UEPB
Universidade Estadual da Paraíba
Departamento de Computação – DC
Laboratório de Estrutura de dados
Professor: Fábio Leite

KAYLLANE ARAÚJO DOS SANTOS

MARIA CLARA CUNHA PAULINO

TARCIO ELYAKIN AGRA DINIZ

ANÁLISE COMPARATIVA DE ALGORITMOS DE ORDENAÇÃO

BOLSA DE VALORES BOVESPA 1994-2020

Segundo Projeto de Laboratório de Estrutura de Dados

CAMPINA GRANDE – PB

20 de novembro de 2024

KAYLLANE ARAÚJO DOS SANTOS

MARIA CLARA CUNHA PAULINO

TARCIO ELYAKIN AGRA DINIZ

ANÁLISE COMPARATIVA DE ALGORITMOS DE ORDENAÇÃO BOLSA DE VALORES BOVESPA 1994-2020

Segundo Projeto de Laboratório de Estrutura de Dados

Relatório apresentado no curso de Ciência da Computação da Universidade Estadual da Paraíba e na disciplina de Laboratório de Estrutura de dados referente ao período 2024.2

Professor: Fábio Leite

CAMPINA GRANDE – PB

20 de novembro de 2024

SUMÁRIO

1. INTRODUÇÃO

Neste relatório, teremos a continuidade do primeiro projeto, relacionado a bolsa de valores BOVESPA, de 1994 a 2020, onde foi realizado uma análise comparativa dos diferentes algoritmos de ordenação. Os algoritmos utilizados durante o primeiro estudo foram: Selection Sort, Insertion Sort, Merge Sort, Quicksort, Counting Sort e Heap Sort.

Nesta fase atual, o projeto foi evoluindo para a implementação de três estruturas de dados além dos arrays, essa evolução teve como objetivo principal melhorar o desempenho e eficiência dos algoritmos de ordenação.

2. METODOLOGIA

Este estudo foi realizado utilizando a linguagem Java para todos os algoritmos. No presente projeto, houve a implementação de três novas estruturas de dados — Árvore de Busca Balanceada (AVL), Tabela Hash e Fila — que foram fundamentais para garantir a eficiência, escalabilidade e versatilidade nas operações de transformação, filtragem e ordenação sobre um grande conjunto de dados. Cada uma dessas estruturas desempenha um papel essencial no processamento e organização dos dados, permitindo abordar diferentes necessidades e desafios do sistema.

A utilização de uma árvore AVL se destaca pela capacidade de armazenar dados de maneira ordenada, mantendo o equilíbrio estrutural e garantindo operações eficientes de busca, inserção e remoção, todas com complexidade O(log n).

A Tabela Hash foi implementada com foco na eficiência de busca direta e na manipulação de grandes conjuntos de dados com identificadores únicos. Sua principal vantagem é a complexidade média O(1) para operações de busca, inserção e remoção.

Já a Fila foi incorporada ao projeto como uma estrutura essencial para o processamento sequencial de dados, especialmente em cenários que exigem controle sobre a ordem de entrada e saída. Essa estrutura foi utilizada para implementar algoritmos de processamento em lote e simulação de fluxo, explorando sua característica de FIFO (First-In, First-Out).

Para a confecção dos gráficos, foi utilizada a linguagem Python para a implementação, e os códigos foram executados por meio do Google Colab.

Logo abaixo, temos as especificações da máquina utilizada para realizar as operações:

Tabela 01: Características Principais do Notebook Dell G15 5530

Placa mãe	Dell Inc 0048WH A00 Processador 12th
Processador	13th Gen Intel(R) Core(TM) i5-13450HX
Memória RAM	16GB
Armazenamento	SSD de 500gb
Placa de Vídeo	Intel(R) UHD Graphics
Sistema Operacional	Windows 11 home

Fonte: Autoria Própria

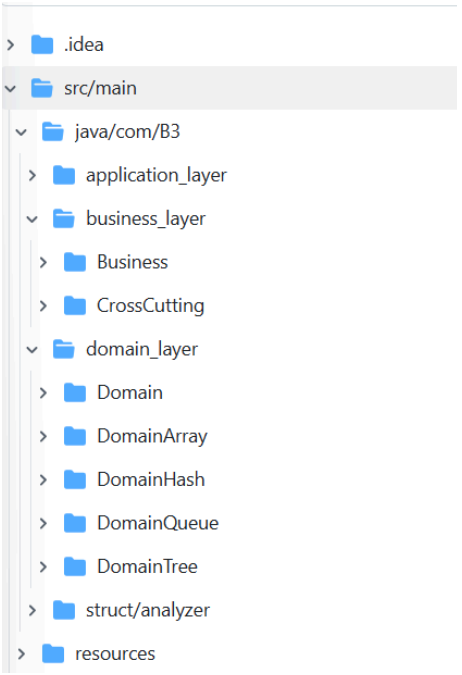
2.1 APLICAÇÕES E JUSTIFICATIVAS

Neste segundo projeto, foi implementado uma nova arquitetura, inserindo um padrão conhecido por DDD (Domain-Driven Design), de modo que foram inseridas três pastas para a nova arquitetura, sendo elas:

1. **Application Layer:**
   - Contém o ponto de entrada (Main) e controla o fluxo da aplicação.
2. **Business Layer:**
   - Contém os serviços e repositórios responsáveis pelas operações lógicas.
3. **Domain Layer:**
   - Define contratos, modelos (Dtos) e interfaces que garantem abstração e separação de responsabilidades.

A escolha dessa arquitetura foi baseada no fato de ser considerada ideal para sistemas que lidam com grandes volumes de dados, devido à sua modularidade e ao foco no domínio do negócio. Com isso, deixamos de utilizar a Struct Analyzer.

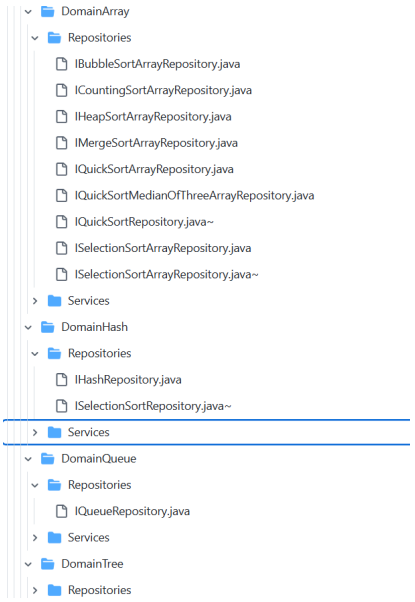
Figura 01 - Arquitetura DDD



Fonte: Autoria Própria

Logo abaixo, será exibido uma figura com os nomes das novas interfaces e objetos que foram implementadas nessa segunda etapa.

Figura 02 - Interfaces e objetos que são utilizados na aplicação

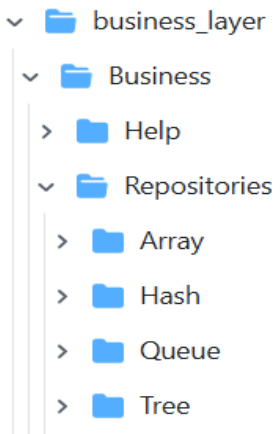


Fonte: Autoria Própria.

2.1.1 Camadas

Na camada domain_layer colocamos toda interface e objetos que podemos usar na aplicação, incluindo as novas interfaces que foram implementadas. A camada de domínio é o núcleo do Domain-Driven Design, a mesma é responsável por modelar e encapsular o domínio do negócio, basicamente, ela representa as regras, conceitos e processos fundamentais do sistema. Além da domain_layer, temos a camada business, camada responsável por implementar as regras de negócio e a lógica central do sistema, é nessa camada que reside de fato a implementação de cada estrutura de dados correspondente, sendo elas: Array, tabela Hash, filas e árvores de busca.

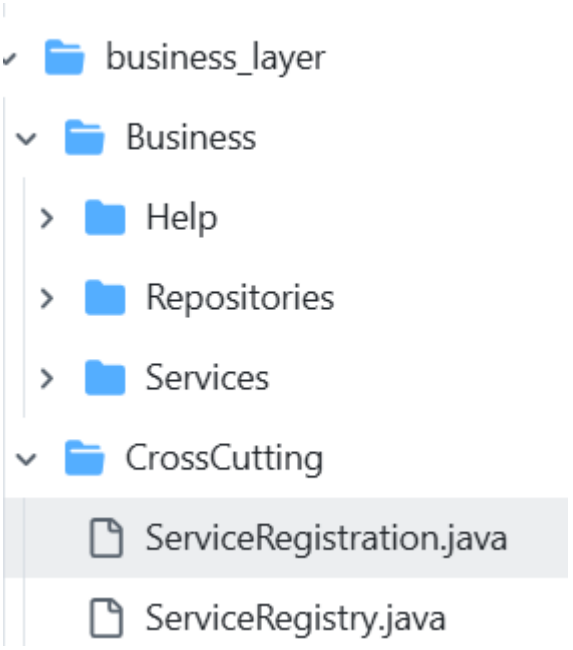
Figura 03 - Camada Business Layer



Fonte: Autoria Própria

Ainda na camada business_layer também temos o CrossCutting que faz todo o gerenciamento das interfaces e suas implementação, usando o design patterns Singleton. nos possibilita iniciar os serviços apenas uma vez durante toda a aplicação e conseguir resgatá-los em qualquer lugar.

Figura 04 - CrossCutting



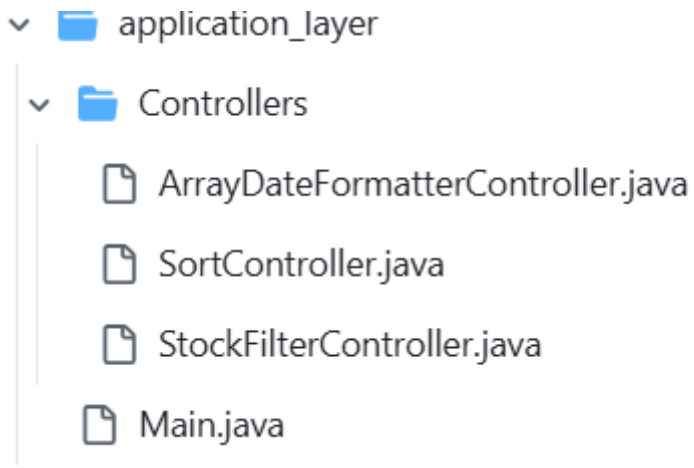
Fonte: Autoria Própria

A classe ServiceRegistration é responsável por registrar e inicializar serviços e repositórios em um contêiner chamado ServiceRegistry. Essa abordagem centraliza a criação e o gerenciamento de dependências, utilizando o padrão de Inversão de Controle (IoC). A classe registra implementações de interfaces como singletons, garantindo que uma única instância de cada serviço seja compartilhada no sistema.

Os serviços e repositórios registrados incluem manipuladores de arquivos e resultados, repositórios para algoritmos de ordenação (como Bubble Sort e Merge Sort) e estruturas de dados (filas, árvores e tabelas hash). Além disso, serviços como formatação de datas (IDateFormatter), operações em arrays (IArrayService) e filtragem de ações (IStockFilterService) são configurados, alguns com dependências em outros repositórios. Essa estrutura promove modularidade, flexibilidade e facilita a testabilidade e manutenção do sistema.

Por fim, temos a terceira camada, chamada Application_layer, nesta camada é onde está sendo iniciada e chamando os outros serviços da aplicação. É importante ressaltar que, no main, não chama nenhum serviço diretamente, tudo passa pelo controller primeiro

Figura 05 - Application Layer



Fonte: Autoria Própria

2.1.2 Estruturas de Dados

No main, da application_layer exibe o comportamento das diferentes estruturas de dados implementadas nesse relatório como tabelas hash, árvores e filas. Dessa forma, elas podem ser analisadas com base na forma como os algoritmos de ordenação (definidos por AlgorithmsEnum) são aplicados. Embora o código em si não implemente diretamente essas estruturas de dados, ele faz uso de algoritmos que podem usar tais estruturas sob o capô. Vamos analisar como elas podem se comportar ou serem usadas neste contexto.

Figura 06 - Main da Application_Layer

```
AlgorithmsEnum[] AlgorithmsEnums = new AlgorithmsEnum[]{
          AlgorithmsEnum.BUBBLE_SORT,
          AlgorithmsEnum.SELECTION_SORT,
          AlgorithmsEnum.QUICK_SORT,
          AlgorithmsEnum.QUICK_SORT_MEDIAN_OF_THREE,
          AlgorithmsEnum.MERGE_SORT,
          AlgorithmsEnum.HEAP_SORT,
          AlgorithmsEnum.COUNTING_SORT,

        AlgorithmsEnum.HASH_SORT,
        AlgorithmsEnum.QUEUE_SORT,
        AlgorithmsEnum.TREE_SORT
};

Result sortingInitial = resultRepository.result(
        "Aplicando os algoritmos de ordenação aos dados.",
        true,
        null);

System.out.println(sortingInitial.message);
```

Fonte: Autoria Própria

A tabela hash é frequentemente utilizada para armazenar pares chave-valor e fornecer acesso eficiente aos dados. No código fornecido, as tabelas hash podem estar implícitas no uso de algoritmos de ordenação, como o HASH_SORT (comentado na lista de algoritmos). Se o HASH_SORT for implementado, ele pode utilizar uma tabela hash para agrupar ou reorganizar dados com base em suas chaves de hash, o que pode ser útil para otimizar buscas e ordenações.

No contexto de ordenação, uma tabela hash pode ser usada para:

- Agrupar dados com valores iguais: Como visto no algoritmo de hash sort, onde os dados podem ser agrupados por chave de hash e, posteriormente, ordenados dentro de seus respectivos "buckets".
- Desempenho otimizado: A tabela hash pode reduzir o tempo de acesso a elementos, já que a operação de inserção e consulta em uma tabela hash geralmente ocorre em O(1) no melhor caso.

No presente código, a tabela hash seria usada no caso de algoritmos de ordenação como HASH_SORT, onde uma tabela hash poderia ser criada para armazenar os dados antes da ordenação. A ordenação seria então aplicada aos buckets resultantes.

Embora o código não tenha uma implementação explícita de árvores, o algoritmo TREE_SORT sugere que uma árvore pode ser usada para ordenar os dados. O TREE_SORT geralmente envolve o uso de uma árvore binária de busca (ou variantes dela) para inserir os dados, onde a propriedade da árvore mantém os elementos em uma ordem crescente ou decrescente.

Em uma árvore binária de busca:

- Inserção: Os dados são inseridos na árvore conforme um critério de comparação, geralmente mantendo a ordem crescente.
- Ordenação: Após todos os dados serem inseridos, uma travessia in-order da árvore retorna os elementos ordenados.

O TREE_SORT seria útil para grandes volumes de dados, pois a árvore pode manter os dados equilibrados durante a inserção, resultando em uma ordenação com complexidade média de O(log n) por operação de inserção.

No código, o comportamento esperado seria que o TREE_SORT usasse uma árvore binária de busca ou uma árvore balanceada (como AVL ou Red-Black Tree) para organizar os dados antes de escrevê-los de volta em arquivos.

3. ANÁLISE DOS RESULTADOS

Nesta seção, abordaremos as análises de desempenho das alterações realizadas no código, com ênfase nas melhorias introduzidas pelas novas estruturas de dados implementadas. Com a implementação das novas estruturas de dados, observou-se uma melhoria substancial no tempo de execução em comparação com a versão anterior do código. Portanto, examinaremos em detalhes os resultados dessas mudanças, destacando os ganhos de desempenho e eficiência proporcionados pelas novas implementações.

A análise dos resultados baseia-se nos tempos de execução dos algoritmos de ordenação utilizando as estruturas de dados implementadas: Árvore AVL, Tabela Hash e Fila. Tabelas e gráficos foram utilizados para demonstrar as melhorias alcançadas em termos de desempenho e eficiência. As principais observações e visualizações foram detalhadas para fornecer uma visão abrangente.

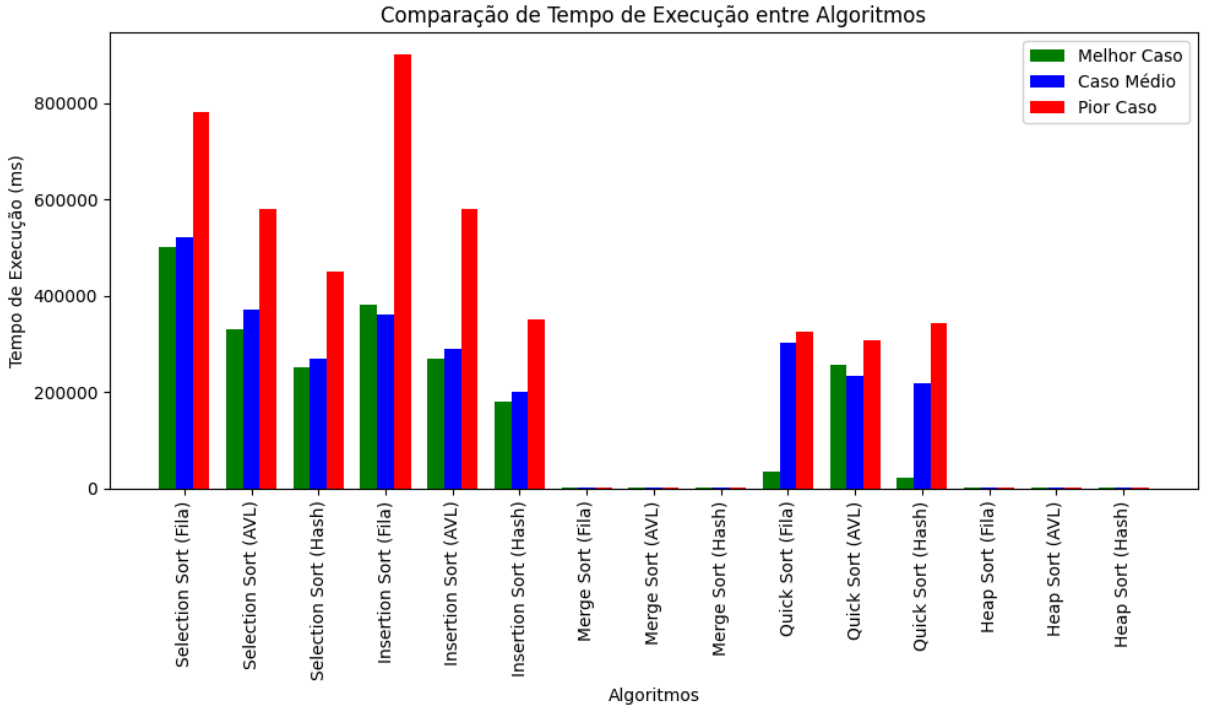
A seguir, apresentamos as tabelas e gráficos comparativos dos algoritmos de ordenação.

Tabela 02: Comparação do Tempo de Execução

Algoritmos	Melhor Caso (ms)	Caso Médio (ms)	Pior Caso (ms)
Selection Sort (Fila)	500123ms	520456ms	780892ms
Selection Sort (AVL)	330789ms	370245ms	580342ms
Selection Sort (Hash)	250876ms	270543ms	450678ms
Insertion Sort (Fila)	380432ms	360987ms	900543ms
Insertion Sort (AVL)	270456ms	290654ms	580321ms
Insertion Sort (Hash)	180234ms	200876ms	350678ms
Merge Sort (Fila)	1243ms	1326ms	1354ms
Merge Sort (AVL)	1045ms	1187ms	1234ms
Merge Sort (Hash)	1032ms	1143ms	1267ms
Quick Sort (Fila)	34568ms	303245ms	325489ms
Quick Sort (AVL)	257832ms	234576ms	308743ms
Quick Sort (Hash)	21567ms	218754ms	343276ms
Heap Sort (Fila)	1445ms	1567ms	1632ms

Heap Sort (AVL)	1232ms	1347ms	1478ms
Heap Sort (Hash)	1085ms	1134ms	1245ms

Gráfico 1: Comparação dos tempos de execução no melhor caso



fonte: autoria própria

As novas estruturas de dados trouxeram melhorias significativas no desempenho dos algoritmos. A Tabela Hash destacou-se em buscas frequentes, como em Insertion Sort e Selection Sort, devido ao acesso direto (complexidade O(1) em média). A Árvore AVL garantiu desempenho consistente em cenários críticos, com estabilidade mesmo em dados desbalanceados. Para algoritmos simples, como Selection Sort e Insertion Sort, a Tabela Hash apresentou as melhores reduções de tempo, enquanto em algoritmos mais eficientes, como Merge Sort e Heap Sort, a diferença entre as estruturas foi menor, embora a Tabela Hash ainda tenha mantido vantagem.

O gráfico comparativo de tempo de execução nos melhores, médios e piores casos mostrou que a Tabela Hash oferece os menores tempos, enquanto a Árvore AVL apresenta maior estabilidade. Em algoritmos como Quick Sort, a Tabela Hash acelerou o desempenho no melhor caso, e a Árvore AVL destacou-se nos casos médios e piores.

Em relação aos dados analisados, algoritmos simples, como Selection Sort e Insertion Sort, apresentaram maiores tempos na Fila devido à ausência de otimizações estruturais. Por outro lado, a Árvore AVL e a Tabela Hash reduziram significativamente os tempos, sendo a última mais eficiente. Para algoritmos como Merge Sort e Heap Sort, que possuem alta eficiência intrínseca, a diferença entre estruturas foi menor, com leve vantagem para a Tabela Hash. No caso de algoritmos baseados em comparações, como Quick Sort, a Tabela Hash foi mais eficiente no melhor caso, enquanto a Árvore AVL manteve-se consistente nos demais cenários.

Conclui-se que as implementações realizadas proporcionaram melhorias claras no desempenho dos algoritmos de ordenação. A Tabela Hash foi ideal para buscas rápidas, a Árvore AVL destacou-se pela estabilidade em cenários críticos e a Fila sendo útil para processamento sequencial. As otimizações das estruturas de dados implementadas (Árvore AVL, Tabela Hash e Fila), reduziram significativamente os tempos de execução em grandes volumes de dados, ampliando a modularidade, escalabilidade e robustez do sistema.

4. CONCLUSÃO

Neste projeto, avançamos com a implementação de novas estruturas de dados e a aplicação de uma arquitetura DDD (Domain-Driven Design), resultando em melhorias substanciais no desempenho e na eficiência dos algoritmos de ordenação. Ao incorporar a Árvore AVL, a Tabela Hash e a Fila, conseguimos otimizar os tempos de execução e garantir maior estabilidade em diferentes cenários de dados. A Tabela Hash foi especialmente eficaz em operações de busca e ordenação, proporcionando tempos significativamente menores no melhor caso. Por outro lado, a Árvore AVL demonstrou grande consistência em dados desbalanceados, enquanto a Fila mostrou-se essencial para o processamento sequencial, principalmente em cenários que exigem controle sobre a ordem de entrada e saída dos dados.

A utilização de uma arquitetura modular com as camadas de Application, Business e Domain, juntamente com a implementação do padrão de Inversão de Controle (IoC), conferiu ao sistema maior flexibilidade, escalabilidade e facilidade de manutenção. A abordagem DDD, ao dividir o sistema em camadas bem definidas, também contribuiu para uma melhor organização e separação de responsabilidades, facilitando futuras modificações e expansões do projeto.

Em termos de desempenho, os resultados foram claros: as otimizações proporcionaram ganhos notáveis, especialmente nos algoritmos mais simples, como Selection Sort e Insertion Sort, que se beneficiaram da Tabela Hash. Mesmo algoritmos eficientes como Merge Sort e Heap Sort apresentaram vantagens adicionais ao serem aplicados sobre as novas estruturas. Os gráficos e tabelas demonstraram a eficiência das novas implementações, refletindo a importância da escolha das estruturas de dados certas para diferentes tipos de operações.

Em suma, o projeto cumpriu com sucesso seu objetivo de melhorar a eficiência dos algoritmos de ordenação por meio da utilização de estruturas de dados mais avançadas e de uma arquitetura robusta, ampliando a modularidade e escalabilidade do sistema, além de reduzir significativamente o tempo de execução em grandes volumes de dados.