Universidade Federal de Juiz de Fora
Departamento de Ciência da Computação

ESTRUTURA DE DADOS 2
BOOK DEPOSITORY DATASET

Ana Beatriz Kapps dos Reis - 201835006 - Turma A

Felipe Israel - 201835041 - Turma B

Ian Couto de Paula - 201876002 - Turma A

Rodrigo Torres Rego - 201876029 - Turma A

José Jeronimo Camata

Marcelo Caniato Renhe

Juiz de Fora

25 de Novembro de 2020

Sumário

1 Execução

1. Baixe e instale o Java JDK 8 ou acima:

   - Windows

     https://adoptopenjdk.net/

   - Linux

     ```
     sudo apt install default-jre
     ```

2. Clone o repositório do projeto do github:

   ```
   git clone https://github.com/Rodrigo947/trabalhoed2
   ```

3. Faça o download do dataset no link abaixo e extraia os arquivos na pasta data do projeto.

   https://drive.google.com/file/d/1akPqLZg-_Ug2SUyPWJ4bwjcWFP9tsKgL/view?usp=sharing

4. Abra um terminal na pasta e execute o comando:

   ```
   java -cp trabalhoed2.jar Main
   ```

2 Instruções de Uso

Ao executar programa o seguinte menu será mostrado:

```
TRABALHO DE ESTRUTURA DE DADOS 2
1-Cenario 1
2-Cenario 2
3-Cenario 3
4-Parte 2
5-Parte 3
6-Gerar Arquivo de Objetos
7-Gerar Arquivos de Arrays
0-Sair
Opcao:
```

Antes de processar quaisquer opções de 1 à 5 é necessário rodar as opções 6 e 7 em sequência.

2.1 Opção 6

Responsável por fazer o pré-processamento do dataset criando um arquivo de objetos onde cada objeto guarda um registro do dataset.

O arquivo gerado está em *data/datasetOBJ.txt*.

2.2 Opção 7

A partir do arquivo *data/entrada.txt*, essa opção gera os arquivos de arrays aleatórios que serão usados por cada cenário e os guarda na pasta *data/arrays/*. Uma pasta é criada para cada tamanho de vetor e o nome de dos arquivos seguem o seguinte padrão: **tipoDeArray_tamanhoDoArray_seed.txt**

OBS.: A geração desses arquivos podem demorar muito tempo, portanto caso queira pular a execução das opções 6 e 7, faça o download de todos os arquivos tratados e extraia na pasta data.

https://drive.google.com/file/d/1aktD1ncPi1pcvm-8jznUlVvDC4oOfZT_/view?usp=sharing

3 Cenários

Os resultados de cada cenário são armazenados em *resultados*. A pasta está divida pelo nome das ordenações, então dependendo do cenário executado, as determinadas pastas serão preenchidas.

—Cenário 1—
Executa apenas o QuickSort Recursivo.
Pasta a ser preenchida:
resultados/QuickSort/cenario1/Recursivo

—Cenário 2—
Executa os seguintes tipos de QuickSort: Recursivo, Mediana, Inserção.
Pastas a serem preenchidas:
resultados/QuickSort/cenario2/Recursivo
resultados/QuickSort/cenario2/Mediana
resultados/QuickSort/cenario2/Insercao

—Cenário 3—
Executa as ordenações Heap Sort ,Insertion Sort, Merge Sort e Tree Sort
Pastas a serem preenchidas:
resultados/HeapSort
resultados/InsertionSort
resultados/MergeSort
resultados/TreeSort

4 Descrição das Classes

Com o objetivo de organizar o código do trabalho, os arquivos foram divididos da seguinte forma:

Tabela 1: Descrição dos arquivos

Nome da Classe	**Descrição**
Livro	Definição do objeto Livro com todos os seus atributos
GerarArquivos	Classe responsável por gerar o arquivo datasetOBJ.txt e os arquivos de arrays
GerarArrays	A partir do datasetOBJ.txt gera arrays aleatórios além de ler os arquivos de arrays
Leitura	Responsável pelo pré-processamento do dataset
HeapSort	Implementação de todas as funções necessárias para executar a ordenação do tipo HeapSort
InsertionSort	Implementação de todas as funções necessárias para executar a ordenação do tipo InsertionSort
MergeSort	Implementação de todas as funções necessárias para executar a ordenação do tipo MergeSort
TreeSort	Implementação de todas as funções necessárias para executar a ordenação do tipo TreeSort
QuickSort	Implementação de todas as funções necessárias para executar os sequintes tipos de QuickSort: Recursivo, Mediana, Inserção

5 Pré-processamento do dataset

Antes de começar a executar as ordenações, o pré-processamento do dataset foi necessário já que muitas inconsistências foram encontradas.

Os principais problemas estão relacionados com os atributos description, title, edition-statement. Eles possuem vírgulas e aspas duplas no meio do dado, o que impede a identificação do começo e fim de algum atributo. Além disso existiam dados que ocupavam mais de duas linhas e no meio delas existiam linhas em branco. Por conta desse problema, alguns atributos começavam em uma linha e terminavam em outra.

A fim contornar esses problemas, um dado só é considerado como finalizado quando for preenchido todos os 25 atributos. Para os atributos gerais, o final de um atributo é contado somente quando for encontrado a sequência de aspas duplas + vírgula + aspas duplas.

Para description, title, edition-statement, foram feitos tratamentos especiais usando expressões regulares.

6 Resultados

6.1 Configurações do ambiente de testes

Cada cenário compara tipos de ordenação com vetores de diferentes tamanhos. Para cada tamanho de vetor, a ordenação é executada 5 vezes com dados aleatórios. Para que os resultados fossem precisos, é necessário que o mesmo vetor desordenado de tamanho N, usado em uma ordenação, fosse executado para outra ordenação. Afim de garantir essa equidade, foram usadas seeds variando de 1 à 5.

Os cenários extraem métricas com o objetivo de comparar cada ordenação. As métricas são: número de comparações de chaves, número de cópias de registro e o tempo total gasto para finalizar a ordenação.

Todos os resultados são dados pela média das 5 execuções em cada tamanho de vetor.

6.2 Cenário 1: Impacto de diferentes estruturas de dados

Neste cenário foi avaliado o desempenho do método de ordenação Quicksort Recursivo em diferentes estruturas de dados. Essa cenário foi dividido em duas partes. A primeira parte se tratou da execução da ordenação em vetores do tipo string com os seguintes

tamanhos: 1000, 5000, 10000, 50000 e 100000. O mesmo procedimento foi realizado na segunda parte, porém os vetores eram do tipo objeto.

A partir das Tabelas 2 e 3 apresentadas, é possível ver que o número de comparações de chaves e o número de cópias de registro foi exatamente o mesmo já que o método de ordenação não foi alterado, e que somente o tempo de execução da ordenação variou como pode ser visto na Figura 1. Além disso, percebeu-se que o tempo do vetor de objeto foi maior pelo fato de ser uma estrutura mais complexa em vista do vetor de string.

Tabela 2: Número de comparações nos testes para estrutura de dados

Número de comparações					
Tipo de vetor/Tamanho	1000	5000	10.000	50.000	100.000
Strings	7.475	49.584	105.183	627.977	1.425.969
Objetos	7.475	49.584	105.183	627.977	1.425.969

Tabela 3: Número de cópias nos testes para estrutura de dados

Número de cópias					
Tipo de vetor/Tamanho	1000	5000	10.000	50.000	100.000
Strings	2.592	15.619	33.599	195.729	411.643
Objetos	2.592	15.619	33.599	195.729	411.643

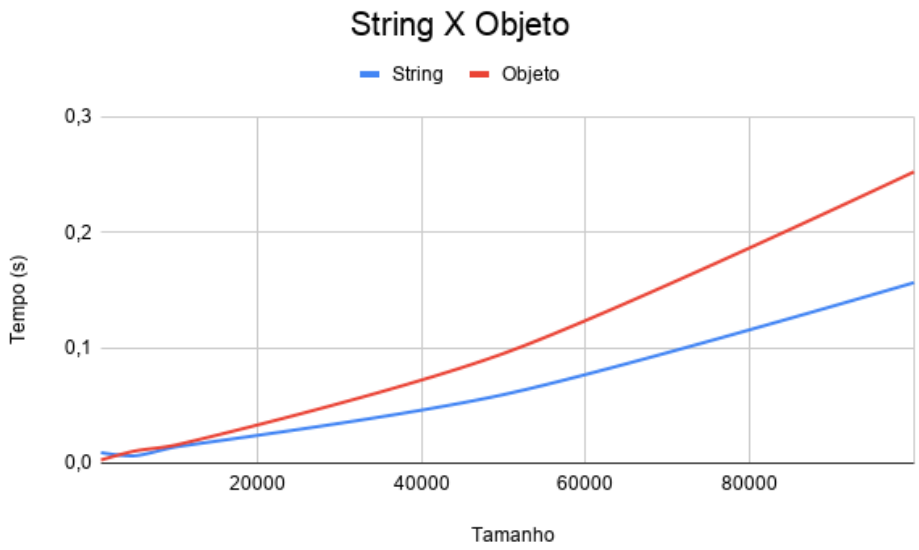

Figura 1: Gráfico que exibe tempo de execução dos vetores de string e objetos de acordo com o tamanho

6.3 Cenário 2: Impacto de variações do Quicksort

Para o cenário 2, foi comparado o desempenho de diferentes variações do QuickSort, ordenando um conjunto de N strings. As variações do QuickSort implementadas foram: QuickSort Recursivo, QuickSort Mediana e QuickSort Inserção. No QuickSort Mediana é escolhido um pivô para partição como sendo a mediana de k elementos do vetor, aleatoriamente escolhidos. Para análise foi escolhido k=3 e k=5. Já o QuickSort Inserção modifica o Quicksort Recursivo para utilizar o algoritmo de Inserção na ordenação das partições (isto é, pedaços do vetor) com tamanho menor ou igual a m. Para análise foi escolhido m =10 e m = 100.

Nos testes foram usados vetores de tamanho N = 1000, 5000, 10000, 50000, 100000 e 500000.

A partir dos gráficos apresentados nas Figuras 2, 3 e 4, podemos retirar as seguintes conclusões:

1. O QuickSort Mediana obteve o menor número de comparações mas o maior tempo: O QuickSort se beneficia muito de um pivô bem escolhido e, como a variação Mediana trabalha exatamente na escolha de um pivô melhor, o número de comparações acaba sendo menor do que as outras variações. Porém o custo dessa escolha acaba acarretando no tempo, já que o pivô é escolhido por uma função a parte. Também é possível dizer que, a medida que o k aumenta a chance de escolha de um bom pivô

também aumenta e a tendência é que o número de comparações diminua.

2. A Inserção teve os piores resultados nas métricas de cópias e comparações: O método de InsertionSort, por si só, já é uma ordenação inferior ao QuickSort. Era de se esperar que mesclar os dois métodos causaria uma piora no desempenho do QuickSort.

3. O Recursivo teve um desempenho mediano no número de comparações, mas em relação ao número de cópias e ao tempo gasto, foi o melhor: Essa variação, em nenhuma métrica, foi a pior entre as demais. Além disso, foi a única que obteve o melhor resultado em duas métricas. Portanto, a melhor variação do QuickSort é Recursivo.

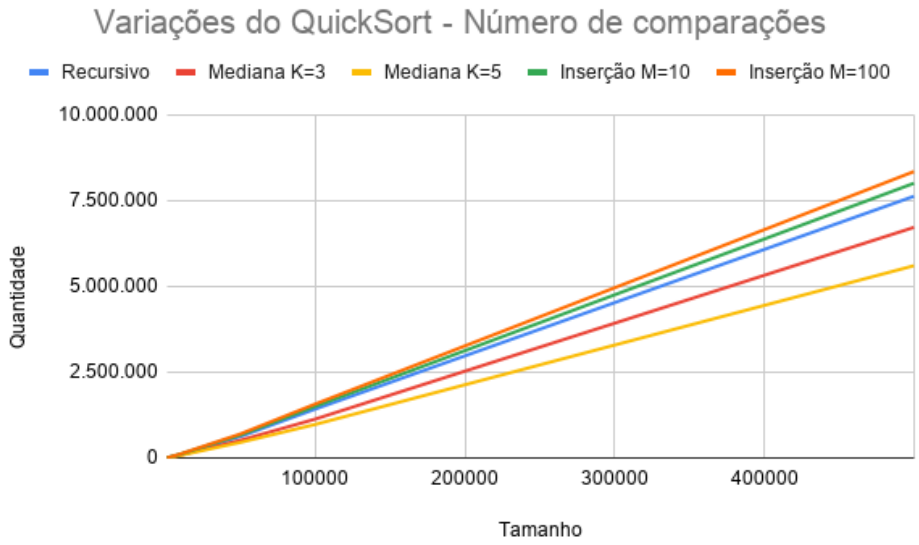

Figura 2: Análise do número de comparações entre as variações do QuickSort

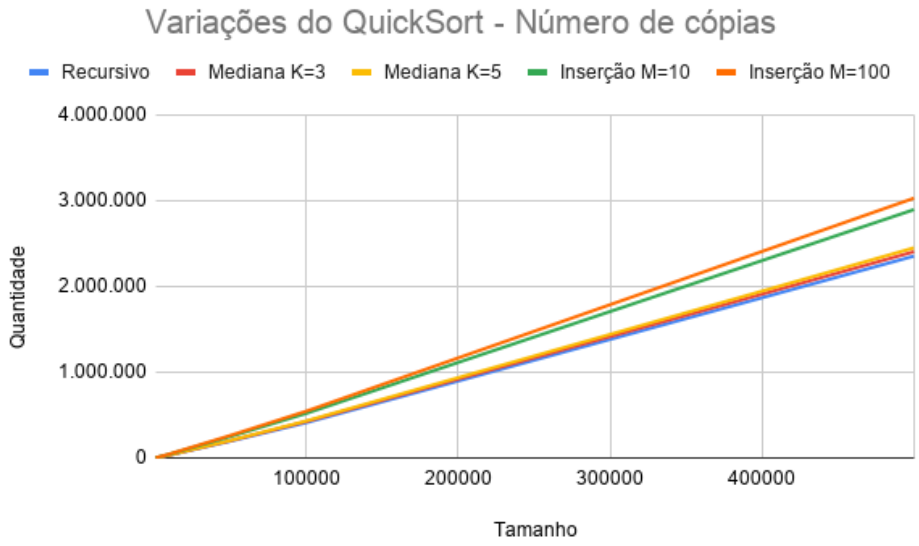

Figura 3: Comparativo do número de cópias entre as variações do QuickSort

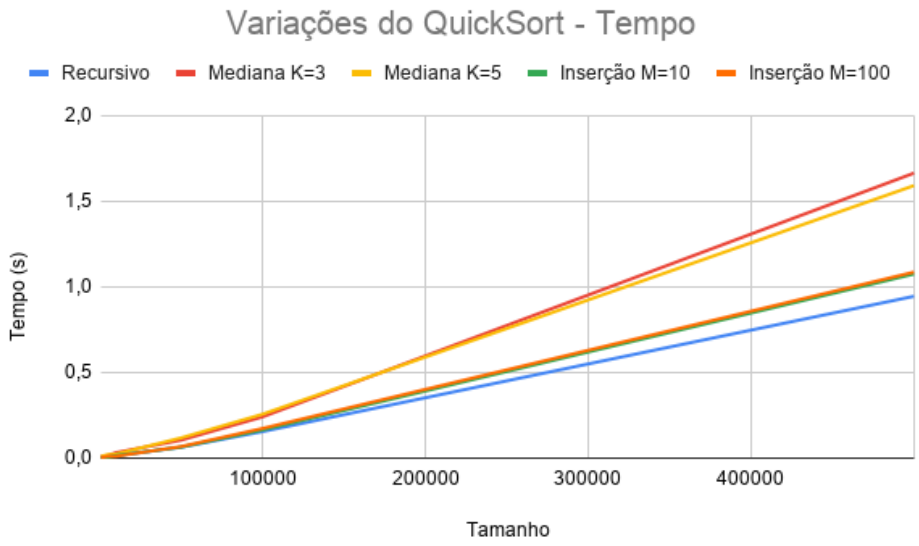

Figura 4: Comparativo do tempo gasto entre as variações do QuickSort

6.4 Cenário 3: Quicksort X InsertionSort X Mergesort X Heapsort

Neste cenário iremos analisar o desempenho do InsertionSort, MergeSort, HeapSort e o melhor resultado das variações do QuickSort, o Recursivo.

Alguns dos gráficos abaixo foram duplicados devido a discrepância dos resultados que acabavam impossibilitando a comparação deles.

A partir da Figura 5 é possível observar que dentre as 4 ordenações comparadas, o HeapSort obteve um maior número de comparações enquanto que o QuickSort Recursivo obteve os melhores resultados. Nota-se também que o MergeSort também obteve um ótimo resultado, quase equivalente ao Recursivo.

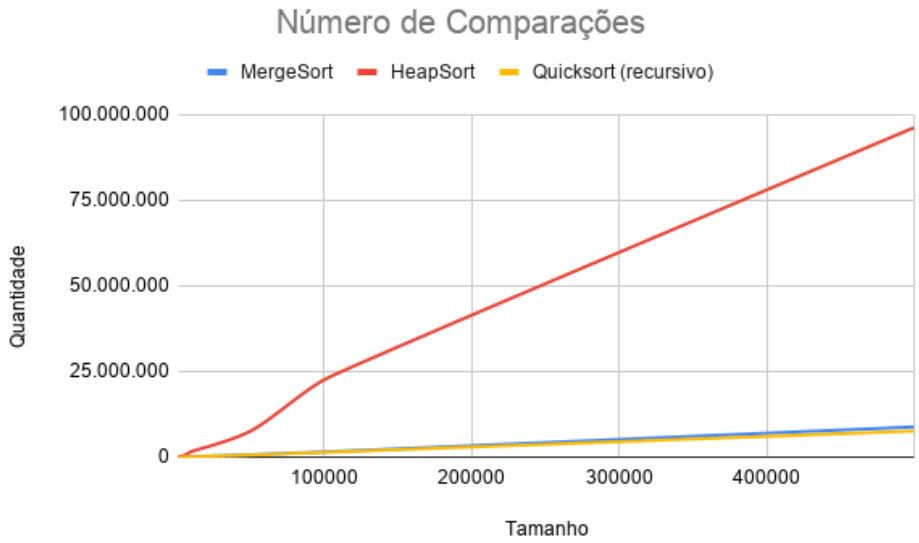

Figura 5: Análise do número de comparações, excluindo o InsertionSort

A Figura 6 mostra que o InsertionSort ficou na casa dos milhões. Esse resultado foi tão alto que impossibilitou a comparação com os outros algoritmos.

Sendo assim, o InsertionSort obteve o maior número de comparações em relação aos outros algoritmos e por isso foi a pior solução.

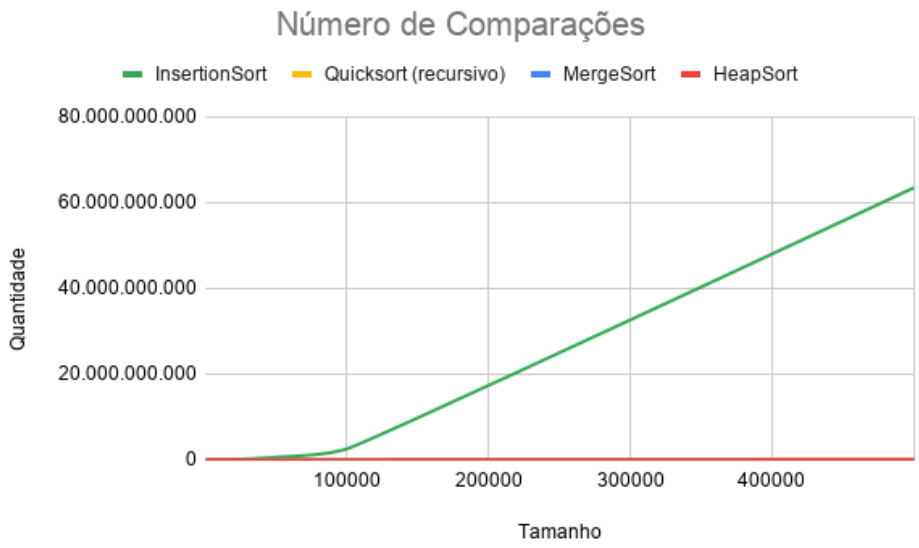

Figura 6: Análise do número de comparações do InsertionSort

Na análise da quantidade de cópias, os saltos dos valores de cada ordenação foram grandes, como é possível observar na Tabela 4.

Assim como aconteceu com o Insertionsort no número de comparações, o Mergesort, desta vez, alcançou números na casa dos trilhões. Sendo assim, ele foi o pior algoritmos em relação ao número de cópias.

O InsertionSort que obteve os piores resultados na métrica anterior, agora se saiu melhor.

A eficiência do QuickSort se sobressaiu dos demais, tendo o menor quantidade de cópias.

Tabela 4: Número de cópias dos algoritmos de ordenação analisados

Número de Cópias						
Tipo/Tamanho	1000	5000	10.000	50.000	100.000	500.000
Quicksort (recursivo)	2.592,00	15.619,40	33.599,20	195.729,40	411.643,20	2.353.423,40
HeapSort	27.230,00	216.450,80	703.078,00	3.162.527,20	9.358.288,80	39.561.716,00
InsertionSort	256.268,00	6.332.318,40	25.301.575,20	635.140.646,20	2.540.309.364,60	63.475.930.909,00
MergeSort	1.914.828,60	61.584.657,00	271.096.672,00	8.255.689.228,60	35.509.210.548,40	1.034.975.159.570,20

Do mesmo modo que fizemos com o número de comparações, tivemos que separar o InsertionSort dos demais algoritmos para não atrapalhar na visualização dos resultados de tempo, como pode ser visto na Figura 7 e 8.

Com isso, observou-se que, em questão de tempo de execução, o QuickSort obteve o menor tempo, enquanto que o InsertionSort obteve o maior.

Por mais que o MergeSort tenha tido o pior resultado no número de cópias, seu tempo quase equiparou com o QuickSort.

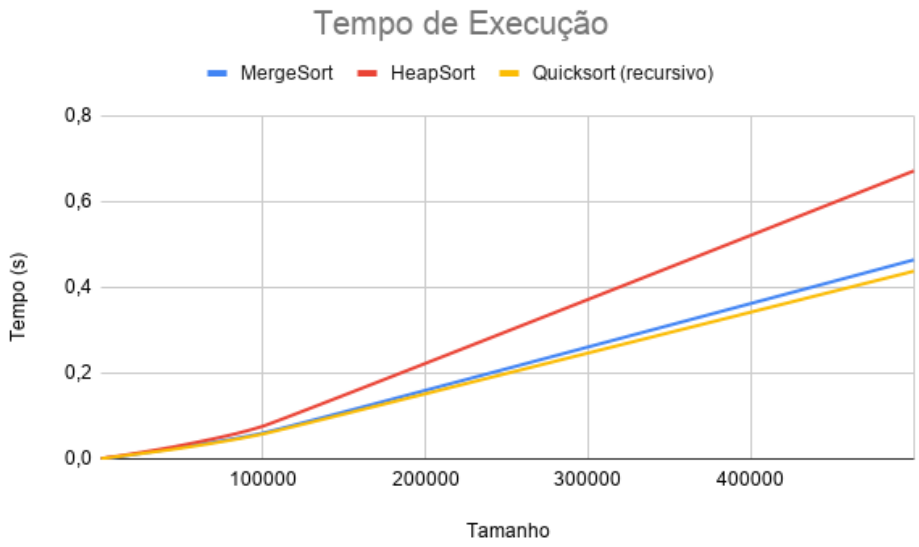

Figura 7: Análise do tempo de execução das ordenações, excluindo o InsertionSort

O InsertionSort, por sua vez, obteve o pior resultado novamente (Figura 8), apesar do baixo número de cópias feito. Provavelmente o que fez com que ele tivesse esse péssimo resultado foi a quantidade elevada de comparações.

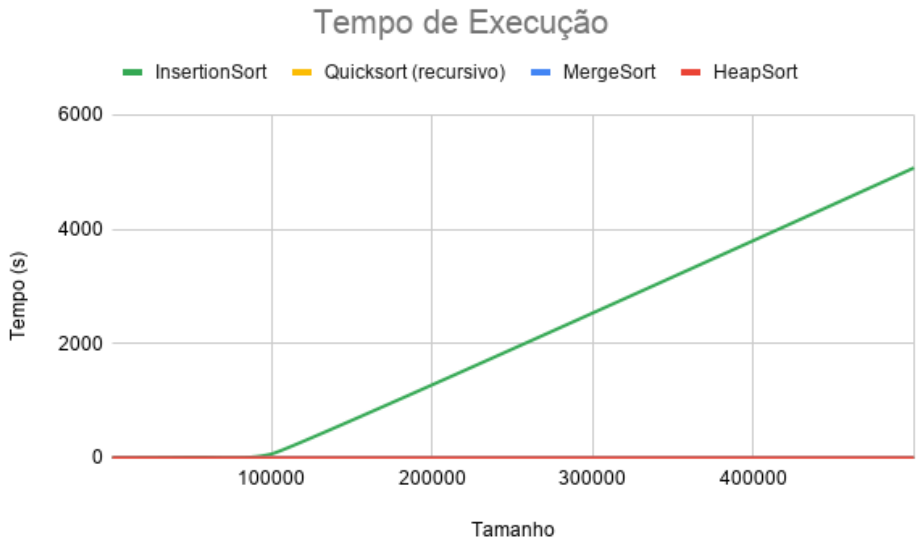

Figura 8: Análise do tempo de execução do InsertionSort

Diante disso podemos concluir que, o InsertionSort foi a pior solução dentre todos os outros algoritmos de ordenação. Por mais que em alguns casos ele tenha conseguido um bom resultado, ainda assim não é a melhor solução para entradas muito grandes.

6.5 Trabalho Atual: QuickSort X MeuSort

Neste cenário definimos o algoritmo TreeSort como o de nossa escolha e o comparamos com a melhor variação do QuickSort, o QuickSort Recursivo.

Tabela 5: Número de comparações

	Média MeuSort	**Média QuickSort**
1000	8461	7475
5000	50675	49584
10.000	107822	105183
50.000	629923	627977
100.000	1315831	1425969
500.000	7606309	7628432

Evidenciado pela Tabela 5, os algoritmos MeuSort e QuickSort usam de sistema de comparações de eficiência próxima, logo este não pode ser um dos critérios para decisão de escolha entre esses algoritmos.

Tabela 6: Número de cópias

	Médias MeuSort	**Médias QuickSort**
1000	1999	2592
5000	9986	15619
10000	19956	33599
50000	99150	195729
100000	197108	411643
500000	953481	2353423

Como é possível observar na Tabela 6, o algoritmo MeuSort utiliza de espaço auxiliar pouco menor que 2n, enquanto que o QuickSort utiliza de espaço auxiliar pouco menor que nLogn, onde n é a quantidade de elementos a serem ordenados. Dessa forma, quando é visado o menor consumo de memória, o algoritmo MeuSort deve ser o escolhido.

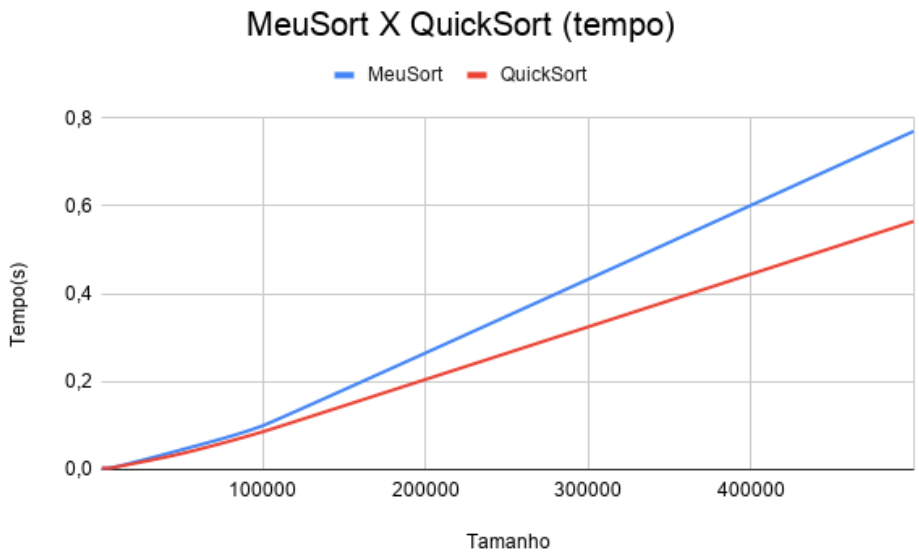

Figura 9: Gráfico que exibe tempo de execução dos algoritmos QuickSort recursivo e MeuSort.

Levando em consideração o gráfico da figura 9, o algoritmo QuickSort apresentou tempo de execução até 36,6% melhor que o algoritmo MeuSort, dessa forma, quando o objetivo é o melhor tempo de execução, deve-se escolher o algoritmo QuickSort.

7 Parte 2:

7.1 Tabelas Hash

Com o intuito de conseguir analisar todo o dataset, na parte 2 uma nova estrutura de dados, chamada LivroAux, foi criada para reduzir as informações contidas para apenas dois atributos, id e autores.

Ademais, foram implementadas duas estruturas do encadeamento coalescido. Esse método foi usado pois, além de possuir o tamanho exato da tabela, foi vista a necessidade de uma ocupação uniforme. Para o mapeamento das chaves foi utilizado o método da divisão.

Com isso, a primeira e a segunda tabela continham, em cada posição, uma estrutura de autores e outra de livros,respectivamente. A estrutura do livro contém um vetor de autores que, a partir desse id, percorre a estrutura de cada autor daquele livro em específico.

Para analisar os resultados, no final da execução o programa imprime os dez autores mais frequentes e, além disso, para a ordenação, foi escolhido o método QuickSort uma vez que apresentou um desempenho bom nos cenários demonstrados acima. Adicionalmente, um arquivo é gerado em "resultaods/Parte2/ranking_de_autores.txt"que contêm o ranking de todos os autores presentes. O resultado dos autores mais frequentes pode ser visto na figura 10.

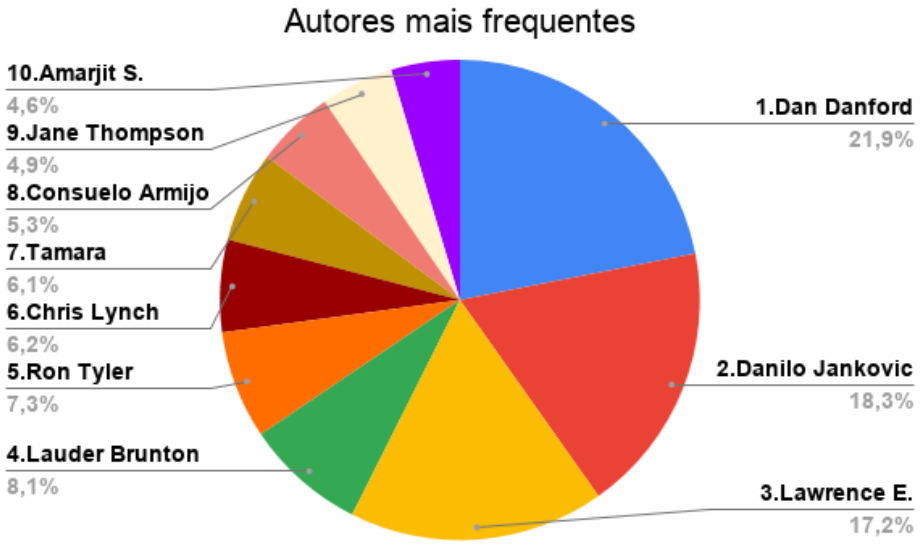

Figura 10: Gráfico que exibe os autores mais frequentes usando o método QuickSort para ordenar

Vale observar que alguns autores não foram encontrados na tabela de autores.

8 Parte 3:

Nesta fase, foram criadas 3 estruturas de Árvore: uma Árvore B, com o tamanho mínimo 2; outra Árvore B, com o tamanho mínimo 20; e uma Árvore Vermelho e Preto. As estruturas das Árvores estão localizadas na pasta Arvores. Cada arvore é composta de dois arquivos: um com a estrutura do nó e outra com a estrutura da árvore propriamente dita.

Para todo N de entrada, foram realizadas N operações de inserção e N operações de busca em cada árvore. Foi analisada o desempenho dessas estruturas. Para isso foram geradas seeds variando de 1 à 5 e os valores de N = 1000, 5000, 10000, 50000, 100000 e 500000. A partir desses elementos, foram geradas as árvores considerando as estatísticas e armazenando os resultados no arquivo "resultados/Parte3/tamanho do vetor/saidaInsercao.txt"e "resultados/Parte3/tamanho do vetor/saidaBusca.txt".

8.1 Operações de inserção

Primeiro foi analisado a operação de inserção. As métricas examinadas foram: número de comparações, número de cópias e tempo de execução.

A partir dos gráficos apresentados nas Figuras 11, 12, 13, podemos tirar as seguintes conclusões sobre a inserção:

1. Algo que chamou atenção foi o fato de a Árvore B com d=20, apesar de apresentar a maior quantidade de cópias de registro e comparações devido ao volume de chaves contidas em cada nó é maior, teve tempo de execução consideravelmente menor do que as Árvores B com d=2 e Vermelho e Preto. Esse fato acontece pois a quantidade de overflows é menor na Árvore B com d=20, e o custo da operação para corrigir o overflow é grande.

2. No geral, em relação ao número de comparações, a Árvore B 2 teve o melhor resultado. No número de cópias a Árvore Vermelho e Preto se saiu melhor. Já no tempo de execução a Árvore B 20 foi mais rápida.

3. Em cada métrica foi observado que um tipo de Árvore diferente se saiu melhor. O que reforça a ideia de que, dependendo de qual o objetivo da aplicação, deve-se usar uma Árvore diferente.

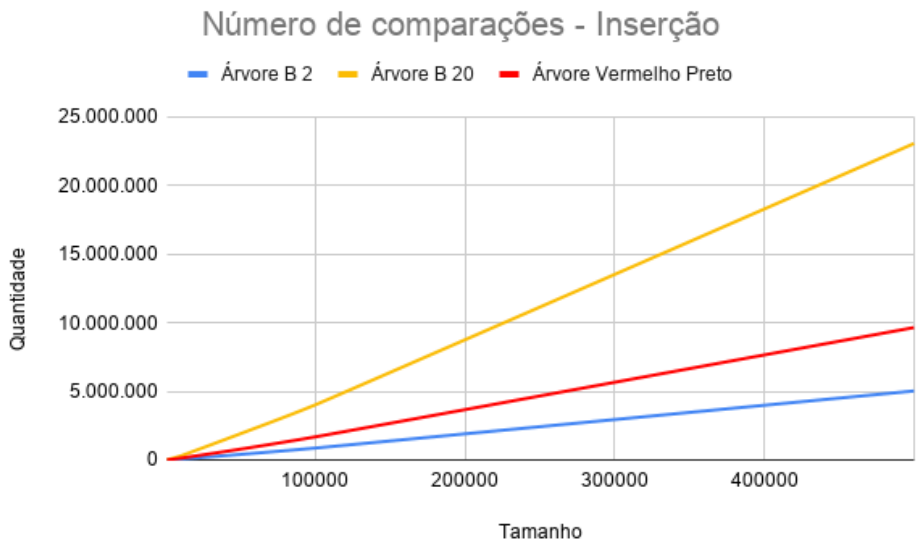

Figura 11: Gráfico que exibe os números de comparações feitos pelas operações de inserção das Árvores.

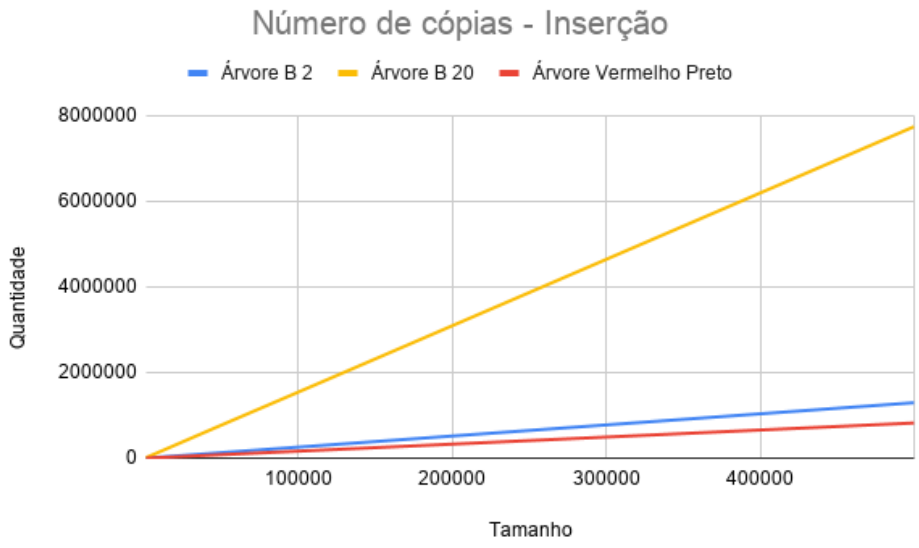

Figura 12: Gráfico que exibe os números de cópias feitos pelas operações de inserção das Árvores.

Figura 13: Gráfico que exibe os tempos de execução das operações de inserção das Árvores.

8.2 Operações de busca

Nas operações de busca não é possível analisar o número de cópias visto que, a estrutura de nenhuma das Árvores analisadas é alterada ao realizar buscas, portanto foi considerado apenas o número de comparações e o tempo de execução.

A partir dos gráficos apresentados nas Figuras 14, 15, podemos tirar as seguintes conclusões sobre a busca:

1. Na a Árvore B com o d=20, o número de comparações foi maior devido ao grande volume de chaves, portanto antes de descer na árvore, cada chave do nó é comparado. Já nas árvores Vermelho-Preto e na B com d=2, o volume de comparações é menor pois a quantidade de chaves em cada nó é pequena.

2. O tempo de busca foi maior na Árvore B com o d=2 visto que a busca é realizada em muitas descidas na árvore, o que tornou o processo muito custoso em vista da B com d=20.

3. As buscas na Árvore Vermelho e Preto foram mais rápidas que a Árvore B 2 porque a rubro-negra é uma árvore de busca binária.

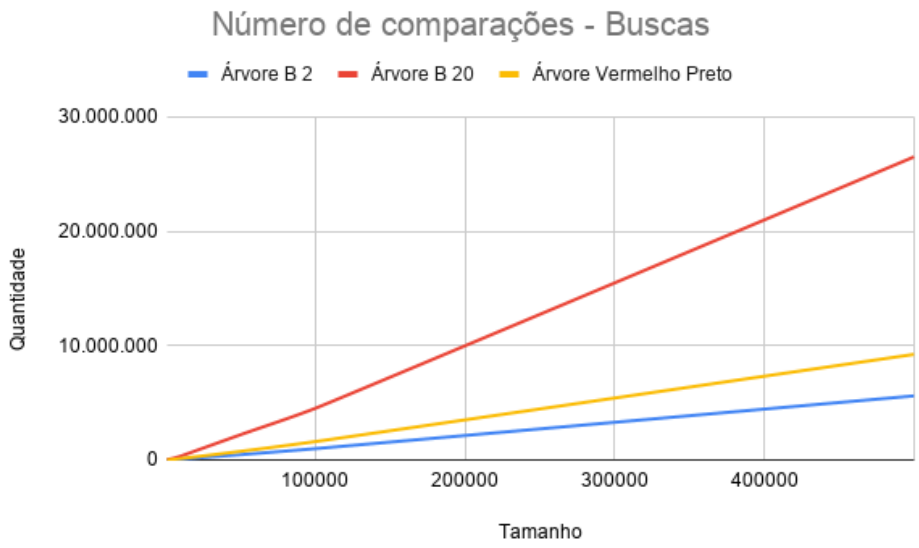

Figura 14: Gráfico que exibe os números de comparações feitos pelas operações de busca das Árvores.

Figura 15: Gráfico que exibe os tempos de execução das operações de busca das Árvores.

9 Divisão do trabalho:

9.1 1º Parte:

Ana Beatriz Kapps: Implementação de funções de ordenação e escrita do relatório
Felipe Israel: Implementação de funções de ordenação e escrita do relatório
Ian Couto: Implementação de funções de ordenação e escrita do relatório
Rodrigo Torres: Tratamento do dataset e escrita do relatório

9.2 2º e 3º Parte:

Nessas partes do trabalho o desenvolvimento foi feito em conjunto, ou seja, todos estavam reunidos na implementação e contribuíram para o progresso, sem uma divisão explicita.