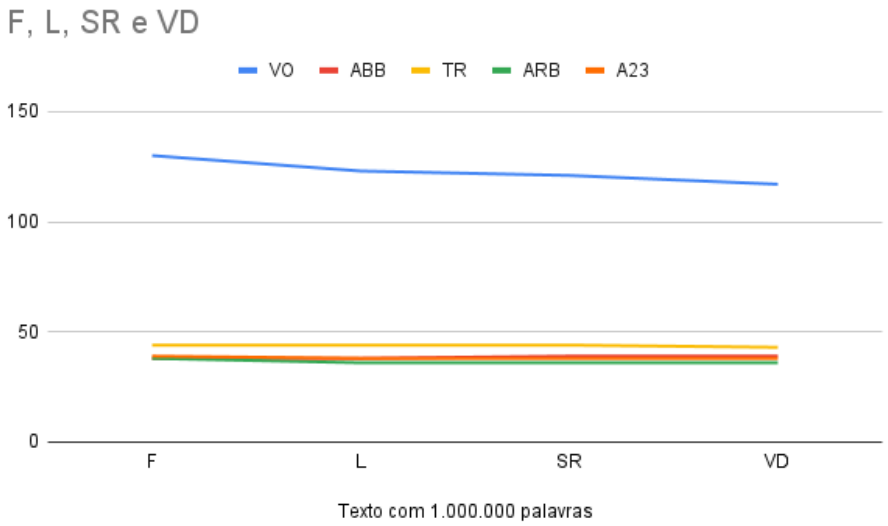

Figura 5: Gráfico Utilizando texto com muitas palavras

2.2.2 Teste com palavras ordenadas

Já aqui, os resultados seguem como o esperado, em que a Árvore de busca binária permanece em seu pior caso, sendo que seu tempo é precisamente maior

do que qualquer um dos outros; Já a estrutura Vetor Ordenado permanece em seu melhor caso, sendo a estrutura mais rápida nesse caso. Não sendo perceptível no gráfico mas sim na tabela abaixo, os valores são bem menores para esse caso.

Tabela 6: Tabela de texto com palavras ordenadas

	F	L	SR	VD
VO	0.019	0.011	0.009	0.007
ABB	0.741	0.940	1.206	1.211
TR	0.022	0.022	0.023	0.022
ARB	0.013	0.012	0.013	0.013
A23	0.015	0.015	0.014	0.014

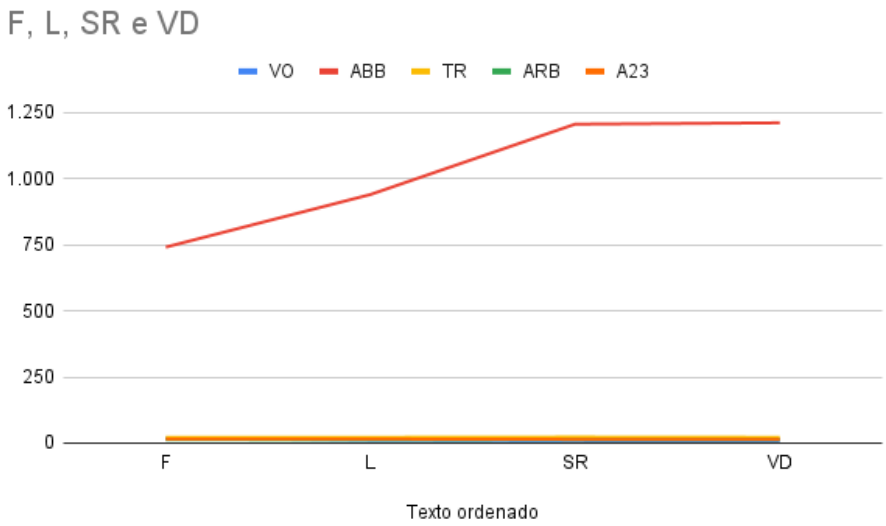

Figura 6: Gráfico Utilizando texto com palavras ordenadas

Nesse gráfico é importante pontuar que ele só é referente às médias das estruturas de um único texto ordenado, não se refere a uma comparação entre os tipos F, L , SR, VD, ou entre diferentes textos ordenados;

3 Conclusão

No decorrer deste trabalho, foram exploradas diferentes estruturas de dados, como árvores de busca binária (ABB) e vetor ordenado (VO), árvores 2-3 (A23), árvores rubro-negras (ARB) e Treaps (TR), para analisar o desempenho e a eficiência na realização de operações específicas. Concluiu-se que a escolha da estrutura de dados adequada é fundamental para obter um desempenho otimizado e reduzir a complexidade dos algoritmos.

Estruturas de dados mais complexas, como árvores 2-3 (A23) e árvores rubro-negras (ARN), mostraram-se mais eficientes em casos que envolvem um grande número de itens a serem armazenados e manipulados. Isso ocorre porque essas estruturas possuem mecanismos internos que garantem o equilíbrio e a distribuição adequada dos elementos, permitindo uma busca eficiente em tempo de O(log n) para a Rubro-negra e log3 N para a 2-3. A complexidade logarítmica significa que o tempo necessário para realizar uma operação aumenta de forma gradual, à medida que o tamanho da estrutura de dados aumenta, tornando-as mais adequadas para lidar com conjuntos de dados extensos.

Por outro lado, o vetor ordenado se mostrou uma opção menos eficiente em relação às estruturas de dados mencionadas anteriormente. O vetor requer que todos os elementos sejam armazenados em uma sequência ordenada, o que implica em realocação e deslocamento de elementos a cada inserção ou remoção. Além disso, a busca em um vetor ordenado possui complexidade linear (O(log n)), mas para fazer os testes é necessário passar um por um deixando o tempo de execução na ordem de (O(n)) ou maior a depender do teste, o que significa que o tempo necessário para encontrar um elemento aumenta proporcionalmente ao tamanho do vetor. Em casos de busca frequente ou com um grande número de elementos, essa complexidade linear pode resultar em tempos de execução significativamente maiores.

Esses resultados corroboram as teorias e conceitos estabelecidos na ciência da computação. A escolha da estrutura de dados adequada deve considerar o tamanho do conjunto de dados, o tipo de operações a serem realizadas e a eficiência desejada. Estruturas como A23 e ARN são projetadas para otimizar o desempenho em cenários de grande escala, mas também podem trazer problemas relacionados a memória do computador, enquanto o vetor ordenado pode ser mais adequado em casos específicos e com quantidades menores de elementos.

Portanto, a análise realizada neste trabalho reforça a importância de compreender as características e desempenho das diferentes estruturas de dados disponíveis. Ao selecionar a estrutura correta para cada situação, é possível obter um melhor desempenho, otimizar recursos computacionais e melhorar a eficiência dos algoritmos.