# TRABALHO FINAL DE ESTRUTURA DE DADOS

## Comparação de Eficiência entre Árvores Binárias

Eduardo Spitzer Fischer - 00290399
Gabriel Lando - 00291399

Segunda-feira, 18 de dezembro de 2017
Porto Alegre, RS

Universidade Federal do Rio Grande do Sul – Instituto de Informática

**RESUMO**

Nesse trabalho, foram testadas as eficiências das estruturas de dados Árvores Binarias de Pesquisa (ABP), Árvores AVL e Árvores Splay para a realização das tarefas de inserção e consulta em uma tabela de conversão Código Morse - ASCII. O objetivo do trabalho é definir qual árvore realiza a tarefa de traduzir um arquivo de texto para o código Morse com a maior eficiência computacional. Para a realização deste, foi necessário escrever um código para processar os arquivos texto de entrada e saída em diferentes estruturas de árvore, sendo uma dessas estruturas a *ABP* e a outra uma árvore balanceada escolhida pelo grupo (AVL, Rubro-Negra ou Splay).

Nosso grupo optou pela Árvore AVL como a segunda estrutura de comparação. No entanto, após a conclusão do trabalho, optamos por implementar uma terceira árvore balanceada, a fim de comparar sua eficiência com a árvore AVL. Para isso, optamos pela árvore Splay por ser uma estrutura de comportamento diferenciado, onde tanto a inserção quanto a busca modificam a raiz da árvore, o que a torna mais eficiente para algumas situações, a exemplo do tratamento de textos em linguagem natural.

Sendo assim, o trabalho consiste na comparação de três tipos de árvores binárias: ABP, AVL e Splay. A metodologia e os resultados serão descritos e explicados no decorrer do relatório

***Palavras-chave*:** Árvores binárias, ABP, AVL, Splay, Código Morse, Estrutura de Dados, Eficiência, Comparação.

# TRABALHO FINAL DE ESTRUTURA DE DADOS

## Comparação de Eficiência entre Árvores Binárias

Árvores binárias são estruturas de dados formadas por nodos interligados entre sí. O primeiro nodo de uma árvore é denominado *raiz*. Todos os outros nodos são *filhos* de algum nodo e cada nodo da árvore pode ter até dois filhos, sendo o filho da esquerda necessariamente *menor* que o nodo pai e o filho da direita necessariamente *maior* que o nodo pai.

Isso faz com que a árvore consiga organizar os dados de forma a otimizar as consultas, uma vez que o resultado será encontrado com muito menos comparações quando comparado a uma estrutura de lista. Árvores são estruturas interessantes de serem estudadas, pois dependendo da forma que programador resolve balanceá-las, podem ter uma maior ou menor eficiência. Para isso, foram realizado alguns testes com diferentes formas de balanceamento em árvores binárias, a fim de comparar sua eficiência em alguns casos de teste, sempre usando textos como entrada e saída.

## ALGORÍTMOS UTILIZADOS

Para a realização desse trabalho utilizamos a linguaguem de programação C, por ser mais simples e possuir uma boa eficiência, tornando nossos resultados válidos para a disciplina de Estrutura de Dados. As principais funções utilizadas estão descritas a seguir:

*criarArvore():* Função responsável pela criação da árvore. Ao inicializar uma árvore, é necessário que ela possua o valor *NULL*, para evitar que lixos de memória afetem o funcionamento da mesma.

*buscaChar():* Funçao que realiza a busca binária nas árvores ABP e AVL, responsável por realizar a contagem de comparações nas buscas para formar o arquivo de saída.

*processaEntrada():* Função chamada na *main*, onde são passadas a árvore e os arquivos de entrada e saída como parâmetros. É essa função que gera o arquivo de saída, de acordo com os resultados da consulta usando a função *buscaChar()*.

*processaEntradaSplay():* Função chama da *main*, semelhante à *processaEntrada()*, utilizada para gerar a saída de acordo com a Árvore Splay.

*AlturaArvore():* Calcula a altura da árvore final, após terminadas as inserções. É utilizada para calcular a altura da ABP e AVL. Não foi utilizada na Splay, pois a altura desse tipo de árvore varia conforme as operações básicas são realizadas.

*removeAcento():* Função usada para remover acentuações do arquivo de entrada e processá-lo normalmente como um arquivo de texto simples, de acordo com a tabela ASCII não extendida.

*Splay():* Função responsável por inserir nodos na árvore Splay. Insere o novo nó na raiz da árvore. Também é utilizada para a busca de um elemento, pois uma busca em Splay retorna o elemento buscado como a raiz da árvore.

## CONTEXTO EXPERIMENTAL

Para a realização deste trabalho, foi utilizado o código em C escrito pelos integrantes do grupo, o qual avalia a eficiência de 3 tipos de árvores binárias: ABP, AVL e Splay. Como parâmetros na chamada do executável são passados: um arquivo de texto com a tabela de tradução do Código Morse, um arquivo de texto de entrada (o qual será convertido para código morse) e o nome do arquivo de saída. Além do arquivo de saída, é exibida no console a quantidade de comparações necessárias para criar a árvore e gerar o arquivo de saída.

Para viabilizar nossos resultados, utilizamos 10 arquivos de entrada e 3 versões da tabela do Código Morse. Os arquivos de entrada de texto possuem os mais diversos tamanhos, desde arquivos bem pequenos até arquivos com milhares de caracteres. Já as versões da tabela de Código Morse consistem em versões embaralhadas do mesmo alfabeto, visto que uma tabela em ordem alfabética gera uma ABP em forma de lista, a qual possui uma eficiência muito baixa nas buscas.

Os resultados do experimento estão descrito abaixo na seção RESULTADOS em forma de tabelas e gráficos. Logo após, a seção CONCLUSÕES indica qual estrutura se mostrou mais eficiente para a realização da tarefa e explicita os motivos para tal conclusão.

## RESULTADOS

Para a realização do experimento, utilizamos um algorítmo que processa todos os casos de entrada e gera um arquivo de texto com as saídas, facilitando a obtenção dos resultados, uma vez que foram realizados 30 casos de teste no total. Após isso, chegamos nos seguintes resultados:

**CASO 1:** Tabela com Código Morse disponibilizada no Moodle da disciplina:

- **Comparações realizadas para a montagem das árvores:**

| | **Inserção na Árvore** | **Altura da Árvore** |
|---|---|---|
| ABP | 519 | 27 |
| AVL | 812 | 7 |
| Splay | 476 | ---- |

- **Resultados da ABP:**

| | **Quantidade de Caracteres** | **Geração da Saída** | **Total de Comparações** |
|---|---|---|---|
| Entrada 1 | 72 | 1.593 | 2.112 |
| Entrada 2 | 344 | 6.767 | 7.286 |
| Entrada 3 | 1.350 | 26.197 | 26.716 |
| Entrada 4 | 4.779 | 92.140 | 92.659 |
| Entrada 5 | 10.222 | 195.545 | 196.064 |
| Entrada 6 | 13.752 | 266.185 | 266.704 |
| Entrada 7 | 34.269 | 662.570 | 663.089 |
| Entrada 8 | 51.134 | 984.317 | 984.836 |
| Entrada 9 | 102.469 | 1.972.116 | 1.972.635 |
| Entrada 10 | 340.434 | 6.519.056 | 6.519.575 |
| **Total** | **558.825** | **10.726.486** | **10.731.676** |

- **Resultados da AVL:**

| | **Quantidade de Caracteres** | **Geração da Saída** | **Total de Comparações** |
|---|---|---|---|
| Entrada 1 | 72 | 505 | 1.317 |
| Entrada 2 | 344 | 2.409 | 3.221 |
| Entrada 3 | 1.350 | 9.727 | 10.539 |
| Entrada 4 | 4.779 | 34.000 | 34.812 |
| Entrada 5 | 10.222 | 73.229 | 74.041 |
| Entrada 6 | 13.752 | 98.127 | 98.939 |
| Entrada 7 | 34.269 | 245.086 | 245.898 |
| Entrada 8 | 51.134 | 365.455 | 366.267 |

| | | | |
|---|---|---|---|
| Entrada 9 | 102.469 | 732.710 | 733.522 |
| Entrada 10 | 340.434 | 2.430.556 | 2.431.368 |
| **Total** | **558.825** | **3.991.804** | **3.999.924** |

- **Resultados da Splay:**

| | **Quantidade de Caracteres** | **Geração da Saída** | **Total de Comparações** |
|---|---|---|---|
| Entrada 1 | 72 | 229 | 705 |
| Entrada 2 | 344 | 919 | 1.395 |
| Entrada 3 | 1.350 | 3.445 | 3.921 |
| Entrada 4 | 4.779 | 12.051 | 12.527 |
| Entrada 5 | 10.222 | 25.709 | 26.185 |
| Entrada 6 | 13.752 | 34.673 | 35.149 |
| Entrada 7 | 34.269 | 86.250 | 86.726 |
| Entrada 8 | 51.134 | 128.509 | 128.985 |
| Entrada 9 | 102.469 | 257.634 | 258.110 |
| Entrada 10 | 340.434 | 855.050 | 855.526 |
| **Total** | **558.825** | **1.404.469** | **1.409.229** |

De acordo com os resultados obtidos, percebemos que ao usar uma tabela praticamente em ordem alfabética, tivemos um desempenho muito ruim usando a ABP, visto que a árvore adquire um altura relativamente elevada, dificultando as buscas em letras do final do alfabeto.

Já a AVL possuiu um desempenho mediano, visto que mantém sua estrutura balanceada, facilitando as buscas. Por outro lado, letras no início do alfabeto acabam ficando mais longe da raiz, o que dificulta a busca destes elementos.

A árvore Splay obteve um desempenho muito bom, pois a maior parte das buscas acontecem em letras próximas, o que acaba por facilitar a busca dos caracteres seguintes devido à sua propriedade de manter as últimas buscas próximas à raiz.

**CASO 2:** Tabela com Código Morse em ordem aleatória:

- **Comparações realizadas para a montagem das árvores:**

| | **Inserção na Árvore** | **Altura da Árvore** |
|---|---|---|
| ABP | 340 | 13 |
| AVL | 773 | 7 |
| Splay | 860 | ---- |

- **Resultados da ABP:**

| | Quantidade de Caracteres | Geração da Saída | Total de Comparações |
|---|---|---|---|
| Entrada 1 | 72 | 863 | 1.203 |
| Entrada 2 | 344 | 4.139 | 4.479 |
| Entrada 3 | 1.350 | 16.413 | 16.753 |
| Entrada 4 | 4.779 | 58.488 | 58.828 |
| Entrada 5 | 10.222 | 125.215 | 125.555 |
| Entrada 6 | 13.752 | 169.321 | 169.661 |
| Entrada 7 | 34.269 | 419.616 | 419.956 |
| Entrada 8 | 51.134 | 624.757 | 625.097 |
| Entrada 9 | 102.469 | 1.253.142 | 1.253.482 |
| Entrada 10 | 340.434 | 4.160.362 | 4.160.702 |
| **Total** | **558.825** | **6.832.316** | **6.835.716** |

- **Resultados da AVL:**

| | Quantidade de Caracteres | Geração da Saída | Total de Comparações |
|---|---|---|---|
| Entrada 1 | 72 | 643 | 1.416 |
| Entrada 2 | 344 | 2.951 | 3.724 |
| Entrada 3 | 1.350 | 11.655 | 12.428 |
| Entrada 4 | 4.779 | 41.000 | 41.773 |
| Entrada 5 | 10.222 | 87.715 | 88.488 |
| Entrada 6 | 13.752 | 118.467 | 119.240 |
| Entrada 7 | 34.269 | 439.811 | 440.584 |
| Entrada 8 | 51.134 | 365.455 | 366.267 |
| Entrada 9 | 102.469 | 881.798 | 882.571 |
| Entrada 10 | 340.434 | 2.925.274 | 2.926.047 |
| **Total** | **558.825** | **4.874.769** | **4.882.538** |

- **Resultados da Splay:**

| | Quantidade de Caracteres | Geração da Saída | Total de Comparações |
|---|---|---|---|
| Entrada 1 | 72 | 218 | 1.078 |
| Entrada 2 | 344 | 927 | 1.787 |
| Entrada 3 | 1.350 | 3.458 | 4.318 |
| Entrada 4 | 4.779 | 12.058 | 12.918 |
| Entrada 5 | 10.222 | 25.706 | 26.566 |
| Entrada 6 | 13.752 | 34.681 | 35.541 |
| Entrada 7 | 34.269 | 86.245 | 87.105 |
| Entrada 8 | 51.134 | 128.522 | 129.382 |
| Entrada 9 | 102.469 | 257.630 | 258.490 |

| | | | |
|---|---|---|---|
| Entrada 10 | 340.434 | 855.047 | 855.907 |
| **Total** | **558.825** | **1.404.492** | **1.413.092** |

Nessa bateria de testes, percebemos uma redução considerável na quantidade de comparações necessárias para a criação do arquivo de saída usando a ABP, uma vez que ao inicializar a árvore com uma letra do meio da tabela a estrutura adquire um certo balanceamento.

Para as outras duas árvores analisadas, a diferença foi bem menor. Isso pode ser explicado pelo fato de as árvores serem auto-balanceadas. Sendo assim, a formação final das árvores nao difere muito daquela formada na primeira bateria de testes. No entanto, essa pequena diferença se deve à ordem de inserção dos nós na árvore, fazendo com que haja uma pequena diferença na ordem dos nós, que por consequência interfere na quantidade de comparações final.

**CASO 3:** Tabela com Código Morse em ordem inversa:

- **Comparações realizadas para a montagem das árvores:**

| | **Inserção na Árvore** | **Altura da Árvore** |
|---|---|---|
| ABP | 553 | 28 |
| AVL | 869 | 7 |
| Splay | 496 | ---- |

- **Resultados da ABP:**

| | **Quantidade de Caracteres** | **Geração da Saída** | **Total de Comparações** |
|---|---|---|---|
| Entrada 1 | 72 | 1.865 | 2.418 |
| Entrada 2 | 344 | 9.259 | 9.812 |
| Entrada 3 | 1.350 | 36.585 | 37.138 |
| Entrada 4 | 4.779 | 128.288 | 128.841 |
| Entrada 5 | 10.222 | 275.205 | 275.758 |
| Entrada 6 | 13.752 | 369.857 | 370.410 |
| Entrada 7 | 34.269 | 920.762 | 921.315 |
| Entrada 8 | 51.134 | 1.374.337 | 1.374.890 |
| Entrada 9 | 102.469 | 2.758.168 | 2.758.721 |
| Entrada 10 | 340.434 | 9.181.298 | 9.181.851 |
| **Total** | **558.825** | **15.055.624** | **15.061.154** |

- **Resultados da AVL:**

| | **Quantidade de Caracteres** | **Geração da Saída** | **Total de Comparações** |
|---|---|---|---|
| Entrada 1 | 72 | 731 | 1.600 |
| Entrada 2 | 344 | 3.383 | 4.252 |
| Entrada 3 | 1.350 | 13.099 | 13.968 |
| Entrada 4 | 4.779 | 46.130 | 46.999 |
| Entrada 5 | 10.222 | 98.893 | 99.762 |
| Entrada 6 | 13.752 | 133.025 | 133.894 |
| Entrada 7 | 34.269 | 331.550 | 332.419 |
| Entrada 8 | 51.134 | 494.227 | 495.096 |
| Entrada 9 | 102.469 | 989.954 | 990.823 |
| Entrada 10 | 340.434 | 3.289.544 | 3.290.413 |
| **Total** | **558.825** | **5.400.536** | **5.409.226** |

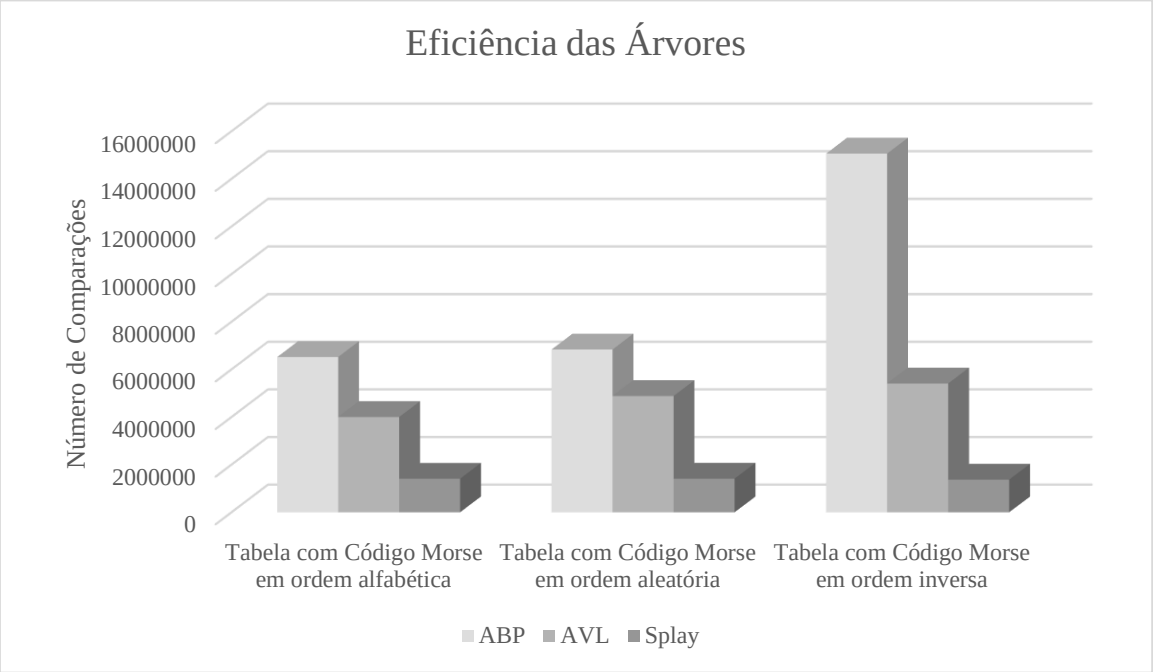
- **Resultados da Splay:**

| | **Quantidade de Caracteres** | **Geração da Saída** | **Total de Comparações** |
|---|---|---|---|
| Entrada 1 | 72 | 234 | 730 |
| Entrada 2 | 344 | 885 | 1.381 |
| Entrada 3 | 1.350 | 3.372 | 3.868 |
| Entrada 4 | 4.779 | 11.727 | 12.223 |
| Entrada 5 | 10.222 | 24.959 | 25.455 |
| Entrada 6 | 13.752 | 33.721 | 34.217 |
| Entrada 7 | 34.269 | 83.884 | 84.380 |
| Entrada 8 | 51.134 | 124.988 | 125.484 |
| Entrada 9 | 102.469 | 250.495 | 250.991 |
| Entrada 10 | 340.434 | 831.197 | 831.693 |
| **Total** | **558.825** | **1.365.462** | **1.370.422** |

Na nossa última bateria de testes, resolvemos inverter a ordem da tabela de Código Morse a fim de testar a eficiência da ABP em um caso onde a árvore formada se assemelharia a uma lista em ordem decrescente de valores na Tabela ASCII. Sendo assim, os casos de teste deveriam gerar um valor muito mais alto de comparações nesse tipo de árvore, uma vez que a maior parte dos caracteres mais utilizados estariam distantes da raiz.

Como pode ser percebido nos resultados, a ABP se tornou muito pouco eficiente, precisando de quase 3 vezes mais comparaçoes que a AVL para gerar a mesma saída e aproximadamente 11 vezes mais que a árvore Splay.

No gráfico abaixo, os valores totais da quantidade de comparações são exibidos para as três baterias de testes, tornando a visualização dos dados mais intuitiva.



**CONCLUSÃO:**

Após o encerramento dos nossos testes, podemos concuir que a árvore binária mais eficiente foi a Splay, tendo um trabalho de processamento de apenas 21,6% daquele necessário ao utilizar a ABP e 35,2% em relação a AVL. Analizando esses dados, percebemos que essa grande vantagem do uso de árvores Splay foi causada pelo uso de textos em linguagem natural nos casos de teste (ao invés de números, por exemplo), uma vez que as letras mais utilizadas tendem a ficar mais próximas uma das outras e da raiz nessa árvore. Como a Splay mantem o último item pesquisado em sua raiz e o próximo item a ser pesquisado está proximo ao anterior, a quantidade de comparações acaba se tornando muito menor, melhorando muito a eficiência da árvore Splay quando comparada às outras duas estruturas utilizadas.

Já a árvore ABP foi a que teve o pior resultado em nossos testes, visto que na maioria dos casos, gerou uma árvore extremamente grande, dificultando as buscas, diferentemente da árvore AVL, que manteve os dados balanceados.

Sendo assim, chegamos a conclusão que a árvore mais eficiente para buscas em arquivos de texto em linguagem natural é a árvore Splay.