Laboratório de Estrutura de Dados

Segunda versão do projeto da disciplina

Comparação entre os algoritmos de ordenação elementar e implementação de estruturas de dados dinâmicas

IAN DINIZ DE OLIVEIRA.
LUKAS CHRISTOPHER DE SOUZA SANTANA.
NICOLAS MARTINS DE MOURA.

1. Introdução

Esta é a segunda parte do projeto da disciplina de “laboratório de estrutura de dados”. O projeto consiste na implementação de estruturas de dados dinâmicas como forma de melhorar o desempenho e simplificar o código anteriormente desenvolvido na primeira parte do projeto. Em suma, o projeto consiste na implementação das nossas próprias estruturas de dados dinâmicas “hash map”, “lista dinamica”, “linked list” e “pilhas”. Essas estruturas foram desenvolvidas e implementadas com forma de substituir os “arrays” convencionais.

Após implementarmos essas estruturas rodamos novamente os algoritmos de ordenação propostos a fim de testar o desempenho dos mesmos após a implementação das estruturas.

O projeto escolhido foi o “Los Angeles Metro Bike Share”, um sistema de compartilhamento de bicicletas na cidade de Los Angeles, Califórnia e área metropolitana. O sistema usa uma frota de cerca de 1400 bicicletas e inclui mais de 93 estações na região denotada de trabalho do sistema. Os dados referentes ao registro de locações das bicicletas controladas pelo sistema entre os anos de 2016 e 2021, são disponibilizados em um arquivo no formato CSV.

A primeira etapa do projeto consiste em realizar algumas transformações nos arquivos *stations.csv* e *LA_Metro_BikeSharing_CLEANED_2016quater3-2021q3.csv.* Essas transformações são:

1. Gerar um arquivo único chamado LAMetroTrips.csv substituindo o id das estações pelo nome (campo station_name) que está contido no arquivo (stations.csv).

2. Considerando o arquivo gerado no Item 1 (LAMetroTrips.csv), filtrar apenas as viagens que estão nas estações de Pasadena. Gerar um arquivo chamado LAMetroTrips_F1.csv

3. Considerando, o arquivo gerado no Item 1 (LAMetroTrips.csv), filtra apenas as viagens que possuem duração maior que a média geral.

Para as transformações, foram utilizadas algumas importações da biblioteca padrão do java. Para a leitura e escrita dos arquivos utilizamos as classes BufferedReader, FileReader. FileWriter e File. Todas necessitam de apenas alguns segundos para carregar o arquivo no *array* principal de forma que possamos realizar as transformações e criar os novos arquivos. As classes responsáveis pela modificação do arquivo base podem ser encontradas no arquivo do projeto. São elas as classes: P0ConsertarDatas, P1SubstituirId, P2FiltrarStationName e P3FiltrarMedia.

Seguindo com a segunda parte do projeto, foi requisitada a ordenação dos dados do arquivo gerado no item 1 - LAMetroTrips.csv. Os algoritmos usados foram: Selection Sort, Insertion Sort, Merge Sort, Quick Sort, QuickSort com Mediana de 3, Counting Sort e HeapSort. Por fim, deve-se gerar um arquivo para cada algoritmo de ordenação e o tipo de caso(melhor, pior e médio).

As ordenações exigidas são as seguintes:

1. Ordenar o arquivo completo pelo nomes das estações (campo station_name/ station_id) em ordem alfabética.

2. Ordenar o arquivo LAMetroTrips.csv pelo campo de duração da viagem (campo duration) do menor para o maior

3. Ordenar o arquivo LAMetroTrips.csv pela data de início da viagem (campo Start_time) mais recente para o mais antigo.

Os resultados sobre tempo de execução e uso de memória dos algoritmos podem ser visualizados nos seguintes gráficos:

2. Alterações realizadas

2.1 MeuConjuntoDinamico.java

A primeira alteração realizada foi a implementação de um algoritimo de lista dinamica, o "**MeuConjuntoDinamico.java**", esse algoritimo permite a insersão de dados sem a preocupação com o tamanho do "array" uma vez que após delimitarmos o tamanho de um "array" não podemos mais aumentar ou diminuir a quantidade de elementos. Também temos que nos preocupar com a quantidade total de elementos que vamos inserir no array para poder alocar a quantidade certa de memória para podermos trabalhar.

Com a implementação da lista dinâmica conseguimos enxugar bastante a classe responsável por abrir os arquivos. Antes da implementação da lista dinâmica era necessário todo um trecho de código para obter o tamanho do arquivo, isto é, a quantidade de linha para podermos delimitar o tamanho do "array". Com a implementação do conjunto dinâmico não precisamos mais nos preocupar com isso, oque permitiu apagar todo esse trecho de código. Contudo Essas alterações não vem da graça, o programa acabou perdendo um pouco de despenhou na hora de ler os dados do arquivo, já que ele terá de realizar suas interações toda vez que "array" da class "**MeuConjuntoDinamico.java**" estiver cheio, sendo nessesairo criar um novo "array", maior, e copiar todos os elementos do array antigo para o novo.

2.2 MinhaPilha.java

Durante o desenvolvimento da primeira versão do projeto tivemos um problema com o algoritmo QuickSort, ambas as versões com mediana de 3 e sem mediana não funcionavam para o melhor e pior caso, isso ocorria por que o QuickSort é um algoritmo recursivo que depende da escolha de um bom pivô para ser eficiente. Quando estamos trabalhando com massas de dados que estão muito desordenadas o QuickSort se sai mais eficiente já que ele escolhe melhor o pivô. Contudo, como as massas de teste para pior e melhor caso estavam ordenadas, o algoritmo escolhe maus pivôs, sendo necessário realizar muitas chamadas reativas o'que acarretava em stackoverflow.

A solução encontrada para o problema foi implementar uma versão iterativa do QuickSort, a fim de minimizar a quantidade de chamadas reativas. Para isso, tivemos que construir uma estrutura de dados de pilha, a "**MinhaPilha.java**", que guarda parte dos

dados para a atenção, evitando muitas chamadas e chuvas e impedindo o estouro de memória.

Após a implementação desse algoritmo foi observado um aumento total de desempenho no QuickSort M3 que passou a funcionar para todas as massas de dados com um tempo médio de 12 segundos para todos os casos, gastando um pouco mais de memória. Em contrapartida, o desempenho do QuickSort normal piorou levando horas para ordenar o melhor e pior caso.

2.3 MyHashMap.java

Na primeira versão do projeto criamos uma pequena, ou melhor, uma grande interface via terminal para selecionar o tipo de dado a ser ordenado, o tipo de caso (melhor, médio e pior) e o tipo de algoritmo utilizado. Essa interface foi implementada para os casos de “station” e “duration”, devido às complicações oriundas da utilização de apenas array simples tivemos dificuldades para criar um código limpo e bem organizado para a interface. Graças ao “**MyHashMap.java**” conseguimos limpar o código e fazer uma interface elegante e funcional, que praticamente não possui quaisquer estruturas condicionais. A classe “**GerarArquivosOrdenados.java**” fica responsavel por garanti a seleção do elementos e gerar o arquivo com o nome apropriado, esse classe possui divesços objetos “**MyHashMap.java**”, que contribuiem para limpeza e clareza do código, a classe possui pouco mais de 100 linhas de código enquanto as classe antigas possuíam mais 300 linhas cada.

3. Descrição geral sobre o método utilizado

Para o projeto foi necessário utilizarmos três computadores pessoais referentes a cada integrante do grupo. Cada integrante do grupo ficou responsável por realizar uma das tarefas de manipulação do arquivo principal bem como cada um ficou com a tarefa de implementar sua própria versão dos algoritmos de ordenação já citados. O programa desenvolvido para o projeto abre, altera e cria novos arquivos. Foi construída a classe “CsvFile.java”, responsável pela função de abrir e ler todos os arquivos e colocar seus dados na lista dinâmica. Com o arquivo carregado na lista dinâmica podemos utilizar a classe “CsvMetods.java” para obter uma coluna específica do arquivo principal, também podemos utilizar essa classe para podermos alterar toda uma coluna do arquivo principal.

Agora que temos como abrir e selecionar uma coluna específica do arquivo podemos focar no processo de ordenação. Foi criada uma nova classe a fim de deixar o código mais limpo. A classe “SortAlgorithm” é responsável por ordenar e salvar os arquivos. Com essa classe podemos definir o tipo de dado a ser ordenado, o arquivo e o tipo de algoritmo de ordenação.

Descrição geral do ambiente de testes

- Especificações do dispositivo utilizado na ordenação por Data:
  CPU: Intel(R) Core(TM) i3-2310M CPU @ 2.10GHz
  Ram: 6GB DDR3
  Sistema: Ubuntu 23.04

- Especificações do dispositivo utilizado na ordenação por Nome da estação:
  CPU: Athlon 3000g 3.50 GHz
  Ram: 8GB DDR4
  Sistema: Windows 10

- Especificações do dispositivo utilizado na ordenação por duração da viagem:
  CPU: Xeon E5-2650 V2
  Ram: 16GB DDR3
  Sistema: Windows 10

- Especificações do dispositivo utilizado na ordenação por Duração da viagem:

4. Resultados e Análise

4.1 Merge Sort e Heap Sort

Os algoritmos de Merge e Heap, mantiveram o mesmo patamar de eficiência. O'Que tivemos de alteração graças às estruturas de dados dinâmicas foi no QuickSort M3 que passou a ordenar as três massas de dados, em poucos segundos. Os algoritmos de complexidade O(n^2) mantiveram sua complexidade, afinal não é possível implementar uma estrutura de dados que possam diminuir a complexidade desses algoritmos. Logo não faz sentido aplicar quaisquer estruturas de dados dinâmicas para aumentar sua eficiência, oque podemos fazer com essas estruturas é limpar o código e deixá-lo mais limpo, portanto não tivemos grandes mudanças de desempenho desses algoritmos.

As leves quedas de desempenho que podem ser observadas nesses algoritmos se devem pela implementação da lista dinâmica na hora de carregar os dados e converter os dados da lista dinâmica em array. Como a lista dinâmica crescer o seu tamanho conforme entrada de mais dados, é necessário de tempos em tempos expandir o array interno da lista, e copiar todos os elementos do array antigo para o novo. Acarretando em uma leve perda de eficiência.

4.2 Counting Sort

O Counting Sort não teve grandes alterações em suas estruturas, assim como o Meger e o Heap Sort. Sua perda de eficiência se deve pelas questêos abordadas no tópico anterior.

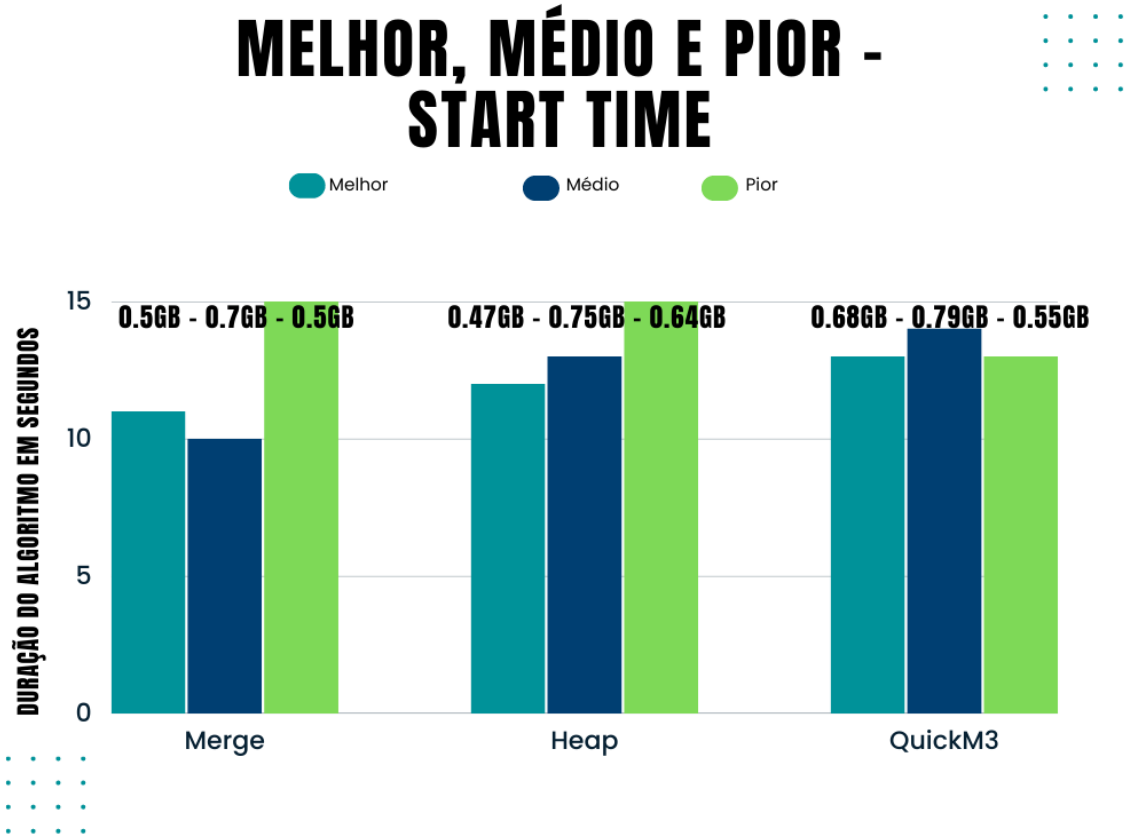
4.3 Quick e Quick Sort M3

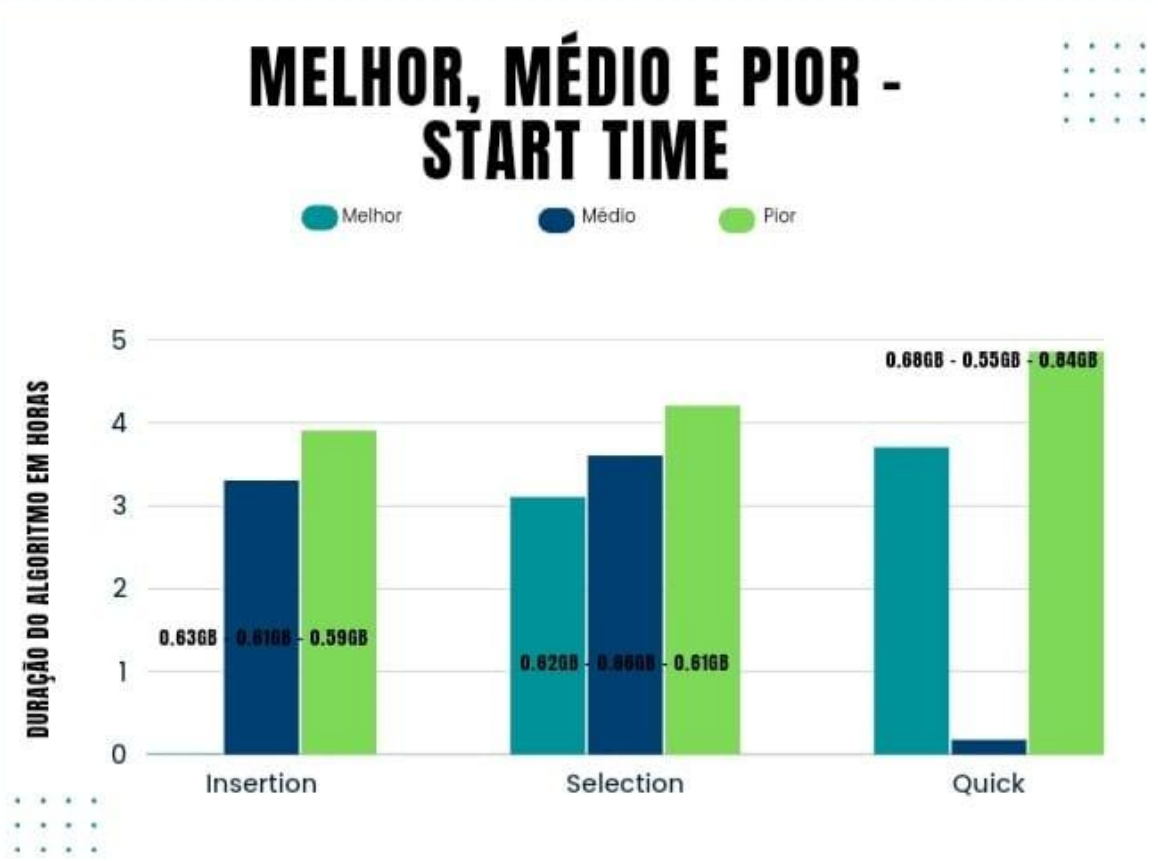
Ambos os algoritmos de Quicksort sofreram modificações decorrentes da implementação de uma estrutura de dados pilha. Essa alteração garante o funcionamento dos algoritmos para o pior e o melhor caso. No mais, o desempenho de ambos os algoritmos se mantiveram semelhantes aos algoritmos sem a pinha.

4.4 Selection Sort e Insertion Sort

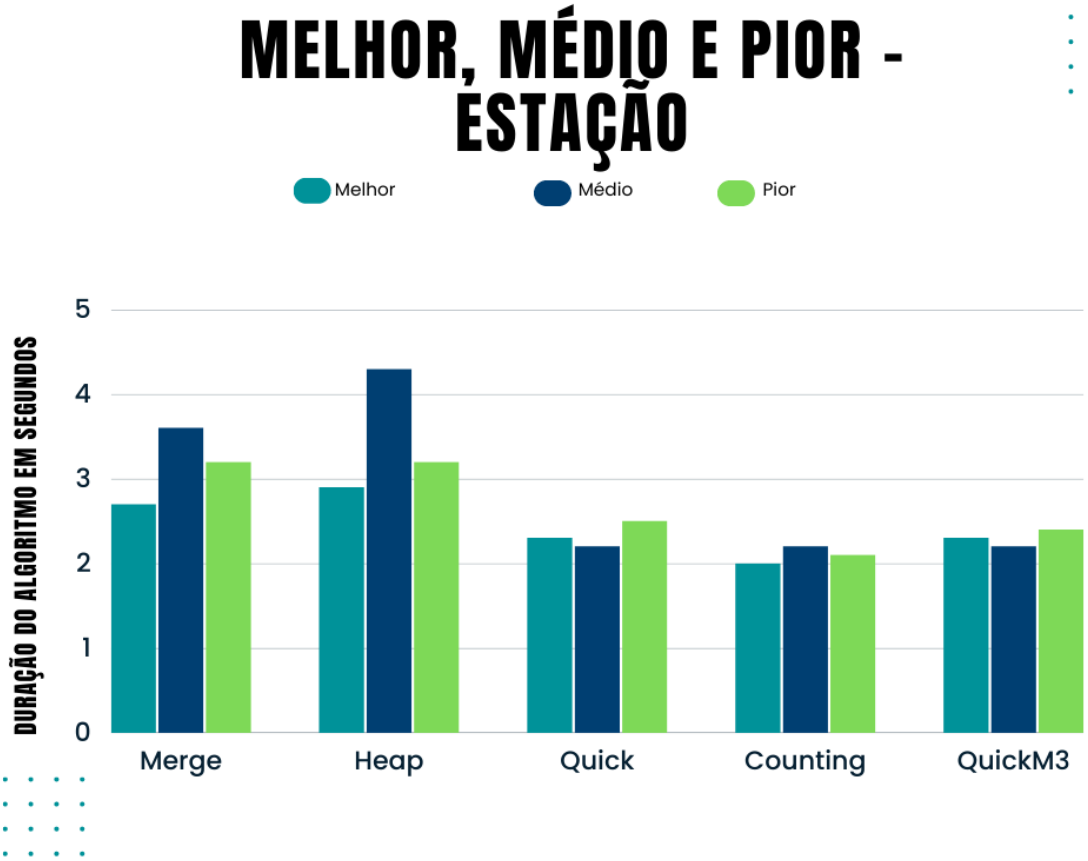
Os algoritmos Selection e Insertion Sort continuam sem grandes alterações de desempenho, por mais que existam outras implementação desses algoritmos com listas encadeadas e listas dinâmicas, contudo não faz sentido implementar essas estruturas nesses algoritmos pois suas complexidade continuará a mesma, podendo até piorar em desempenho. No mais, não tivemos grandes diferenças de tempo e consumo de memória.

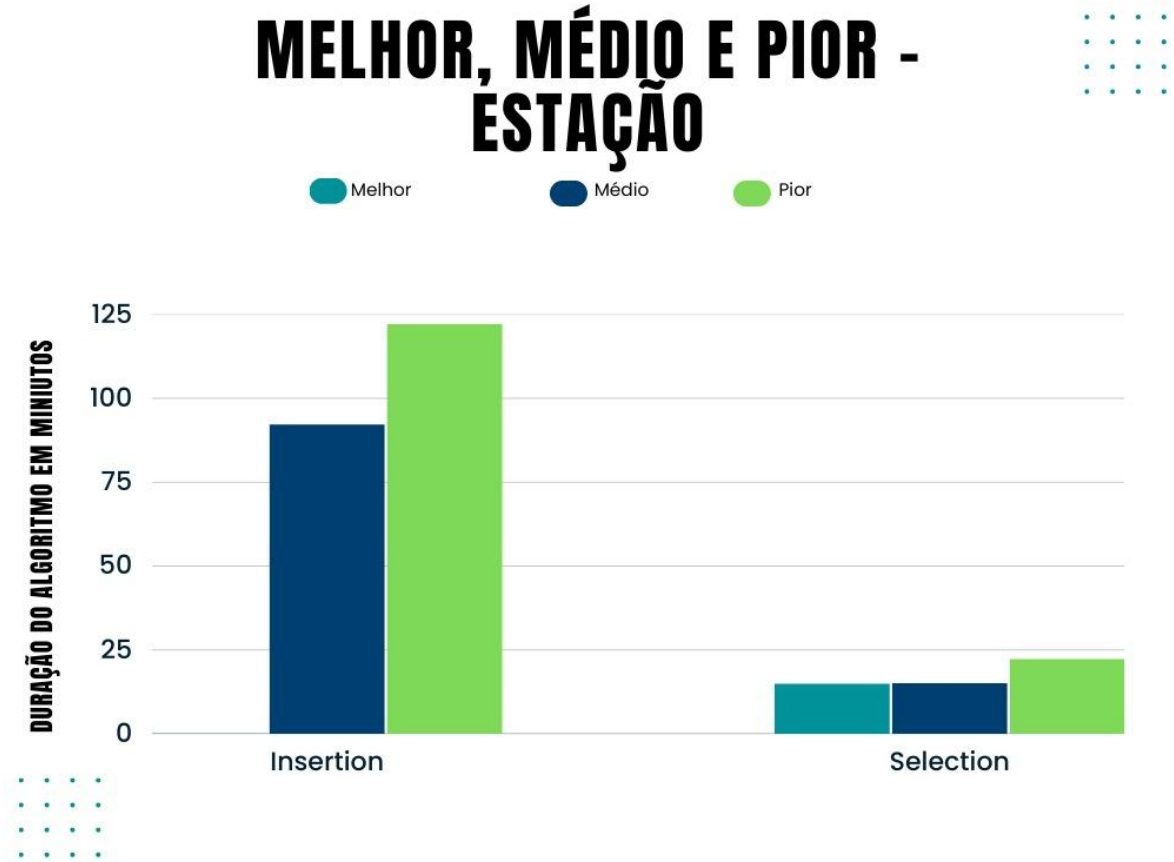
Start Time - Tempo de execução e Consumo de Memória





Estação - Tempo de Execução e Consumo de Memória





Duração- Tempo de Execução e Consumo de Memória