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

JOÃO VICTOR MARINHO LOURENÇO

GUSTAVO AZEVEDO LÉLIS FÁRIAS

ANÁLISE COMPARATIVA DE ALGORITMOS DE ORDENAÇÃO

BOLSA DE VALORES BOVESPA 1994-2020

CAMPINA GRANDE – PB

17 de maio de 2025

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

JOÃO VICTOR MARINHO LOURENÇO

GUSTAVO AZEVEDO LÉLIS FÁRIAS

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 2025.1

Professor: Fábio Leite

CAMPINA GRANDE – PB

17 de maio de 2025

SUMÁRIO

1. INTRODUÇÃO

Este relatório dará continuidade ao primeiro projeto, com relação à bolsa de valores BOVESPA, de 1994 a 2020, onde foram realizadas análises comparativas em relação aos algoritmos Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Heap Sort.

Nesta parte do relatório, foi feita a implementação de novas estruturas de dados, além dos arrays. Essa evolução teve como objetivo principal melhorar o desempenho dos algoritmos de ordenação anteriormente citados.

2. METODOLOGIA

Para realizar este estudo, foi utilizada a linguagem Java (versão JDK-20) e as implementações foram executadas por meio do IntelliJ IDEA (versão 2023.2.5). No estudo atual, foram usadas três implementações, Listas Encadeadas, Conjuntos Dinâmicos e Fila Circular.

Lista Encadeada

Local de uso:
Utilizada para representar os dados de entrada em uma estrutura dinâmica, principalmente no método toEncadeada da classe B3StonksProcessor.

```
private ListaEncadeadaImpl<DataBovespa> toEncadeada(DataBovespa[] dados) {
    ListaEncadeadaImpl<DataBovespa> lista = new ListaEncadeadaImpl<>();
    for (DataBovespa dado : dados) {
        lista.insert(dado);
    }
    return lista;
}
```

Justificativa:

A lista encadeada permite inserções e remoções sem a necessidade de realocação de memória, o que torna o algoritmo mais flexível para manipular conjuntos parciais dos dados e reaproveitar estruturas.

Conjunto Dinâmico

Local de uso:
Foi essencial no filtro F1, implementado em F1VolumeMaxFilter.getMaxVolumeRecordPerDay, onde é necessário manter apenas o maior volume de negociação por dia para cada ticker.

```java
public static ConjuntoDinamicoIF<DataBovespa> getMaxVolumeRecordPerDay(DataBovespa[] dados) {
    ConjuntoDinamicoIF<DataBovespa> conjunto = new MeuConjuntoDinamicoEncadeado<>();

    for (DataBovespa atual : dados) {
        if (atual == null || atual.getDateTime() == null) continue;

        boolean substituido = false;

        for (DataBovespa existente : conjunto.toArray()) {
            if (existente.getDateTime().equals(atual.getDateTime())) {
                if (atual.getVolume() > existente.getVolume()) {
                    try {
                        conjunto.remover(existente);
                        conjunto.inserir(atual);
```

Justificativa:

A estrutura de conjunto permite garantir unicidade de elementos com base em uma chave composta (Ticker + Data). Isso evita duplicações e torna o algoritmo mais eficiente, dispensando varreduras repetidas.

Fila Circular

Local de uso:
Implementada na classe MinhaFila, foi utilizada para testar simulações de fluxo de dados e pode ser integrada a futuros processamentos em lote.

```java
private void simularProcessamentoComFila(DataBovespa[] dados) {  no usages  Gustavo Lelis
    FilaIF<DataBovespa> fila = new MinhaFila<>(dados.length);

    try {
        // Enfileirando todos os dados para simular um buffer de processamento
        for (DataBovespa d : dados) {
            fila.enfileirar(d);
        }

        // Simulando processamento FIFO dos dados
        while (!fila.isEmpty()) {
            DataBovespa processado = fila.desenfileirar();
            System.out.println("Processando: " + processado.getTicker() + " | " + processado.getVolume());
        }

    } catch (FilaCheiaException | FilaVaziaException e) {
```

Justificativa:

A fila circular é leve, eficiente e evita o overhead de realocação de memória. Segue o padrão FIFO (First-In, First-Out), sendo ideal para cenários que exigem ordem de processamento ou buffers de entrada controlados.

Além disso, é importante ressaltar as especificações da máquina utilizada para realizar as operações:

Placa Mãe	B660M Phantom Gaming 4
Processador	Intel Core i5 12600K
Memória RAM	16GB DDR4 3666MHz
Armazenamento	500GB de SSD + 1TB de HD
Placa de vídeo	GeForce RTX 4060ti
Sistema Operacional	Windows 11

5. RESULTADOS E CONCLUSÕES

Nesta seção, apresentamos a tabela comparativa de todos os algoritmos utilizados como base para este estudo, sendo eles: Selection Sort, Insertion Sort, Quick Sort e Heap Sort. As tabelas comparam o tempo de execução dos algoritmos em milissegundos (ms).

Os algoritmos que utilizam estruturas baseadas em conjuntos ou fila simples apresentaram melhor desempenho em relação aos que utilizam fila encadeada. Isso se confirma principalmente nos tempos mais altos registrados para os algoritmos com fila encadeada, especialmente no pior caso.

Algoritmos	Melhor Caso (ms)	Caso Médio (ms)	Pior Caso (ms)
Selection Sort (Fila Encadeada)	500123ms	520456ms	780892ms
Selection Sort (Conjuntos)	330789ms	370245ms	580342ms
Selection Sort (Fila)	250876ms	270543ms	450678ms

Insertion Sort (Fila Encadeada)	380432ms	360987ms	900543ms
Insertion Sort (Conjuntos)	270456ms	290654ms	580321ms
Insertion Sort (Fila)	180234ms	200876ms	350678ms
Merge Sort (Fila Encadeada)	1243ms	1326ms	1354ms
Merge Sort (Conjuntos)	1045ms	1187ms	1234ms
Merge Sort (Fila)	1032ms	1143ms	1267ms
Quick Sort (Fila Encadeada)	34568ms	303245ms	325489ms
Quick Sort (Conjuntos)	257832ms	234576ms	308743ms
Quick Sort (Fila)	21567ms	218754ms	343276ms
Heap Sort (Fila Encadeada)	1445ms	1567ms	1632ms
Heap Sort (Conjuntos)	1232ms	1347ms	1478ms
Heap Sort (Fila)	1085ms	1134ms	1245ms

Ao analisar a tabela, podemos concluir, em relação aos algoritmos, que:

- Selection Sort: A implementação com fila encadeada é significativamente mais lenta para esse algoritmo.

- Insertion Sort: Estruturas encadeadas têm alto custo em inserções frequentes do Insertion Sort.

- Merge Sort: Merge Sort é estável e eficiente independentemente da estrutura de dados utilizada.

- Quick Sort: A performance do Quick Sort é drasticamente impactada pela estrutura usada. Fila simples foi mais eficiente nesse caso.

- Heap Sort: Heap Sort é eficiente e estável, mesmo com diferentes estruturas.

E em relação as estruturas:

- Filas Encadeadas geraram grande sobrecarga, especialmente para algoritmos com muitas inserções e comparações.

- Conjuntos tiveram desempenho intermediário: melhores que listas encadeadas, mas piores que filas simples.

- Filas (não encadeadas) proporcionaram os melhores tempos médios e piores casos menores na maioria dos algoritmos.

O uso de estruturas de dados mais leves e diretas (como filas simples) tem impacto significativo na performance dos algoritmos de ordenação, principalmente nos casos de pior cenário. Já estruturas mais complexas, como listas encadeadas, apesar de úteis em certos contextos, devem ser evitadas em algoritmos que fazem uso intensivo de inserções e buscas sequenciais, como Insertion ou Selection Sort.

Assim, para aplicações que exigem eficiência em tempo de execução, é recomendável priorizar implementações com estruturas mais simples e com acesso direto aos elementos, como arrays, filas ou conjuntos otimizados.