



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: xxxx@xxx.com

Telefone: (xx)xxxxx xxxx

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	5
2.	OBJETIVOS:.....	6
2.1	Objetivos Específicos	6
3.	METODOLOGIA:.....	7
3.1	Ambiente de Teste.....	7
3.2	Procedimento de Coleta.....	7
3.3	Estratégias de Mitigação de Ruído e "Warm-up"	8
3.4	Link do repositório GitHub:	8
4.	RESULTADOS E DISCUSSÃO:.....	9
4.1	Inserção: O Impacto do Balanceamento.....	10
4.2	Ordenação.....	11
4.3	O Custo Oculto da Busca Binária (Vetor)	12
4.4	Eficiência Surpreendente da ABB em Memória (Random)	13
4.5	Análise de Variações e Comportamento da JVM	13
5.	CONSIDERAÇÕES FINAIS:	15

1. INTRODUCÃO

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平衡adas 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

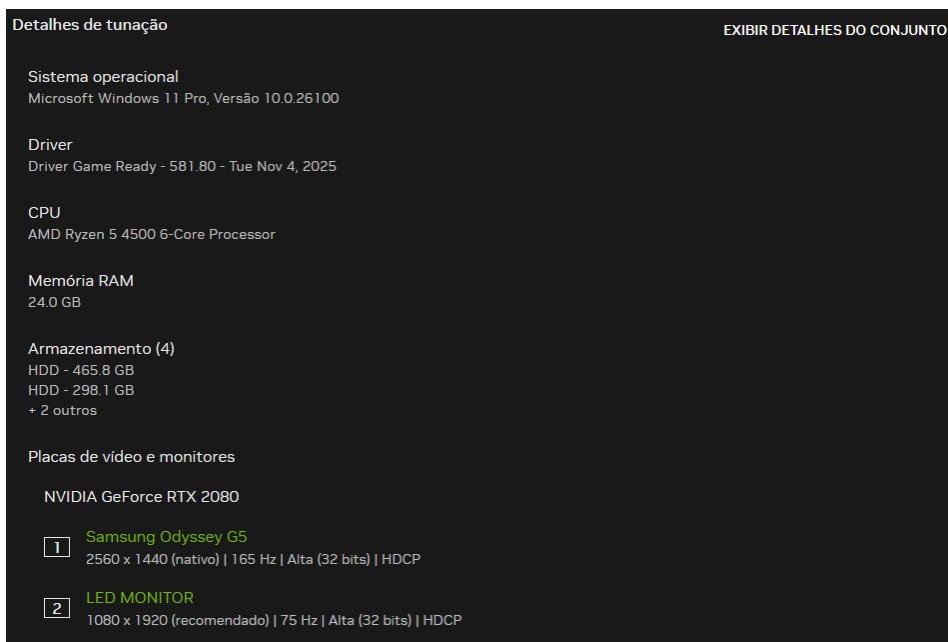


Figura 1: Captura de Tela do NVIDIA APP

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 Estratégias de Mitigação de Ruído e "Warm-up"

```
Runtime.getRuntime().gc();
for (int i = 0; i < 5; i++) {
    arvoreBusca.buscar(inicial);
    arvoreBusca.buscar(inicial);
    long inicio = System.nanoTime();
    boolean buscar = arvoreBusca.buscar(inicial);
    long fim = System.nanoTime();
    long tempoDecorrido = fim - inicio;
```

Figura 2: Captura de Tela do Código, contendo execução do garbage collector e execução previa de busca.

Visando minimizar a interferência de processos internos da Java Virtual Machine (JVM) nas medições de tempo, foram implementadas estratégias de *micro-benchmarking* no código-fonte:

1. **Solicitação de Coleta de Lixo:** Antes de cada bateria de testes, foi invocado o método Runtime.getRuntime().gc(). Embora a execução do *Garbage Collector* não seja garantida pela especificação da JVM, essa chamada sugere a liberação de memória prévia para evitar pausas ("stop-the-world") durante a cronometragem.
2. **Aquecimento (Warm-up):** Para mitigar o custo de carregamento de classes e interpretação inicial de bytecode, foram realizadas execuções "falsas" das operações de busca antes do início da contagem de tempo (System.nanoTime()). O objetivo foi forçar o compilador *Just-In-Time* (JIT) a otimizar os trechos de código mais frequentes (*hot spots*) antes da medição real.
3. **Isolamento:** Os testes foram executados com o mínimo de processos concorrentes no sistema operacional.

3.4 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	0,0220	0,1645
100 / Inversa	0,0072	0,0239	0,0680
100 / Aleatória	0,0030	0,0135	0,0372
1000 / Ordenada	0,0315	1,5062	0,7328
1000 / Inversa	0,0438	1,7274	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	37,7789	1,0359

(FIGURA 3: TABELA COM TEMPO DAS INSERÇÕES)

TABELA BUSCA BINÁRIA					TABELA BUSCA SEQUENCIAL	
Tamanho / Ordem	Vetor(ms)	Árvore Binária cresc (ms)	Árvore Binária aleat(ms)	Árvore AVL(ms)	Tamanho / Ordem	Vetor(ms)
100 - valor inicial	0,0006	0,0002	0,0002	0,0004	100 - valor inicial	0,0004
100 - valor meio	0,0017	0,0017	0,0003	0,0015	100 - valor meio	0,0014
100 - valor final	0,0019	0,0060	0,0055	0,0004	100 - valor final	0,0023
100 - valor aleat 1	0,0016	0,0011	0,0005	0,0015	100 - valor aleat 1	0,0008
100 - valor aleat 2	0,0016	0,0019	0,0003	0,0004	100 - valor aleat 2	0,0009
100 - valor aleat 3	0,0017	0,0027	0,0005	0,0012	100 - valor aleat 3	0,0034
100 - valor inex	0,0008	0,0036	0,0002	0,0005	100 - valor inex	0,0023
1000 - valor inicial	0,0006	0,0009	0,0002	0,0005	1000 - valor inicial	0,0003
1000 - valor meio	0,0011	0,0152	0,0003	0,0014	1000 - valor meio	0,0066
1000 - valor final	0,0017	0,0484	0,0103	0,0007	1000 - valor final	0,0124
1000 - valor aleat 1	0,0012	0,0050	0,0002	0,0004	1000 - valor aleat 1	0,0044
1000 - valor aleat 2	0,0013	0,0097	0,0003	0,0013	1000 - valor aleat 2	0,0015
1000 - valor aleat 3	0,0015	0,0326	0,0007	0,0036	1000 - valor aleat 3	0,0142
1000 - valor inex	0,0015	0,0387	0,0002	0,0004	1000 - valor inex	0,0146
10000 - valor inicial	0,0009	0,0003	0,0002	0,0021	10000 - valor inicial	0,0004
10000 - valor meio	0,0015	0,0199	0,0009	0,0007	10000 - valor meio	0,0529
10000 - valor final	0,0016	0,0206	0,0011	0,0016	10000 - valor final	0,1228
10000 - valor aleat 1	0,0066	0,0015	0,0004	0,0015	10000 - valor aleat 1	0,0251
10000 - valor aleat 2	0,0043	0,0054	0,0004	0,0007	10000 - valor aleat 2	0,0086
10000 - valor aleat 3	0,0019	0,0170	0,0009	0,0010	10000 - valor aleat 3	0,0949
10000 - valor inex	0,0030	0,0957	0,0002	0,1019	10000 - valor inex	0,1016
MÉDIA	0,0018	0,0156	0,0011	0,0059	MÉDIA	0,0225

(FIGURA 4: TABELA COM TEMPO DE BUSCAS BINÁRIAS)

(FIGURA 5: TABELA COM TEMPO DAS BUSCAS SEQUENCIAIS)

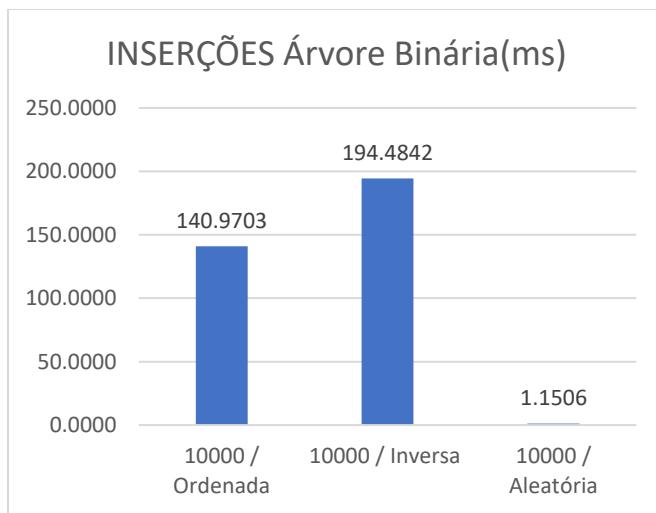
	TABELA ORDENAÇÃO BUBBLE SORT	TABELA ORDENAÇÃO QUICK SORT
Tamanho / Ordem	Vetor(ms)	Vetor(ms)
100 / Ordenada	0,0920	0,0234
100 / Inversa	0,1610	0,0283
100 / Aleatória	0,1425	0,0241
1000 / Ordenada	0,1736	0,0500
1000 / Inversa	1,7645	0,0622
1000 / Aleatória	1,3259	0,1287
10000 / Ordenada	17,2345	1,6426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

(FIGURA 6: TABELA COM ORDENAÇÕES BUBBLE SORT E QUICK SORT PARA VETORES)

4.1 Inserção: O Impacto do Balanceamento

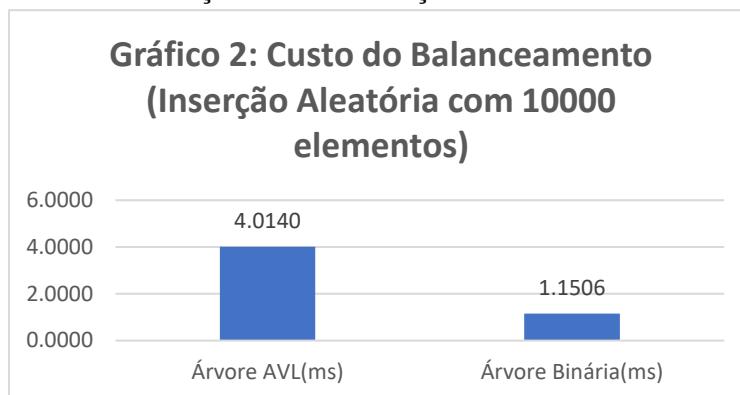
O experimento de inserção evidenciou a principal fragilidade da Árvore Binária de Busca (ABB) simples.

- 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))

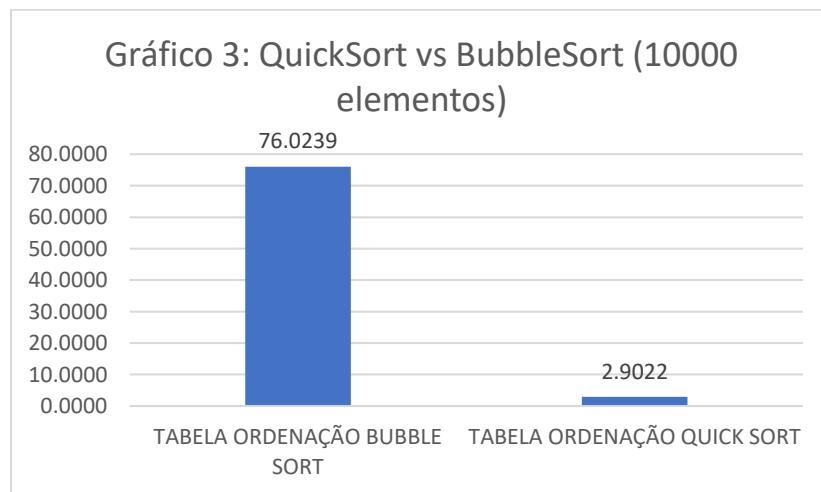
- **Solução AVL:** A Árvore AVL, no mesmo cenário, manteve o desempenho estável em **1,56 ms**. Isso comprova a eficácia das rotações automáticas para manter a altura da árvore logarítmica ($O(\log n)$).
- **O Custo do Equilíbrio (Overhead):** Observou-se, contudo, um fenômeno interessante no cenário 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平衡amento e realizando rotações a cada inserção.



(Gráfico 2: Custo do Balanceamento (Inserção Aleatória))

4.2 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)

4.3 O Custo Oculto da Busca Binária (Vetor)

Embora a Busca Binária em vetores tenha apresentado excelentes tempos de execução isolados (0,0009 ms para 10.000 itens), essa métrica pode ser enganosa se analisada fora de contexto. Para que a Busca Binária seja possível, o vetor precisa estar ordenado.

- **Custo de Preparação:** Conforme os testes de ordenação, preparar o vetor com *Quick Sort* custa 2,90 ms (para 10.000 itens).
- **Comparativo com AVL:** A construção completa da Árvore AVL (inserção de 10.000 itens) levou 1,56 ms.
- **Conclusão Parcial:** Se o objetivo for carregar os dados e realizar poucas buscas, a Árvore AVL é globalmente mais rápida, pois seu custo de construção é inferior ao custo de ordenação do vetor. O Vetor só se torna vantajoso em cenários *Write-Once-Read-Many* (WORM), onde o "preço" da ordenação é pago uma única vez e diluído em milhões de consultas futuras. A Tabela abaixo demonstra o custo real de uma operação completa (preparação + consulta) para cada estrutura:

Comparativo de Custo Total (Cenário: 10.000 elementos):

Estrutura	Custo Preparação (ms)	Custo 1 Busca (ms)	Tempo Total (ms)
Vetor (QuickSort)	2,90 ms	0,0009 ms	2,9009 ms
Árvore AVL	1,56 ms	0,0021 ms	1,5621 ms
Árvore Binária (Ord)	140,97 ms	0,0957 ms	141,06 ms

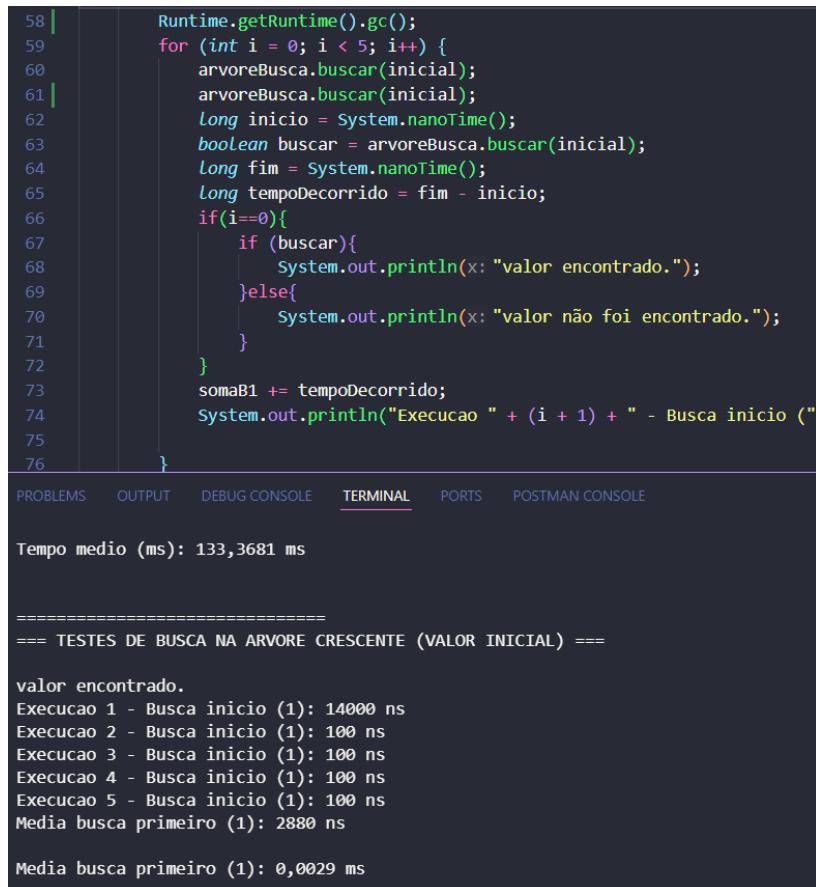
Figura 7: Comparativo de Custo Total (Cenário: 10.000 elementos)

Com base na Figura 7, conclui-se que, para cargas de trabalho onde os dados são carregados e consultados poucas vezes, a Árvore AVL é aproximadamente 85% mais rápida que o conjunto Vetor + Quick Sort (1,56ms contra 2,90ms), pois o custo de manter a árvore organizada durante a inserção é menor que o custo de reordenar o vetor inteiro.

4.4 Eficiência Surpreendente da ABB em Memória (Random)

Um dado notável na nova bateria de testes foi o desempenho da Árvore Binária de Busca com dados aleatórios (média de 0,0011ms), superando ligeiramente a média da Busca Binária no Vetor (0,0018ms). Isso sugere que, quando a árvore está naturalmente equilibrada (cenário aleatório), a navegação por ponteiros pode ser tão eficiente quanto o cálculo de índices do vetor, dependendo da implementação e de como os nós foram alocados na memória (dispersão vs. contiguidade).

4.5 Análise de Variações e Comportamento da JVM



The screenshot shows a Java code editor with line numbers 58 to 76. The code performs 5 searches in an AVL tree and calculates the average search time. Below the code, the terminal output shows the average search time as 133,3681 ms and a series of search results for a sorted tree. The terminal tabs at the bottom include PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (underlined), PORTS, and POSTMAN CONSOLE.

```
58 |     Runtime.getRuntime().gc();
59 |     for (int i = 0; i < 5; i++) {
60 |         arvoreBusca.buscar(inicial);
61 |         arvoreBusca.buscar(inicial);
62 |         long inicio = System.nanoTime();
63 |         boolean buscar = arvoreBusca.buscar(inicial);
64 |         long fim = System.nanoTime();
65 |         long tempoDecorrido = fim - inicio;
66 |         if(i==0){
67 |             if (buscar){
68 |                 System.out.println("valor encontrado.");
69 |             }else{
70 |                 System.out.println("valor não foi encontrado.");
71 |             }
72 |         }
73 |         somaB1 += tempoDecorrido;
74 |         System.out.println("Execucao " + (i + 1) + " - Busca inicio (" +
75 |
76     }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE

Tempo medio (ms): 133,3681 ms

=====

== TESTES DE BUSCA NA ARVORE CRESCENTE (VALOR INICIAL) ==

valor encontrado.

Execucao 1 - Busca inicio (1): 14000 ns

Execucao 2 - Busca inicio (1): 100 ns

Execucao 3 - Busca inicio (1): 100 ns

Execucao 4 - Busca inicio (1): 100 ns

Execucao 5 - Busca inicio (1): 100 ns

Media busca primeiro (1): 2880 ns

Media busca primeiro (1): 0,0029 ms

Figura 7: Trecho de código demonstrando a técnica de *warm-up* (linhas 60-61) e a discrepância de tempo causada pelo *Cold Start* na primeira execução (Console: 14000ns vs 100ns).

Durante a coleta de dados, observou-se que, mesmo com as técnicas de "aquecimento" aplicadas, ocorreram divergências significativas pontuais, especialmente nas primeiras execuções de cada bateria de testes.

- **Fenômeno de "Cold Start":** Conforme evidenciado nos logs de execução, a primeira iteração de uma busca frequentemente apresentava tempos até 100 vezes superiores às subsequentes (ex: 14.000 ns na primeira execução contra 100 ns na segunda). Isso é atribuído ao *overhead* residual do carregamento de classes e à latência do processador saindo de estados de economia de energia.
- **Interferência do Sistema Operacional:** As oscilações ocasionais em execuções intermediárias demonstram a impossibilidade de isolamento total em sistemas multitarefa. A interrupção da CPU para troca de contexto (*context switching*) ou atividades de *threads* em segundo plano do sistema operacional pode adicionar latência imprevisível na ordem de microssegundos, o que justifica a escolha da média aritmética de 5 execuções para suavizar esses *outliers* (valores fora da curva).

5. CONSIDERAÇÕES FINAIS:

Os experimentos refutam a ideia de que existe uma estrutura "melhor" baseada apenas em tempos de busca isolados. A escolha correta depende da relação entre frequência de escrita (inserção/ordenação) e frequência de leitura (busca).

1. **Cenários Dinâmicos (Alta Inserção):** A **Árvore AVL** provou ser a estrutura mais robusta e eficiente. Seu tempo total de operação (construção + busca) foi superior ao combo "Vetor + QuickSort". Além disso, ela permite inserir novos elementos mantendo a eficiência de busca ($O(\log n)$), enquanto o vetor exigiria uma custosa reordenação ou deslocamento de memória ($O(n)$) a cada nova inserção.
2. **Cenários Estáticos (Alta Leitura):** O **Vetor Ordenado** continua sendo uma opção válida, mas **apenas** se o número de buscas for grande o suficiente para justificar o alto custo inicial da ordenação (*Quick Sort*). Para poucas consultas, o custo de pré-processamento torna o vetor ineficiente comparado às árvores.
3. **O Perigo da ABB Simples:** Embora tenha brilhado no cenário aleatório (média de 0,0011ms), a ABB simples mostrou-se instável. No pior caso (dados ordenados), sua busca degradou para **0,0156ms** (quase 10x mais lento que o vetor), reafirmando a necessidade de algoritmos de balanceamento (AVL) para aplicações críticas onde a entrada de dados não é controlada.