UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA

RODRIGO DE CASTRO MICHELASSI
NUSP: 13672703

# Exercício-Programa 2: Tabela de Símbolos

São Paulo
2023

**Sumário**

## 1 Introdução

Nesse EP, tínhamos como objetivo implementar algoritmos para organizar tabelas de símbolos utilizando C++ e algumas estruturas de dados vistas em aula. O EP foi desenvolvido com diversos arquivos *.h* e *.cpp* que contém as estruturas de dados em cada um e suas diversas operações, e um arquivo principal *ep*2.*cpp*, responsável por conectar todo o projeto.

Ao fim do programa, conseguimos testar e dar funcionalidade para diversos casos de teste grandes, os quais serão explorados mais a frente nesse relatório e enviados junto ao programa para correção.

## 2 Estruturas de Dados

Primeiramente, foram implementadas as estruturas de dados: Vetor Ordenado (feito de tamanho fixo, ao invés de dinâmico, para evitar problemas com memória), Árvore Binária de Busca, Treap, Árvore Rubro Negra. A ideia do EP é que pudéssemos comparar o desempenho dessas diversas estruturas de dados de formas diferentes, em diferentes situações.

Para utilizarmos essas estruturas de dados, todas possuem um padrão $String\ palavra$, a "chave"da tabela de símbolos, e $Item\ info$, onde Item é uma classe que armazena três parâmetros, quantidade de vogais sem repetição, quantidade de ocorrências de uma palavra e o tamanho de cada palavra.

As operações básicas implementadas em cada estrutura de dados foram as operações de inserção, o construtor do objeto e as operações secundárias, utilizadas como auxiliares na inserção de novos elementos. A ordenação das palavras é feita conforme a tabela ASCII.

## 3 Arquivos de Teste e Operações

Para que o EP possa ser compilado de forma mais fácil, foi enviado um arquivo MAKEFILE junto aos códigos. Para compilar, basta rodar no terminal o comando *make*, e o programa será compilado.

Junto ao EP foram enviados também 5 arquivos .txt usados diversas vezes nos testes. São eles:

- Exemplo: *exemplo.txt*, com 39 palavras
- Hamlet: *hamlet.txt*, com 33287 palavras
- MobyDick: *mobydick.txt*, com 209329 palavras.
- Bíblia: *holybible.txt*, com 826074 palavras.
- In Search of Lost Time: *holybible.txt*, com 1356895 palavras, ordenado.

Esses valores são importantes pois devem ser informados no começo da execução do programa. Vale lembrar que foram escolhidos esses 5 arquivos pois possuem tamanhos variados, para mostrar que a execução funciona para casos grandes e pequenos, e é se mostra também eficiente, como poderá ser consultada na seção de testes desse relatório. Caso seja necessario utilizar um outro arquivo de texto para testes, a quantidade de palavras pode ser obtida com o comando "wc -w nomeArquivo.txt". Além disso, estamos enviando um arquivo de texto ordenado, com a maior quantidade de palavras, para forçar o pior caso da ABB.

Foram feitas também as 5 operações pedidas no enunciado. O número de ocorrências da palavra no texto é a única operação, dentre essas 5 que foi feita em pós-processamento, todavia ainda funciona de maneira rápida. Ela segue um algoritmo iterativo, para o Vetor Ordenado, e um algoritmo recursivo para cada uma das árvores, e possui um desempenho satisfatório. Para o restante das operações executadas, todas elas são feitas com Pré-Processamento, na função *main* do código, antes que as palavras sejam enviadas para suas respectivas estruturas de dados. Com isso, garantimos que o resultado da consulta nessas operações seja instantâneo (O(m)), onde "m"é o tamanho do vetor a ser impresso.

A função *imprime*(), presente em todos os arquivos relativos às classes, não está sendo utilizada no código, porém foi utilizada nas fases iniciais de testes das estruturas de dados, para verificar que os dados recebidos estavam corretos. O código consta também com uma struct, presente no arquivo *item.h*, que contém um vector<string> e uma variável

frequência, e é utilizado para o pós processamento das palavras mais frequentes, presente nas operações das classes implementadas.

## 4 Execução

A ordem de execução do programa entregue segue a ordem pedida no enunciado, porém com uma pequena alteração na entrada do texto.

Ao iniciar o programa, é perguntado a estrutura desejada para testes. Nesse caso, o usuário deve digitar o número correspondente às opções de estruturas de dados mostrada. Após isso, é pedido a quantidade de palavras no arquivo que será testada, e para os arquivos de teste enviados essa quantidade é encontrada na seção Arquivos de Teste e Operações, desse relatório. Com isso, chega a diferença do nosso programa para a entrada dada no enunciado. Nosso programa pede que o usuário digite o nome de um arquivo do qual será retirado cada uma das palavras. É importante digitar o número correto de letras e o nome completo do arquivo, incluindo sua extensão, para que o programa funcione corretamente. As próximas entradas dizem respeito às consultas, e funcionam de forma análoga à escolha da estrutura de dados.

É fácil notar que para arquivos grandes, como o caso de *mobydick.txt* há um pequeno tempo de execução para o envio de todas as palavras para as funções de inserção, devido à realização de busca, para inserir, e os trechos que percorrem cada uma das palavras inteiramente, para realizar pré-processamentos, com base na análise de presença de vogais nela.

## 5 Exemplos de Execução

**Algoritmo 1** Consultas com Vetor Ordenado

| | | |
|---|---|---|
| 1: | 1 | ▷ Selecionando a estrutura de dados |
| 2: | 209329 | ▷ Número de Palavras no texto |
| 3: | mobydick.txt | ▷ Nome do arquivo a ser lido |
| 4: | 4 | ▷ Número de Consultas a serem feitas |
| 5: | 4 | ▷ SR |
| 6: | → outspreadingly | ▷ Saída |
| 7: | 3 | ▷ L |
| 8: | → tastefullyornamented | ▷ Saídas |
| 9: | → ninterpenetratingly | ▷ |
| 10: | → immortalitypreserver | ▷ |
| 11: | 2 | ▷ O(palavra) |
| 12: | the | ▷ Palavra a ser consultada |
| 13: | → A palavra the ocorreu 13553 vezes | ▷ Saída |
| 14: | 1 | ▷ Palavras mais frequentes |
| 15: | → the | ▷ Saída |

**Algoritmo 2** Consultas com ABB

| | | |
|---|---|---|
| | 2 | ▷ Selecionando a estrutura de dados |
| 2: | 826074 | ▷ Número de Palavras no texto |
| | holybible.txt | ▷ Nome do arquivo a ser lido |
| 4: | 3 | ▷ Número de Consultas a serem feitas |
| | 3 | ▷ L |
| 6: | → jonathelemrechokim | ▷ Saídas |
| | → mahershalalhashbaz | ▷ |
| 8: | 1 | ▷ F |
| | → the | ▷ Saída |
| 10: | 4 | ▷ SR |
| | → buryingplace | ▷ Saídas |
| 12: | → unprofitable | ▷ |

Note que, nesse caso de teste, ao fazer as consultas 3 e 4, ocorre uma repetição dessas palavras da saída. Esse problema não foi encontrado no código e acredita-se ser devido a leitura do arquivo.

**Algoritmo 3** Consultas com Treap

```
    3                                   ▷ Selecionando a estrutura de dados
    39                                  ▷ Número de Palavras no texto
3:  exemplo.txt                         ▷ Nome do arquivo a ser lido
    3                                   ▷ Número de Consultas a serem feitas
    2                                   ▷ O(palavra)
6:  uma                                 ▷ palavra a ser comsultada
    → 4                                 ▷ Saídas
    4                                   ▷ SR
9:  → demais                            ▷ Saídas
    → senhor                            ▷
    → quiser                            ▷
12: 1                                   ▷ SR
    → uma                               ▷ Saídas
```

**Algoritmo 4** Consultas com Rubro Negra

```
    5                                   ▷ Selecionando a estrutura de dados
    32081                               ▷ Número de Palavras no texto
    hamlet.txt                          ▷ Nome do arquivo a ser lido
    4                                   ▷ Número de Consultas a serem feitas
5:  3                                   ▷ L
    → transformation                    ▷ Saída
    4                                   ▷ SR
    → unprofitable                      ▷ Saída
    5                                   ▷ VD
10: → behaviour                         ▷ Saídas
    → favourite                         ▷
    2                                   ▷ O(Palavra)
    the                                 ▷ palavra a ser comsultada
    → 870 vezes                         ▷
```

## 6 Testes e Desempenho

Nessa seção será testado o desempenho do nosso programa. Para isso, iremos fazer 3 testes para cada categoria, em cada um dos nossos arquivos de saída, utilizando a biblioteca Chrono do C++. Ao fim, também utilizaremos o mesmo arquivo porém ordenado, para medir o tempo de construção nessas condições também.

Os critérios utilizados para medir o tempo serão:

**I. Tempo de construção** → tempo gasto para fazer os pré-processamentos das funções e enviar as palavras lidas para as estruturas de dados respectivas.

**II. Palavras mais Frequentes** → essa é a única função de consulta do programa que não funciona por pré-processamento, sendo assim, também deve ser testada separadamente, para sabermos quanto tempo essa consulta leva.

Os dados sobre os arquivos podem ser encontrados nesse mesmo relatório, na seção Arquivos de Teste e Operações. Note que todas as estruturas de dados testadas tem uma ordem de execução aproximadamente $O(logN)$, onde N é o número de palavras salvas, ou nós salvos na estrutura, que varia para o caso da árvore 2-3.

### *6.1 exemplo.txt*

| | Tempo de Construção (s) | | |
|---|---|---|---|
| Estrutura | Teste 1 | Teste 2 | Teste 3 |
| VO | 0,000294 | 0,000323 | 0,000254 |
| ABB | 0,000287 | 0,000278 | 0,000253 |
| Treap | 0,000423 | 0,000432 | 0,000433 |
| 2-3 | 0,000298 | 0,000331 | 0,000295 |
| Red Black | 0,000292 | 0,000286 | 0,000331 |

| Estrutura | Palavras mais Frequentes | | |
|---|---|---|---|
| | Teste 1 | Teste 2 | Teste 3 |
| VO | 0,000112 | 0,000070 | 0,000085 |
| ABB | 0,000096 | 0,000088 | 0,000083 |
| Treap | 0,000089 | 0,000087 | 0,000081 |
| 2-3 | 0,000100 | 0,000105 | 0,000094 |
| Red Black | 0,000107 | 0,000136 | 0,000087 |

Não há muito para se analisar por aqui. Pelo fato do arquivo de teste ser muito pequeno, todas as estruturas de teste apresentam um resultado similarmente satisfatórios.

### *6.2 hamlet.txt*

| Estrutura | Tempo de Construção (s) | | |
|---|---|---|---|
| | Teste 1 | Teste 2 | Teste 3 |
| VO | 0,121476 | 0,119298 | 0,122485 |
| ABB | 0,044699 | 0,044370 | 0,044067 |
| Treap | 0,046664 | 0,046352 | 0,046100 |
| 2-3 | 0,041992 | 0,043810 | 0,042933 |
| Red Black | 0,046967 | 0,046497 | 0,043912 |

| Estrutura | Palavras mais Frequentes | | |
|---|---|---|---|
| | Teste 1 | Teste 2 | Teste 3 |
| VO | 0,000080 | 0,000036 | 5,000076 |
| ABB | 0,000871 | 0,000886 | 0,000881 |
| Treap | 0,000861 | 0,000912 | 0,000940 |
| 2-3 | 0,000651 | 0,000661 | 0,000673 |
| Red Black | 0,000861 | 0,000882 | 0,000858 |

Nesse novo exemplo, podemos analisar algo muito interessante. A construção de todas estruturas se mostram pelo menos uma casa decimal mais rápida que o Vetor Ordenado, levando cerca de metade do tempo para a construção, todavia a busca de palavras mais frequentes ocorre muito mais rapidamente no Vetor Ordenado em relação às estruturas de árvores. Todavia, o texto ainda é muito pequeno para se tirar conclusões mais aprofundadas.

*6.3 mobydick.txt*

| Estrutura | Tempo de Construção (s) | | |
|---|---|---|---|
| | Teste 1 | Teste 2 | Teste 3 |
| VO | 1,302720 | 1,302169 | 1,304292 |
| ABB | 0,218433 | 0,219486 | 0,220362 |
| Treap | 0,253461 | 0,229507 | 0,227533 |
| 2-3 | 0,198636 | 0,200580 | 0,222083 |
| Red Black | 0,237663 | 0,236290 | 0,258039 |

| Estrutura | Palavras mais Frequentes | | |
|---|---|---|---|
| | Teste 1 | Teste 2 | Teste 3 |
| VO | 0,000072 | 0,000070 | 0,000071 |
| ABB | 0,003449 | 0,003017 | 0,002980 |
| Treap | 0,002999 | 0,002984 | 0,002955 |
| 2-3 | 0,002221 | 0,002219 | 0,002247 |
| Red Black | 0,003016 | 0,003005 | 0,002987 |

Novamente, o Vetor Ordenado se mostrou mais eficiente para a busca de palavras do que o restante das estruturas, porém agora, com um texto maior, essa diferença se tornou ainda mais imponente. Todavia, esse é o único sentido em que o VO tem um desempenho melhor que o das árvores, tendo em vista que a construção de todas ainda é muito mais rápida, e, como para o restante das operações são feitos pré-processamentos, ainda parece mais vantajoso utilizar uma estrutura de árvore, como a ABB ou árvore 2-3, que estão apresentando os melhores resultados até o momento.

### *6.4 holybible.txt*

| Estrutura | Tempo de Construção (s) | | |
|---|---|---|---|
| | Teste 1 | Teste 2 | Teste 3 |
| VO | 1,244538 | 1,209478 | 1,216635 |
| ABB | 0,718552 | 0,718863 | 0,719123 |
| Treap | 0,698154 | 0,696937 | 0,698041 |
| 2-3 | 0,692500 | 0,689868 | 0,687496 |
| Red Black | 0,776190 | 0,781916 | 0,771738 |

| Estrutura | Palavras mais Frequentes | | |
|---|---|---|---|
| | Teste 1 | Teste 2 | Teste 3 |
| VO | 0,000056 | 0,000057 | 0,000076 |
| ABB | 0,002147 | 0,002098 | 0,002119 |
| Treap | 0,002101 | 0,002096 | 0,002115 |
| 2-3 | 0,002116 | 0,002120 | 0,001634 |
| Red Black | 0,002102 | 0,002141 | 0,002110 |

O mesmo resultado observado anteriormente se mantém aqui, todavia todos os códigos se mostram muito eficientes ainda. Para testar ainda mais profundamente, iremos utilizar um arquivo ordenado, chamado $in-search.txt$, versão ordenada do livro In Search of Lost Time, com 1356895 palavras.

### *6.5 in-search.txt (Ordenado)*

| Estrutura | Tempo de Construção (s) | | |
|---|---|---|---|
| | Teste 1 | Teste 2 | Teste 3 |
| VO | 0,250477 | 0,244912 | 0,248788 |
| ABB | 0,608824 | 0,597949 | 0,594551 |
| Treap | 0,538847 | 0,533045 | 0,989031 |
| 2-3 | 0,170583 | 0,166734 | 0,166235 |
| Red Black | 0,664922 | 0,664913 | 0,664839 |

| | Palavras mais Frequentes | | |
|---|---|---|---|
| Estrutura | Teste 1 | Teste 2 | Teste 3 |
| VO | 0,000067 | 0,000064 | 0,000078 |
| ABB | 0,603243 | 0,599950 | 0,606858 |
| Treap | 0,536941 | 0,534958 | 0,531198 |
| 2-3 | 0,169915 | 0,166673 | 0,174950 |
| Red Black | 0,165286 | 0,166445 | 0,171576 |

Nesse último teste, podemos ver um crescimento gigante para a busca feita nas estruturas de árvores, subindo um total de duas casas decimais para um arquivo maior, com palavras ordenadas. A ABB, assim como previmos, ainda teve a pior performance nesse caso, visto que para um arquivo totalmente ordenado a ABB possui busca de ordem linear. Todavia, a diferença não foi tão grande em comparação com o restante das estruturas, Provavelmente, seria mais eficiente realizar esse teste para arquivos ainda maiores.

## 7 Considerações Finais

Ao realizar esse projeto, foi possível revisar e entender um pouco mais sobre a aplicação das estruturas de dados vistas em sala para a construção de tabelas de símbolos. Assim, além de revisar a forma que as estruturas eram construídas e colocadas em um algoritmo, foi possível também observar formas de utilizar essas estruturas para obter dados desejados e armazená-los, conforme fosse necessário.

Além disso, o EP nos deu a chance de trabalhar com arquivos grandes e formas de alocação de memória feita utilizando a sintaxe de classes em C++, que, embora já tenha sido utilizada antes, foi utilizada novamente de uma forma mais ampla e notável. Com isso também tivemos a oportunidade de utilizar templates (no caso, Vector), do STL, ferramenta da biblioteca padrão do C++.

Ademais, foi o primeiro projeto feito na Universidade que nos deu a oportunidade de ver em prática como funciona o pré-processamento de dados e porque ele se mostra importante no uso de programas grandes, visto que, com arquivos maiores de texto, como dado na entrega, performou uma rápida resposta às operações de consulta, o que só foi possível graças ao pré-processamento de dados.

Quanto aos resultados obtidos, podemos dizer que todas estruturas funcionam de forma satisfatória para arquivos com até 1 milhão de parâmetros, que foi o máximo testado e mostrado. Nesse sentido, a estrutura testada mais rápida para construção em casos gerais foi a ABB, porém que se mostrou mais lenta para arquivos já ordenados. Todavia, como textos em média não são ordenados, ela se mostra uma boa candidata, pois além de rápida é também fácil de se implementar. As árvores 2-3 também apresentaram resultados satisfatórios. Já para a busca e consulta, o Vetor Ordenado se mostrou muito mais eficiente que o restante das estruturas, e pode ser considerado para projetos que devem realizar muitas consultas.

Dessa forma, conclui-se que o programa funciona de forma eficiente e satisfatória, e foi de enorme importância para o desenvolvimento intelectual dos alunos.