Centro Universitário FAMINAS
CAMPUS MURIAÉ

Gabriel da Costa Rodrigues

RELATÓRIO DE ANÁLISE COMPARATIVA DE DESEMPENHO: Vetores, Árvores Binárias de Busca e Árvores AVL

MURIAÉ – MG
2025

IDENTIFICAÇÃO DA EQUIPE

Nome: Gabriel da Costa Rodrigues

Titulação: Técnico em Agroecologia; Técnico em Edificações; Graduando em Análise e Desenvolvimento de Sistemas

Campus: Muriaé

E-mail: ~~xxxxx@xxx.com~~

Telefone: ~~(xx) xxxxx-xxxx~~

MURIAÉ – MG
2025

RESUMO

A eficiência das aplicações de software depende intrinsecamente da escolha adequada das estruturas de dados subjacentes. Este trabalho apresenta uma análise comparativa de desempenho entre três estruturas fundamentais: Vetores, Árvores Binárias de Busca (ABB) e Árvores AVL. O objetivo principal foi avaliar o tempo de execução das operações de inserção e busca em diferentes cenários, variando o volume de dados (100, 1.000 e 10.000 elementos) e a ordenação inicial (crescente, decrescente e aleatória). A metodologia consistiu na implementação dos algoritmos em linguagem Java, sem o uso de bibliotecas de coleções nativas, garantindo o controle sobre a alocação e manipulação de memória. Adicionalmente, foram comparados dois algoritmos de ordenação para vetores: *Bubble Sort* e *Quick Sort*. Os resultados experimentais demonstraram que, embora a Árvore Binária de Busca ofereça eficiência em dados aleatórios, seu desempenho degrada significativamente para O(n) quando submetida a entradas ordenadas, comportando-se como uma lista encadeada. Em contrapartida, o uso de Vetores combinados com o algoritmo *Quick Sort* e Busca Binária mostrou-se altamente eficaz para dados estáticos. O estudo conclui reforçando a importância do balanceamento em árvores (AVL) e a inviabilidade de algoritmos de ordenação quadráticos (*Bubble Sort*) para grandes volumes de dados.

Palavras-chave: Estruturas de Dados, Análise de Desempenho, Java, Árvores Binárias, Ordenação.

SUMÁRIO

1. INTRODUÇÃO

A eficiência de sistemas computacionais modernos não depende exclusivamente da capacidade de processamento do hardware, mas fundamentalmente da forma como os dados são organizados, armazenados e manipulados pelo software. No contexto da Ciência da Computação, a escolha adequada da estrutura de dados é uma decisão crítica que impacta diretamente a performance de uma aplicação, determinando se ela será escalável ou se tornará um gargalo à medida que o volume de informações cresce.

Embora a teoria da complexidade computacional (Notação Big O) forneça previsões sobre o comportamento dos algoritmos, a prática muitas vezes revela nuances que apenas a experimentação pode demonstrar. Fenômenos como a degeneração de árvores binárias ou a disparidade de tempo entre métodos de ordenação quadráticos e logarítmicos tornam-se evidentes quando submetidos a testes de estresse com grandes volumes de dados. Além disso, o uso cotidiano de bibliotecas de alto nível (como o java.util) muitas vezes abstrai o funcionamento interno dessas estruturas, criando uma lacuna no entendimento técnico do estudante.

Neste sentido, a implementação manual ("do zero") de estruturas e algoritmos clássicos é uma etapa pedagógica essencial. Ela permite não apenas a compreensão da lógica de ponteiros e alocação de memória, mas também a visualização prática das vantagens e desvantagens de cada modelo em cenários controlados.

Diante disso, este relatório apresenta a análise de desempenho computacional de três estruturas de dados fundamentais: Vetores (Arrays), Árvores Binárias de Busca (ABB) e Árvores AVL. O objetivo central é comparar a eficiência dessas estruturas em operações críticas de inserção, busca e ordenação. Para garantir uma análise abrangente, foram utilizados conjuntos de dados de diferentes magnitudes (100, 1.000 e 10.000 elementos) e disposições (ordenado, inversamente ordenado e aleatório), permitindo identificar o comportamento de cada estrutura tanto no caso médio quanto no pior caso.

2. OBJETIVOS:

Analisar e comparar o desempenho computacional de três estruturas de dados fundamentais (Vetores, Árvores Binárias de Busca e Árvores AVL) quando submetidas a operações de inserção e busca, visando compreender suas vantagens e limitações práticas em diferentes cenários.

2.1 Objetivos Específicos

- Implementar manualmente, em linguagem Java, as estruturas de dados Vetor, Árvore Binária de Busca (ABB) e Árvore AVL, sem o uso de bibliotecas de coleções nativas, para garantir o controle sobre a manipulação de memória.

- Desenvolver e integrar algoritmos de ordenação (*Bubble Sort* e *Quick Sort*) e métodos de busca (Sequencial e Binária) para a estrutura de vetores .

- Medir o tempo de execução das operações de inserção, busca e ordenação utilizando a classe System.nanoTime(), considerando conjuntos de dados de diferentes magnitudes (100, 1.000 e 10.000 elementos) .

- Avaliar o comportamento das estruturas diante de diferentes disposições de dados de entrada: ordenada, inversamente ordenada e aleatória .

- Confrontar os resultados práticos obtidos nos testes de estresse com a complexidade teórica (Notação Big O) esperada para cada algoritmo .

- Demonstrar empiricamente os efeitos da degeneração em Árvores Binárias de Busca não balanceadas e a eficácia dos algoritmos de ordenação logarítmicos (*Quick Sort*) frente aos quadráticos (*Bubble Sort*).

3. METODOLOGIA:

Os algoritmos foram implementados na linguagem Java, sem o uso de bibliotecas de coleções nativas, garantindo a análise pura da lógica implementada.

3.1 Ambiente de Teste

- **Linguagem:** Java
- **IDLE**: VSCode + JGRASP(debug árvores)
- **Hardware:** RYZEN 5 4500, 24GB DDR4 3200 MHz(1X8 XPG + 1X16 GEIL), SSD WDGREEN 480, RTX 2080 8GB
- **Sistema Operacional:** Windows 11 PRO

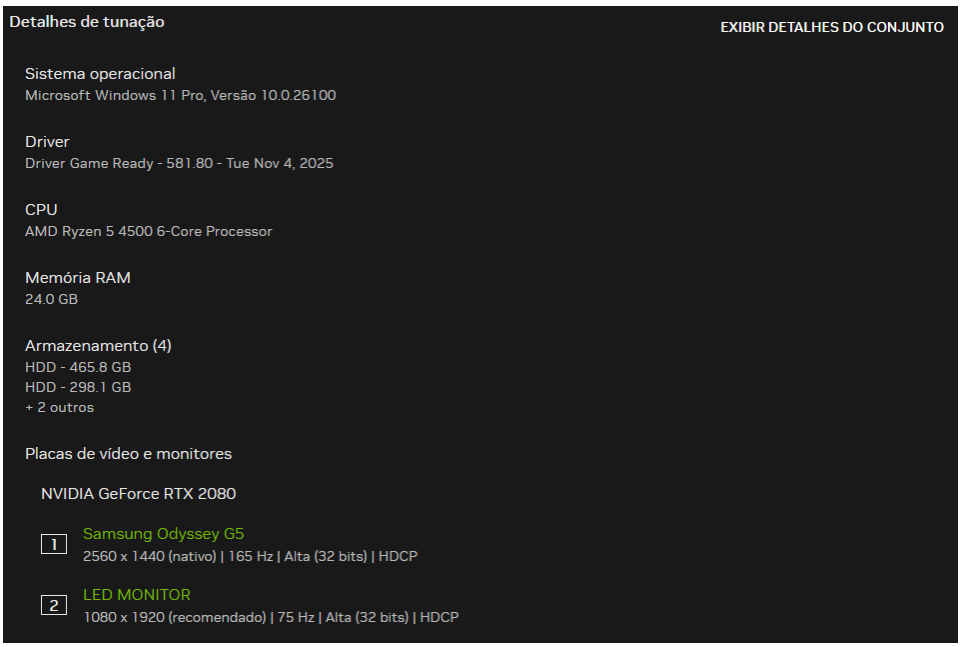


Figura 1: Captura de Tela do NVIDIA APP

3.2 Procedimento de Coleta

Para cada cenário de teste (combinação de estrutura, tamanho e ordenação), foram realizadas **5 execuções**. O tempo registrado nas tabelas e gráficos representa a **média aritmética** dessas execuções, calculada em nanossegundos e convertida para milissegundos para facilitar a leitura.

3.3 Link do repositório GitHub:

https://github.com/gabrielxniper/java-data-structures-performance-analysis

4. RESULTADOS E DISCUSSÃO:

A análise dos dados coletados permite correlacionar o comportamento prático dos algoritmos com a teoria da complexidade computacional.

TABELA INSERÇÃO			
Tamanho / Ordem	Vetor(ms)	Árvore Binária(ms)	Árvore AVL(ms)
100 / Ordenada	0,0047	0,0324	0,1645
100 / Inversa	0,0072	0,1486	0,0680
100 / Aleatória	0,0030	0,0319	0,0372
1000 / Ordenada	0,0315	1,5062	0,7328
1000 / Inversa	0,0438	2,5535	0,6531
1000 / Aleatória	0,0321	0,1120	0,1394
10000 / Ordenada	0,2837	140,9703	1,5621
10000 / Inversa	0,3199	194,4842	1,9520
10000 / Aleatória	0,3846	1,1506	4,0140
MÉDIA	0,1234	37,8877	1,0359

(FIGURA 2: TABELA COM TEM,PO DAS INSERÇÕES)

TABELA BUSCA BINÁRIA				
Tamanho / Ordem	Vetor(ms)	Árvore Binária cresc (ms)	Árvore Binária aleat(ms)	Árvore AVL(ms)
100 - valor inicial	0,0006	0,0002	0,0003	0,0004
100 - valor meio	0,0017	0,0017	0,0002	0,0015
100 - valor final	0,0019	0,0060	0,0055	0,0004
100 - valor aleat 1	0,0016	0,0011	0,0005	0,0015
100 - valor aleat 2	0,0016	0,0019	0,0003	0,0004
100 - valor aleat 3	0,0017	0,0027	0,0005	0,0012
100 - valor inex	0,0008	0,0036	0,0002	0,0005
1000 - valor inicial	0,0006	0,0009	0,0002	0,0005
1000 - valor meio	0,0011	0,0309	0,0003	0,0014
1000 - valor final	0,0017	0,0484	0,0103	0,0007
1000 - valor aleat 1	0,0012	0,0050	0,0002	0,0004
1000 - valor aleat 2	0,0013	0,0097	0,0003	0,0013
1000 - valor aleat 3	0,0015	0,0326	0,0007	0,0036
1000 - valor inex	0,0015	0,0387	0,0002	0,0004
10000 - valor inicial	0,0009	0,0003	0,0002	0,0021
10000 - valor meio	0,0017	0,0235	0,0033	0,0007
10000 - valor final	0,0010	0,0196	0,0014	0,0016
10000 - valor aleat 1	0,0066	0,0015	0,0004	0,0015
10000 - valor aleat 2	0,0005	0,0054	0,0004	0,0007
10000 - valor aleat 3	0,0010	0,0170	0,0009	0,0010
10000 - valor inex	0,0006	0,0957	0,0002	0,1019
MEDIA	0,0015	0,0165	0,0013	0,0059

(FIGURA 3: TABELA COM TEM,PO DE BUSCAS BINÁRIAS)

TABELA BUSCA SEQUENCIAL	
Tamanho / Ordem	Vetor(ms)
100 - valor inicial	0,0004
100 - valor meio	0,0014
100 - valor final	0,0023
100 - valor aleat 1	0,0008
100 - valor aleat 2	0,0009
100 - valor aleat 3	0,0034
100 - valor inex	0,0023
1000 - valor inicial	0,0003
1000 - valor meio	0,0066
1000 - valor final	0,0124
1000 - valor aleat 1	0,0044
1000 - valor aleat 2	0,0015
1000 - valor aleat 3	0,0142
1000 - valor inex	0,0146
10000 - valor inicial	0,0004
10000 - valor meio	0,0529
10000 - valor final	0,1228
10000 - valor aleat 1	0,0251
10000 - valor aleat 2	0,0086
10000 - valor aleat 3	0,0949
10000 - valor inex	0,1016
MÉDIA	**0,0225**

(FIGURA 4: TABELA COM TEMPO DAS BUSCAS SEQUENCIAS)

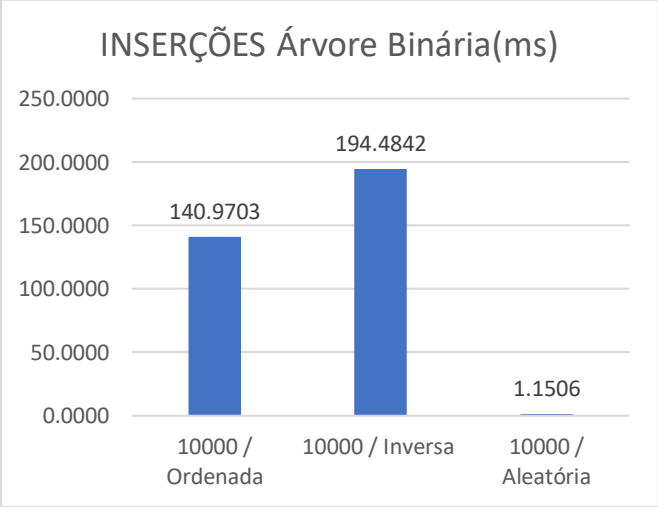
	TABELA ORDENAÇÃO BUBBLE SORT	TABELA ORDENAÇÃO QUICK SORT
Tamanho / Ordem	Vetor(ms)	Vetor(ms)
100 / Ordenada	0,0920	0,0234
100 / Inversa	0,1610	0,0283
100 / Aleatória	0,1425	0,0241
1000 / Ordenada	0,1736	0,0500
1000 / Inversa	1,7645	0,0622
1000 / Aleatória	1,3259	0,1287
10000 / Ordenada	17,2345	1,6426
10000 / Inversa	31,3284	0,4643
10000 / Aleatória	76,0239	2,9022
MÉDIA	14,2496	0,5918

(FIGURA 5: TABELA COM ORDENAÇÕES BUBBLE SORT E QUICK SORT PARA VETORES)

4.1 Inserção: O Impacto do Balanceamento

O experimento de inserção evidenciou a principal fragilidade da Árvore Binária de Busca (ABB) simples.

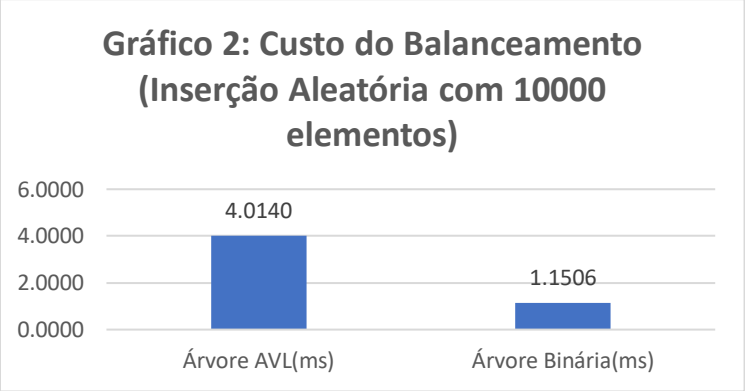
- **Cenário Crítico:** Ao inserir 10.000 elementos ordenados, a ABB comportou-se como uma lista encadeada ($O(n^2)$), registrando um tempo médio de **140,97 ms** (**194,48 ms** com valores decrescentes).



(Gráfico 1: A "Explosão" da ABB (Inserção))

- **Solução AVL:** A Árvore AVL, no mesmo cenário, manteve o desempenho estável em **1,56 ms**. Isso comprova a eficácia das rotações automáticas para manter a altura da árvore logarítmica (O(log n)).

- **O Custo do Equilíbrio (Overhead):** Observou-se, contudo, um fenômeno interessante no cenário aleatório. A inserção na ABB (**1,15 ms**) foi mais rápida que na AVL (**4,01 ms**). Isso ocorre porque, com dados aleatórios, a ABB tende a ficar naturalmente equilibrada sem esforço, enquanto a AVL "perde tempo" verificando fatores de balanceamento e realizando rotações a cada inserção.



(Gráfico 2: Custo do Balanceamento (Inserção Aleatória)

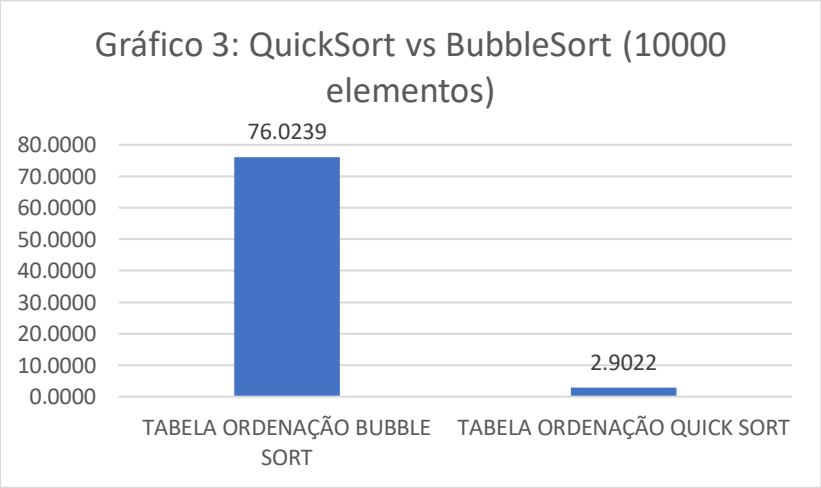
4.2 Desempenho de Busca

A busca binária em vetores ordenados demonstrou ser a técnica mais eficiente, alcançando 0,0006 ms para 10.000 itens, beneficiando-se da indexação direta e cache de memória.

Já nas árvores, a degeneração da ABB refletiu-se diretamente na busca: encontrar um valor inexistente na ABB ordenada levou 0,0957 ms (percorrendo todos os nós), enquanto a AVL manteve tempos baixos na maioria dos casos, embora tenha apresentado leve oscilação no maior conjunto de dados.

4.3 Ordenação

A disparidade entre algoritmos de complexidade quadrática e logarítmica ficou evidente. O *Bubble Sort* levou **76,02 ms** para ordenar 10.000 itens aleatórios, tornando-se inviável para grandes volumes. Em contrapartida, o *Quick Sort* realizou a mesma tarefa em **2,90 ms**, sendo aproximadamente 26 vezes mais rápido.



(Gráfico 3: QuickSort vs BubbleSort)

5. CONSIDERAÇÕES FINAIS:

O estudo conclui que não existe uma estrutura de dados universalmente superior; a escolha depende estritamente da natureza dos dados e das operações prioritárias.

- Viabilidade da ABB: A Árvore Binária de Busca Simples mostrou-se uma estrutura arriscada para dados do mundo real, onde a pré-ordenação é comum. Sua utilização só é justificável se houver garantia estatística de aleatoriedade na entrada, caso em que supera a AVL por não possuir overhead.
- Necessidade da AVL: Para sistemas genéricos e robustos, a Árvore AVL é imperativa. O custo extra de processamento na inserção (cerca de 3 a 4 vezes mais lento que a ABB no caso médio) é um "seguro" necessário para evitar a falha catastrófica de desempenho (100 vezes mais lento) observada na ABB degenerada.
- Vetores e Ordenação: Para coleções estáticas ou com poucas inserções, o uso de Vetores ordenados com Quick Sort e Busca Binária continua sendo a abordagem mais eficiente em termos de tempo de CPU e uso de memória, superando as estruturas baseadas em ponteiros devido à otimização de hardware.