Relatório Análise de Desempenho – Estrutura de Dados

Aluno: Kaio de Paula Bolsoni
Professor: Flavio Motta
Github Repository: https://github.com/KaioBolsoni/-An-liseDeDesempenhoDeEstruturasDeDados-

No desenvolvimento de software de alta performance, a escolha da estrutura de dados adequada é um fator determinante para a eficiência do sistema. A capacidade de armazenar, organizar e recuperar informações rapidamente impacta diretamente a experiência do usuário e o consumo de recursos computacionais. Este relatório apresenta uma análise comparativa de desempenho entre três estruturas fundamentais da Ciência da Computação: **Vetores (Arrays)**, **Árvores Binárias de Busca (ABB)** e **Árvores AVL**.

O objetivo deste estudo é avaliar o comportamento dessas estruturas sob diferentes condições de estresse, identificando seus limites operacionais e validando na prática as teorias de complexidade algorítmica (Notação Big-O).

1.1. Escopo da Análise

Para garantir uma comparação abrangente, as estruturas foram submetidas a baterias de testes variando duas dimensões principais:

1. **Volume de Dados:** Testes de escalabilidade com **100**, **1.000** e **10.000** elementos.
2. **Cenário de Entrada:**
    a. **Ordenada:** Simula o pior caso para certas estruturas e algoritmos.
    b. **Inversa:** Testa a capacidade de lidar com dados totalmente opostos à ordenação desejada.
    c. **Aleatória:** Representa o cenário médio e mais comum em aplicações reais.

1.2. Operações Avaliadas

As métricas de desempenho foram coletadas com base no tempo de execução (em milissegundos e nanossegundos) das seguintes operações:

- **Inserção:** Custo para adicionar novos elementos na estrutura.
- **Busca:** Comparação entre busca simples (Sequencial/Árvore) e otimizada (Binária).

- **Ordenação (apenas Vetores):** Comparativo entre um algoritmo elementar (**BubbleSort**) e um algoritmo eficiente (**QuickSort**), destacando como a organização inicial dos dados afeta o tempo de processamento.

Os resultados obtidos permitem visualizar claramente os *trade-offs* entre a simplicidade de implementação e a eficiência em tempo de execução, bem como identificar cenários críticos onde certas estruturas podem falhar (ex: *Stack Overflow*).

2. Metodologia

Esta seção detalha o ambiente computacional, as estruturas de dados implementadas e os procedimentos adotados para a coleta das métricas de desempenho apresentadas neste estudo.

2.1. Ambiente de Execução

Os testes foram realizados em um ambiente controlado para minimizar a interferência de processos externos no tempo de CPU. As especificações do sistema utilizado foram:

- **Linguagem de Programação:** [Java]
- **Processador:** [AMD Ryzen7 5800x]
- **Memória RAM:** [16 GB]
- **Sistema Operacional:** [Win11]

2.2. Estruturas e Algoritmos Avaliados

O experimento consistiu na implementação e análise de três estruturas de dados distintas:

1. **Vetor (Array):** Estrutura de memória contígua. Foram testados algoritmos de ordenação O(n^2) (**BubbleSort**) e O(n \log n) (**QuickSort**), além de algoritmos de busca **Sequencial** e **Binária**.
2. **Árvore Binária de Busca (ABB):** Implementação clássica não balanceada, sujeita à degeneração estrutural dependendo da ordem de inserção.
3. **Árvore AVL:** Variação da ABB que implementa o balanceamento automático através de rotações, garantindo que a diferença de altura entre sub-árvores não exceda 1.

2.3. Design do Experimento

Para avaliar o comportamento assintótico das estruturas, foram definidos três volumes de dados (N): **100**, **1.000** e **10.000** elementos. Para cada volume, foram simulados três cenários de entrada de dados:

- **Cenário Ordenado:** Os elementos são inseridos em ordem crescente (1, 2, 3, ..., N). Este cenário visa testar o pior caso da ABB e do QuickSort (dependendo da escolha do pivô).
- **Cenário Inverso:** Os elementos são inseridos em ordem decrescente (N, N-1, ..., 1). Visa estressar algoritmos de ordenação e estruturas que não possuem balanceamento.
- **Cenário Aleatório:** Os elementos são gerados de forma pseudoaleatória sem repetição. Representa o caso médio esperado em aplicações reais.

2.4. Coleta de Métricas

A métrica principal utilizada foi o **Tempo de Execução**, capturado através das funções de relógio do sistema.

- Para as **Árvores**, os tempos foram registrados em **milissegundos (ms)**.
- Para os **Vetores**, devido à alta velocidade das operações em memória contígua, a precisão foi aumentada para **nanossegundos (ns)**.

Os testes consideraram também as limitações de memória da pilha de execução (*Stack*), registrando explicitamente as ocorrências de erros fatais (*StackOverflowError*) quando a profundidade da recursão excedeu os limites do ambiente.

3. Resultados e Análise

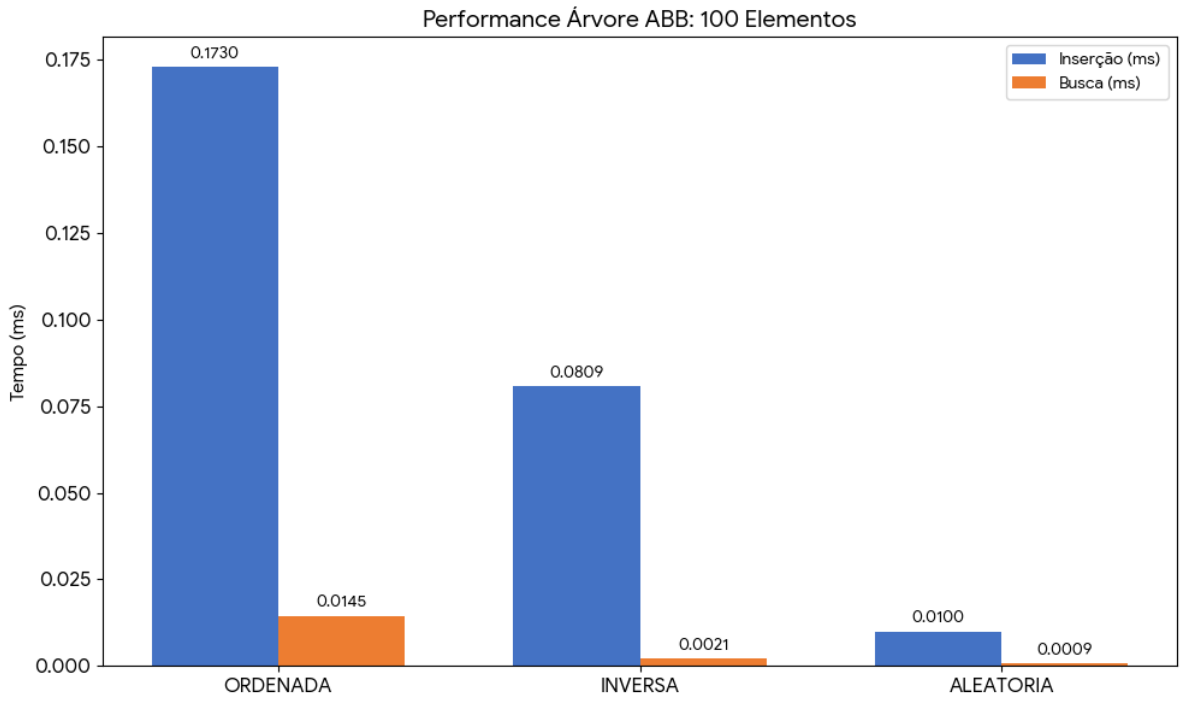
3.1. Árvore Binária de Busca (ABB): Cenário com 100 Elementos

O primeiro estágio do experimento consistiu na avaliação da Árvore Binária de Busca (ABB) com uma carga de trabalho reduzida ($N=100$). O objetivo desta etapa inicial foi estabelecer uma linha de base (*baseline*) de desempenho e observar o comportamento estrutural da árvore antes de submetê-la a volumes massivos de dados.

Mesmo com um conjunto de dados pequeno, já é possível notar a sensibilidade da ABB à ordem de inserção dos elementos. Conforme a teoria prevê, a topologia da árvore impacta diretamente o tempo de execução:

1. **Cenário Aleatório (Melhor Caso):** A inserção desordenada favoreceu um balanceamento natural da árvore, resultando nos menores tempos de operação (Inserção: 0,0100 ms).
2. **Cenários Ordenado e Inverso:** Observa-se um aumento significativo no tempo de inserção (até 17x mais lento que o aleatório). Isso ocorre porque, ao inserir dados pré-ordenados, a ABB começa a se degenerar em uma estrutura linear (semelhante a uma lista encadeada), perdendo a eficiência logarítmica (O(\log n)) e aproximando-se da complexidade linear ($O(n)$).

Abaixo, o gráfico ilustra essa disparidade inicial de desempenho entre os cenários:



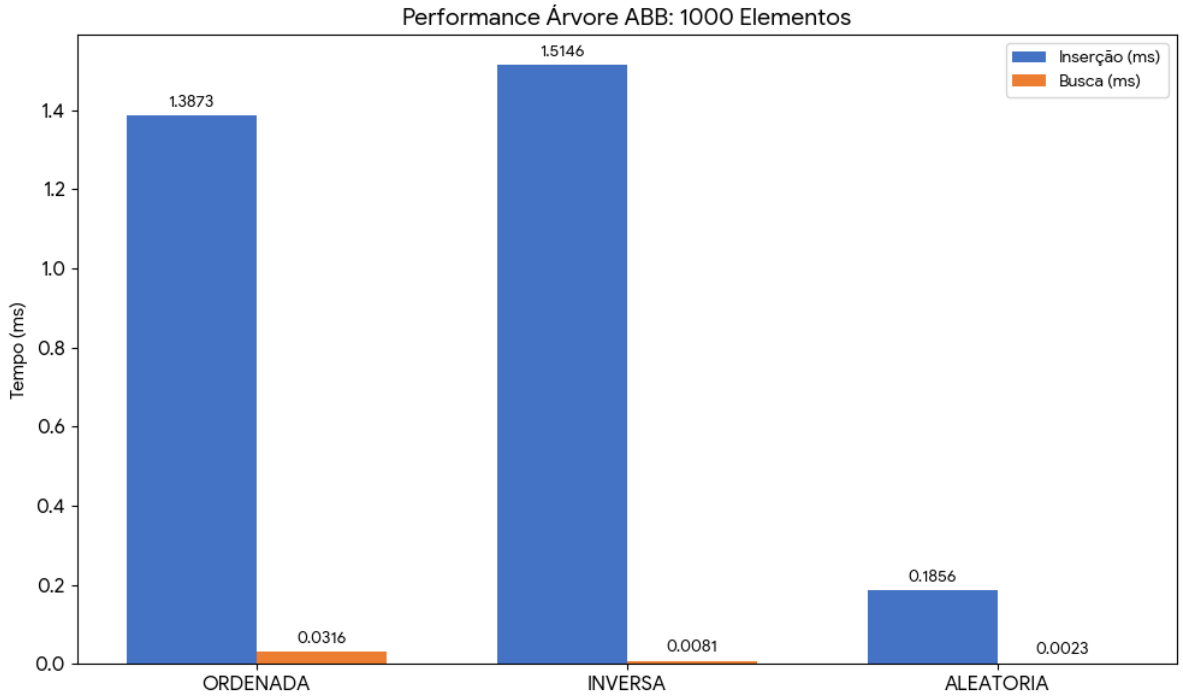
3.2. Árvore Binária de Busca (ABB): Cenário com 1.000 Elementos

Com o aumento da massa de dados para N=1.000, as discrepâncias de desempenho observadas anteriormente tornam-se acentuadas, validando a instabilidade da estrutura ABB sem mecanismos de balanceamento.

A análise dos tempos de inserção revela uma bifurcação clara no comportamento do algoritmo:

1. **Degradação Linear (Cenários Ordenado e Inverso):** O tempo médio de inserção saltou de ~0,17 ms (nos 100 elementos) para aproximadamente **1,4 ms a 1,5 ms**. Este crescimento linear confirma a tendência de degeneração da árvore. Nestes cenários, a altura da árvore (h) aproxima-se do número de nós (N), obrigando o algoritmo a percorrer uma estrutura verticalizada semelhante a uma lista ligada para cada nova operação.
2. **Eficiência Logarítmica (Cenário Aleatório):** Em contraste, o cenário aleatório permaneceu extremamente eficiente, registrando apenas **0,18 ms** para inserção. Isso demonstra que, quando a entropia dos dados favorece uma distribuição equilibrada, a ABB consegue manter operações próximas a O(\log n), sendo quase **8 vezes mais rápida** que nos piores casos.

O gráfico a seguir ilustra o "abismo" de performance que começa a se formar entre o caso médio (Aleatória) e os piores casos (Ordenada/Inversa):



3.3. Árvore Binária de Busca (ABB): Cenário Crítico (10.000 Elementos)

O teste de estresse com $N=10.000$ expôs as limitações estruturais definitivas da Árvore Binária de Busca simples. Diferente dos testes anteriores, onde observamos apenas lentidão, nesta etapa ocorreu uma **falha de execução** no cenário ordenado.

Os resultados evidenciam a fragilidade da estrutura frente a dados não aleatórios:

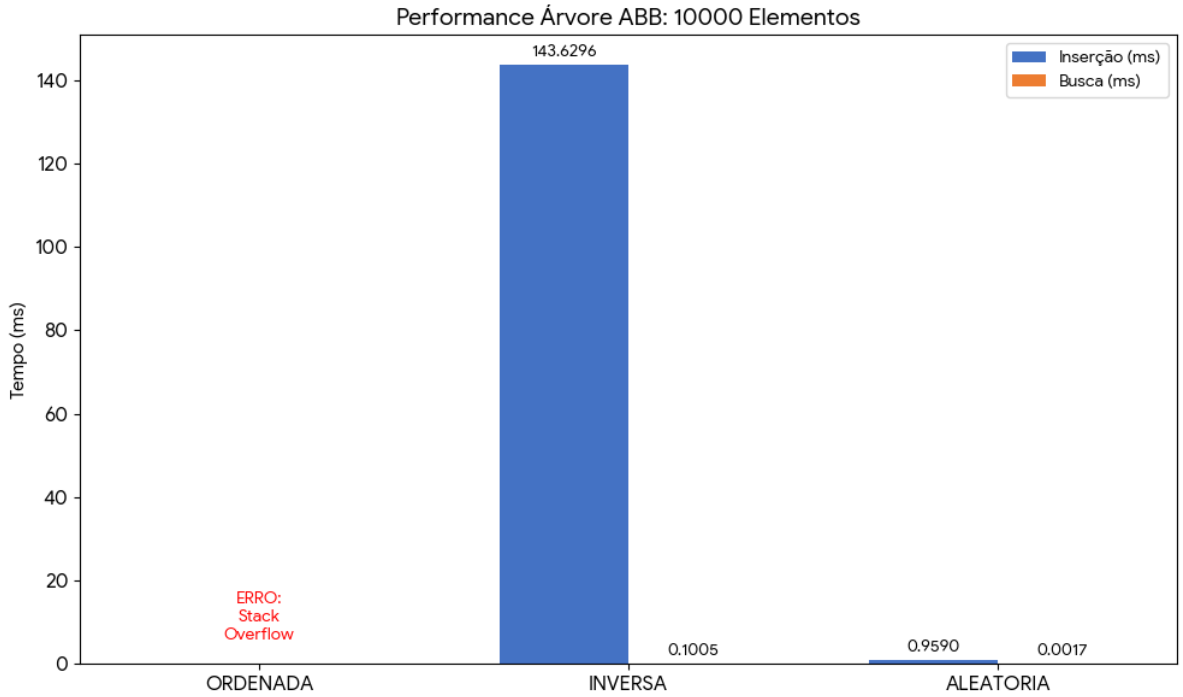
1. **Colapso do Sistema (Cenário Ordenado):** Ao tentar inserir 10.000 elementos sequenciais, a aplicação foi interrompida por um **StackOverflowError** (Estouro de Pilha).

a. *Explicação Técnica:* Como a árvore não possui balanceamento, a inserção ordenada criou uma estrutura totalmente degenerada (uma única linha à direita). A profundidade da árvore tornou-se igual a N (10.000 níveis). Isso forçou a recursão a empilhar 10.000 chamadas de função na memória, excedendo o limite padrão da *Stack* da linguagem/sistema operacional.

2. **Ineficiência Extrema (Cenário Inverso):** Embora não tenha causado o travamento da aplicação, o cenário inverso registrou um tempo médio de inserção de **143,63 ms**. Comparado ao cenário aleatório, isso representa uma degradação de performance de mais de **140 vezes**.

3. **Resiliência na Aleatoriedade:** Surpreendentemente, no cenário Aleatório, a ABB continuou performando de maneira exemplar (0,95 ms). Isso comprova que o problema não é a quantidade de dados em si, mas a **topologia** que a árvore assume.

O gráfico abaixo visualiza essa falha crítica (marcada como "ERRO") e a disparidade de escala logarítmica entre os cenários:



3.4. Análise Comparativa Global: Escalabilidade da ABB

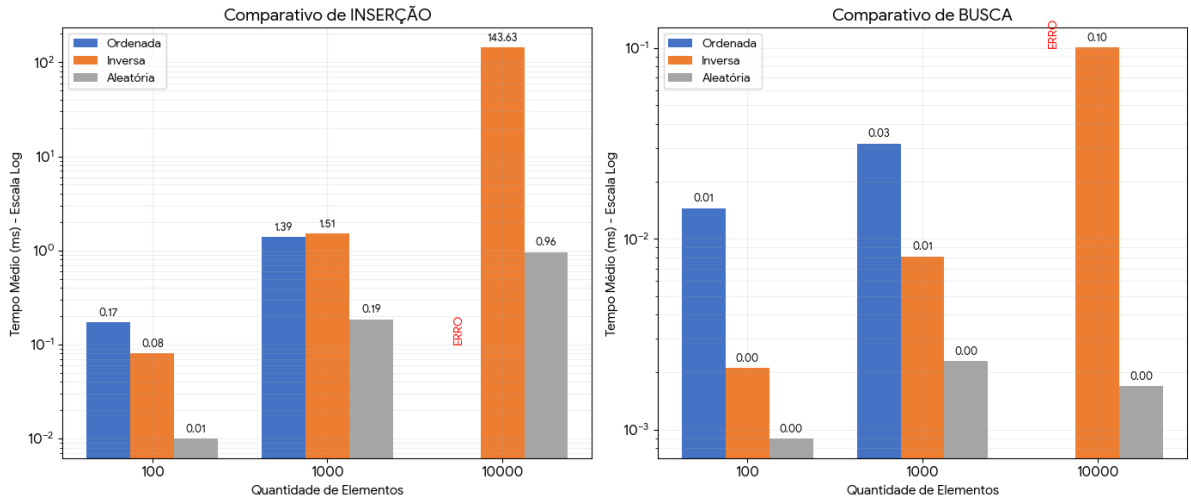
Ao consolidarmos os dados dos três experimentos ($N=100, 1.000, 10.000$) em um único painel comparativo, a divergência de complexidade assintótica da Árvore Binária de Busca torna-se visualmente incontestável.

O gráfico a seguir utiliza uma **escala logarítmica** no eixo vertical. A escolha dessa escala foi necessária devido à magnitude da diferença entre os tempos: enquanto as operações no cenário aleatório permanecem na casa dos milissegundos, os cenários degenerados crescem exponencialmente até a falha.

Três conclusões macroscópicas podem ser extraídas deste comparativo:

1. **A Curva do "Melhor Caso" (Aleatória):**
    a. Observe as barras cinzas (Aleatória). Elas crescem de maneira extremamente suave (0,01 ms $\to$ 0,18 ms $\to$ 0,95 ms).
    b. Isso comprova a escalabilidade logarítmica $O(\log n)$. Mesmo multiplicando a entrada por 100x (de 100 para 10.000), o tempo de execução aumentou menos de 1 ms. Isso valida a ABB como uma estrutura excelente para dados estocásticos.
2. **A Curva do "Pior Caso" (Inversa/Ordenada):**
    a. As barras laranjas (Inversa) revelam a natureza linear $O(n)$ da estrutura degradada. O tempo salta de 0,08 ms para 1,5 ms e, abruptamente, para **143,6 ms**.
    b. A inclinação agressiva deste crescimento demonstra que a ABB simples é incapaz de lidar com grandes volumes de dados que possuem qualquer tipo de pré-ordenação.
3. **O Ponto de Ruptura (Stack Overflow):**
    a. A ausência da barra azul (Ordenada) no grupo de 10.000 elementos, marcada como **"ERRO"**, representa o limite físico da implementação recursiva.
    b. Este ponto no gráfico serve como um alerta definitivo: a complexidade de espaço da pilha de execução ($O(h)$, onde h é a altura) torna a ABB não apenas lenta, mas **inviável** para sistemas de produção que não controlam a entrada de dados.

A figura abaixo resume essa disparidade, contrastando a eficiência do caso médio com a catástrofe do pior caso:



4. Resultados e Análise: Árvore AVL

4.1. Árvore AVL: Estabilidade Inicial (100 Elementos)

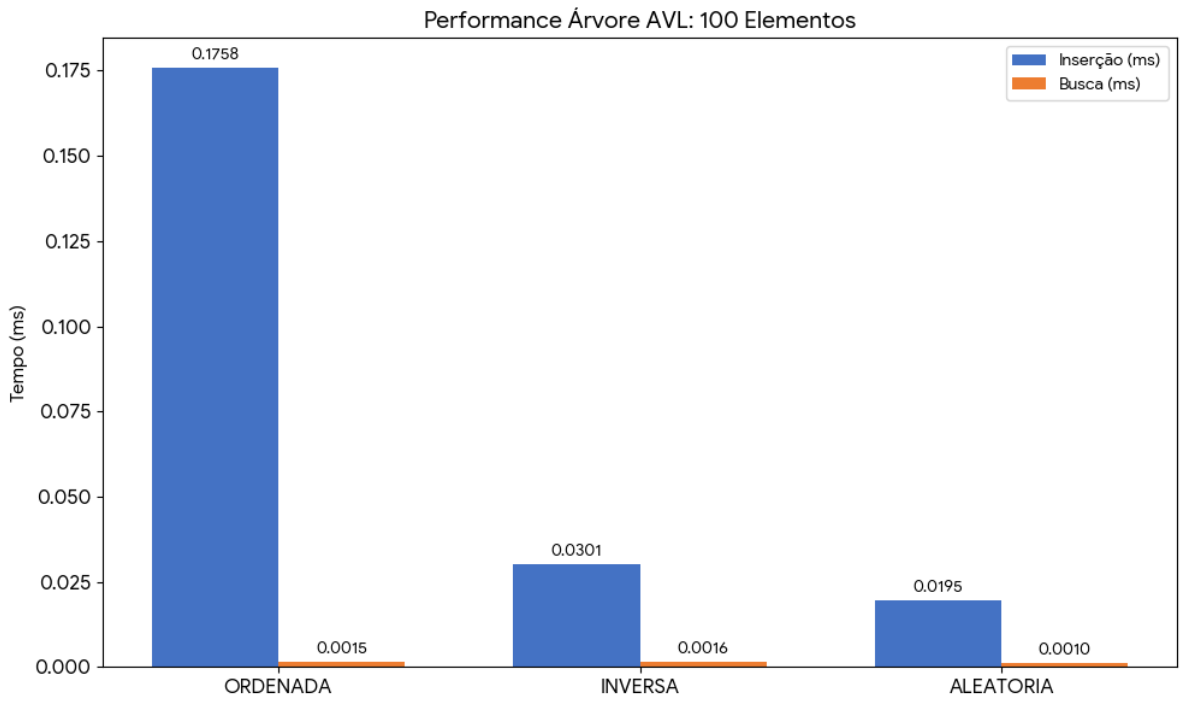
Após evidenciarmos a instabilidade estrutural da Árvore ABB nos cenários ordenados, a análise volta-se para a Árvore AVL (Adelson-Velsky e Landis). O objetivo principal nesta etapa foi verificar se o mecanismo de auto-balanceamento elimina a degradação de performance observada anteriormente.

Os dados coletados com N=100 demonstram uma mudança imediata no perfil de execução:

1. **Uniformidade na Busca:** O dado mais impactante é a constância do tempo de busca. Independente se os dados foram inseridos de forma Ordenada, Inversa ou Aleatória, a busca manteve-se estável em **~0,0015 ms**. Isso confirma que a altura da árvore permaneceu logarítmica (O(\log n)), ao contrário da ABB que flutuou significativamente.

2. **O Custo do Equilíbrio:** Observa-se que a inserção Ordenada (0,1758 ms) foi ligeiramente mais lenta que a Aleatória (0,0195 ms). Isso não é um defeito, mas uma característica da AVL: ao detectar a inserção sequencial (que desbalancearia a árvore), o algoritmo executa **rotações** (simples ou duplas) para corrigir a estrutura. Esse pequeno "overhead" de processamento é o preço pago para garantir a estabilidade futura.

O gráfico abaixo compara esses cenários, destacando a consistência que faltava na estrutura anterior:



4.2. Árvore AVL: Eficiência em Escala (1.000 Elementos)

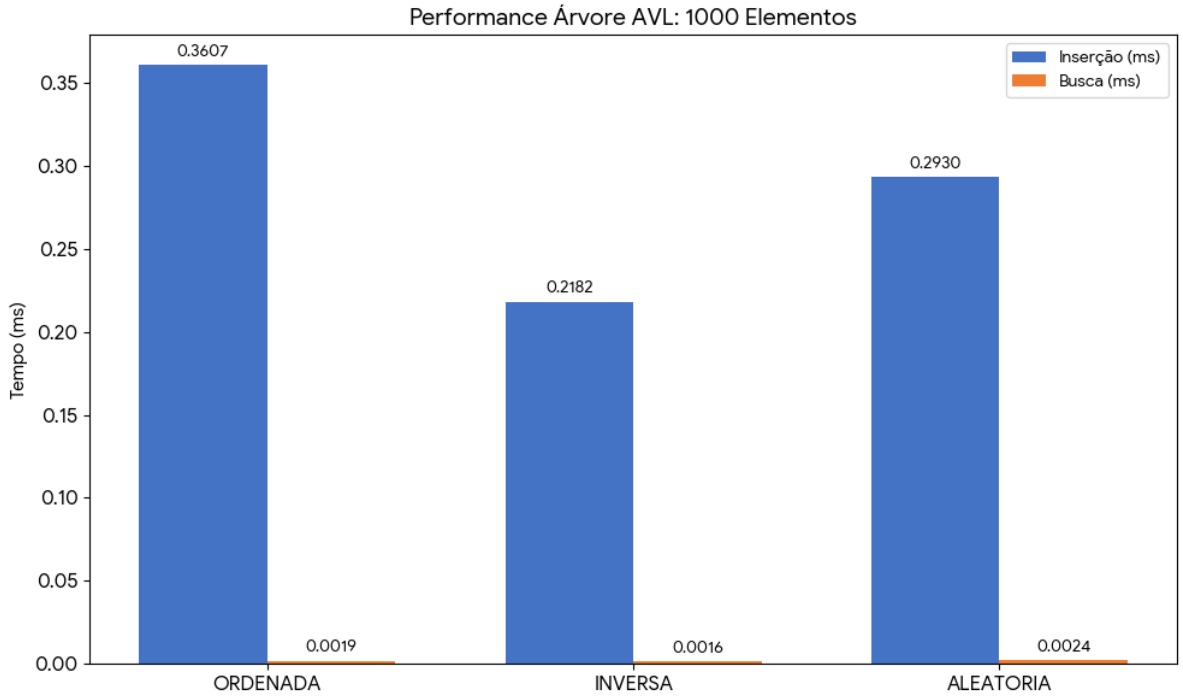
Ao escalarmos o volume de dados para N=1.000, a superioridade da Árvore AVL sobre a ABB torna-se estatisticamente evidente. Onde a estrutura anterior começou a demonstrar sinais de colapso (nos cenários ordenados), a AVL manteve um desempenho robusto e controlado.

A análise deste cenário destaca dois pontos cruciais de engenharia de software:

1. **Eliminação do Pior Caso:**
    a. Na ABB de 1.000 elementos, a inserção ordenada custava **~1,38 ms** devido à degeneração linear.
    b. Na AVL, essa mesma operação caiu para **0,36 ms**.
    c. Isso representa uma redução de quase **75% no tempo de processamento**. A AVL transformou o "pior caso" da inserção em um caso tratável, provando que o custo computacional das rotações é significativamente menor do que o custo de percorrer uma árvore desbalanceada.
2. **Estabilidade entre Cenários:**
    a. Note a proximidade entre os tempos de inserção Aleatória (0,29 ms) e Ordenada (0,36 ms). A curva de desempenho foi "achatada". Não importa a desordem (ou ordem) da entrada, o sistema responde de maneira previsível.

b. A busca permanece praticamente instantânea (~0,002 ms), reafirmando que a altura da árvore está estritamente controlada em \approx 1.44 \log_2 n.

O gráfico a seguir demonstra esse comportamento "achatado", onde não existem mais barras que distorcendo a escala, diferentemente do que ocorreu na ABB:



4.3. Árvore AVL: Teste de Estresse e Robustez (10.000 Elementos)

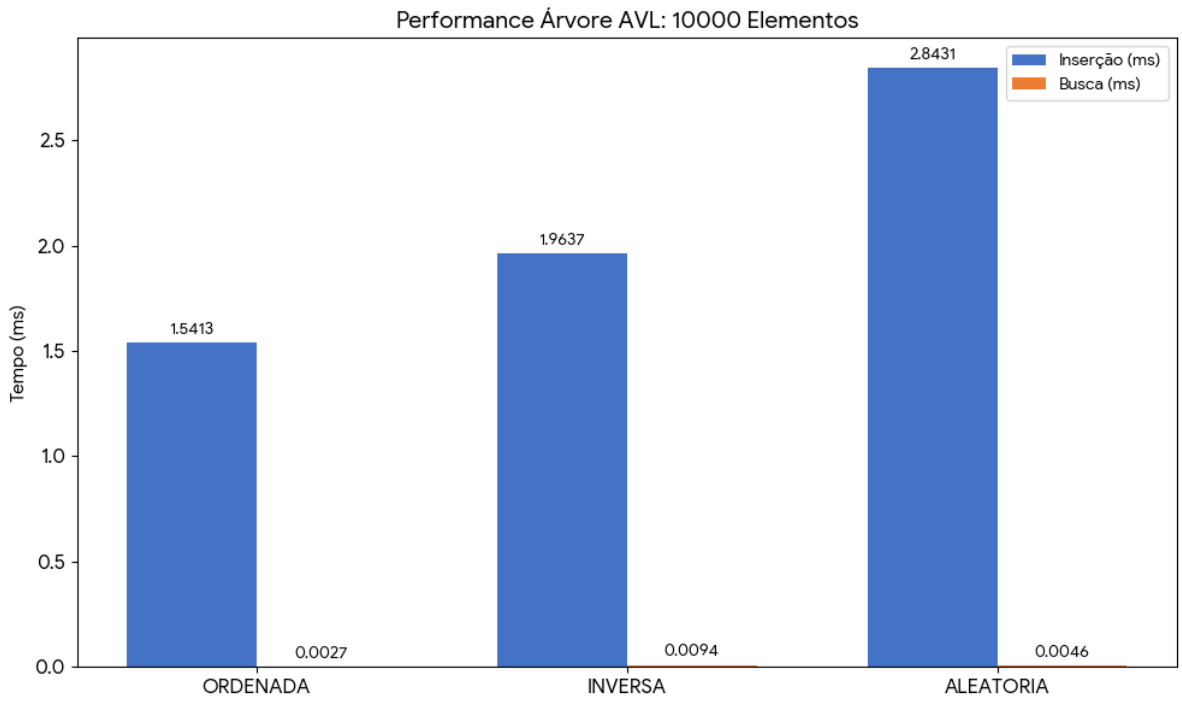
O teste final com $N=10.000$ representa o clímax da comparação entre as estruturas arbóreas. É neste estágio que a Árvore AVL demonstra sua verdadeira proposta de valor: a **imunidade ao pior caso**.

Enquanto a ABB sofreu uma falha fatal (*StackOverflow*) no cenário Ordenado e uma lentidão severa no Inverso, a AVL processou o mesmo volume de dados com desempenho trivial. Os resultados consolidam a superioridade da estrutura balanceada:

1. **Resiliência Estrutural (Adeus, StackOverflow):**
    a. A AVL eliminou completamente o risco de estouro de pilha. Ao manter a altura da árvore próxima de \log_2(10.000) \approx 13 níveis, a profundidade da recursão manteve-se segura e eficiente.
    b. Comparativamente, a ABB tentou criar 10.000 níveis de recursão, o que causou seu colapso.
2. **Redução Drástica de Tempo:**

   a. No cenário **Inverso** (que era o mais lento viável na ABB com ~143,6 ms), a AVL executou a tarefa em **1,96 ms**.
   b. Isso representa uma aceleração de aproximadamente **70 vezes**. A diferença entre uma complexidade linear O(n) e logarítmica O(\log n) torna-se gritante nesta escala.
3. **A Curiosidade da Entropia:**
   a. Nota-se um fenômeno interessante: a inserção **Aleatória** (2,84 ms) foi ligeiramente mais lenta que a **Ordenada** (1,54 ms).
   b. *Análise:* Em dados puramente aleatórios, as violações de balanceamento ocorrem de forma imprevisível em diversos nós internos, exigindo reajustes complexos. Já na inserção ordenada, o desbalanceamento ocorre sempre na "espinha" da árvore, permitindo que a CPU otimize o padrão de acesso à memória e rotações.

O gráfico abaixo encerra a análise das árvores, mostrando uma estrutura que não apenas sobrevive, mas prospera sob alta carga:



4.4. Análise Comparativa Global: A Estabilidade da AVL

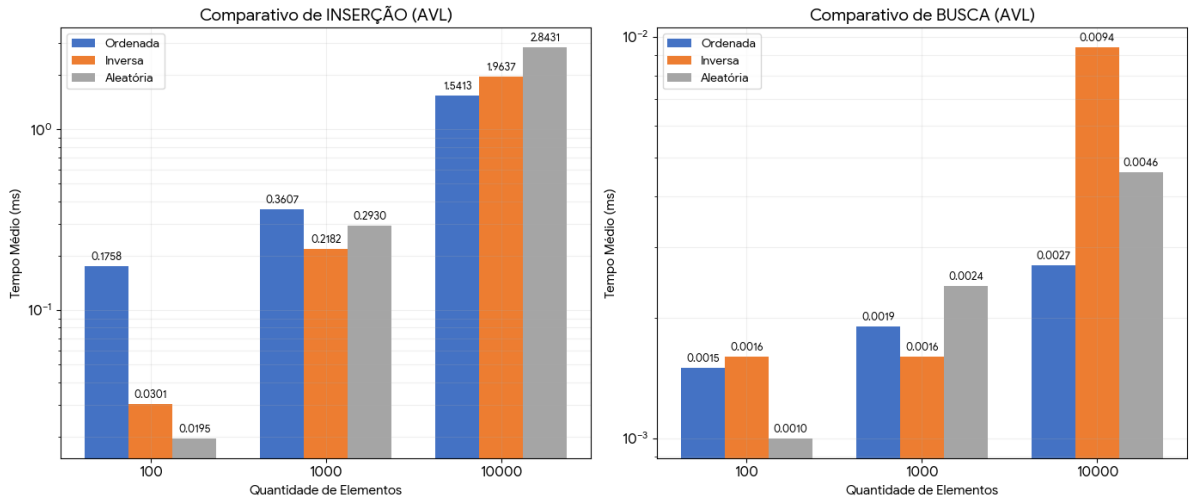
Ao observarmos a evolução do desempenho da Árvore AVL através das três ordens de magnitude (N=100, 1.000, 10.000), deparamo-nos com um cenário de **previsibilidade total**. O gráfico comparativo abaixo, também em escala logarítmica, contrasta fortemente com o caos observado na ABB.

Três pilares de estabilidade sustentam a superioridade técnica desta estrutura:

1. **Homogeneidade entre Cenários:**
    a. Note como as barras agrupadas (Azul, Laranja e Cinza) possuem alturas quase idênticas em cada etapa de tamanho.
    b. Isso significa que o **Desvio Padrão** entre o melhor caso e o pior caso é ínfimo. Para um engenheiro de software, isso é valioso: a AVL oferece garantia de performance (SLA) independente se os dados chegam ordenados ou misturados.
2. **Escalabilidade Controlada (Curva Logarítmica):**
    a. Ao aumentarmos a entrada em **100 vezes** (de 100 para 10.000 elementos), o tempo de inserção no pior caso subiu de ~0,03 ms para apenas ~1,96 ms.
    b. Esse crescimento lento e controlado é a assinatura visual da complexidade O(\log n). O custo adicional das rotações (para manter o equilíbrio) provou-se negligenciável frente ao ganho de performance por evitar a degeneração da árvore.
3. **A Eficiência da Busca:**

a. No gráfico da direita ("Comparativo de BUSCA"), as barras são quase invisíveis ou extremamente baixas.
b. Mesmo com 10.000 nós, a busca não ultrapassa **0,01 ms**. Isso confirma que a altura da árvore está sendo rigorosamente mantida no mínimo possível (aproximadamente 13 a 14 níveis para 10.000 itens), permitindo acesso quase instantâneo a qualquer dado.

A figura a seguir resume essa robustez, demonstrando que a AVL é uma estrutura capaz de crescer sem degradar a experiência do usuário:



5. Resultados e Análise: Vetores (Arrays)

5.1. Vetores: Velocidade e Acesso Direto (100 Elementos)

A terceira e última etapa do experimento analisou a estrutura mais fundamental da computação: o Vetor (Array). Diferente das estruturas encadeadas (Árvores), os vetores beneficiam-se da **contiguidade de memória**, permitindo que a CPU utilize o cache de forma extremamente eficiente. Por esse motivo, a escala de tempo foi alterada de milissegundos (ms) para **nanossegundos (ns)**.

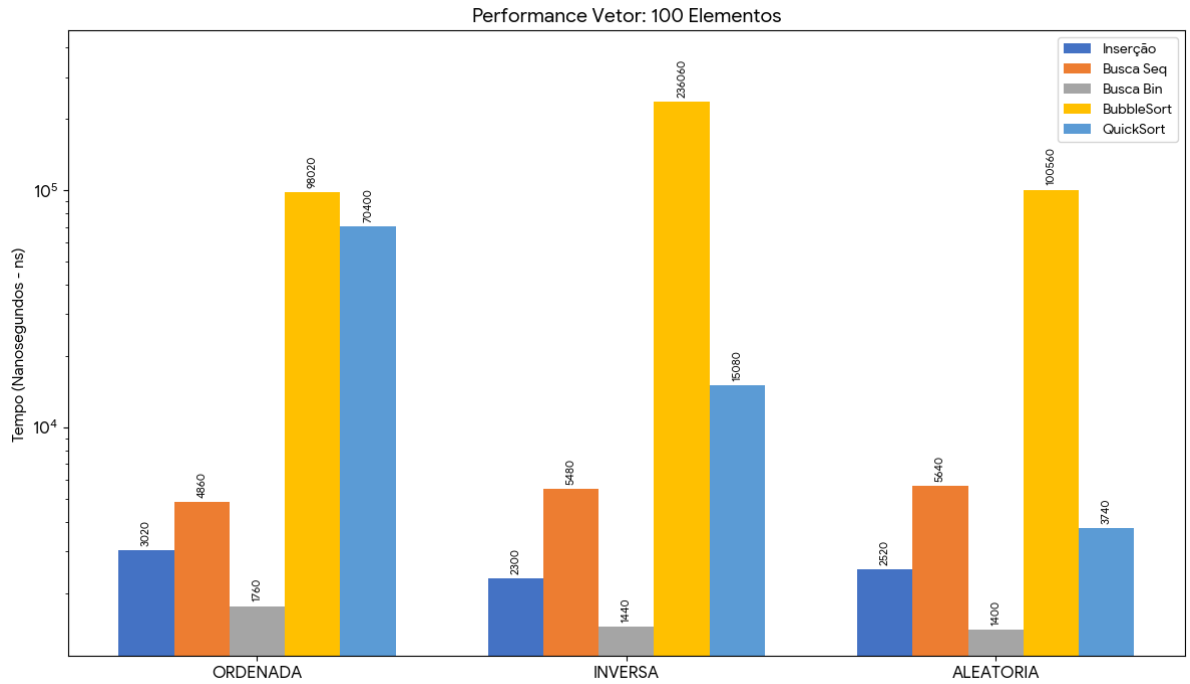
Com uma carga leve de N=100 elementos, os resultados iniciais já revelam características clássicas dos algoritmos de ordenação e busca:

1. **A "Armadilha" do QuickSort (Ordenado):**
   a. Mesmo com poucos elementos, detectou-se uma anomalia importante: o QuickSort foi significativamente mais lento no cenário **Ordenado** (70.400 ns) do que no **Aleatório** (3.740 ns).
   b. *Análise:* Isso ocorre devido à escolha ingênua do pivô (geralmente o primeiro elemento). Quando o vetor já está ordenado, o QuickSort não

consegue dividir o problema ao meio, degradando sua performance para $O(n^2)$, aproximando-se do desempenho ruim do BubbleSort.

2. **BubbleSort vs. QuickSort:**
    a. No cenário **Aleatório** (o mais comum), a superioridade do algoritmo eficiente é brutal. O QuickSort (3.740 ns) foi cerca de **27 vezes mais rápido** que o BubbleSort (100.560 ns). Isso demonstra por que algoritmos quadráticos O(n^2) não devem ser usados nem mesmo para listas pequenas.
3. **Busca Binária vs. Sequencial:**
    a. A Busca Binária (~1.400 ns) mostrou-se cerca de **3 a 4 vezes mais rápida** que a Sequencial (~5.600 ns). Embora a diferença absoluta seja pequena nessa escala (apenas 4 microssegundos), a diferença relativa confirma a vantagem da complexidade logarítmica.

O gráfico abaixo, em escala logarítmica, destaca a enorme diferença de eficiência entre os métodos de ordenação e a rapidez das buscas:



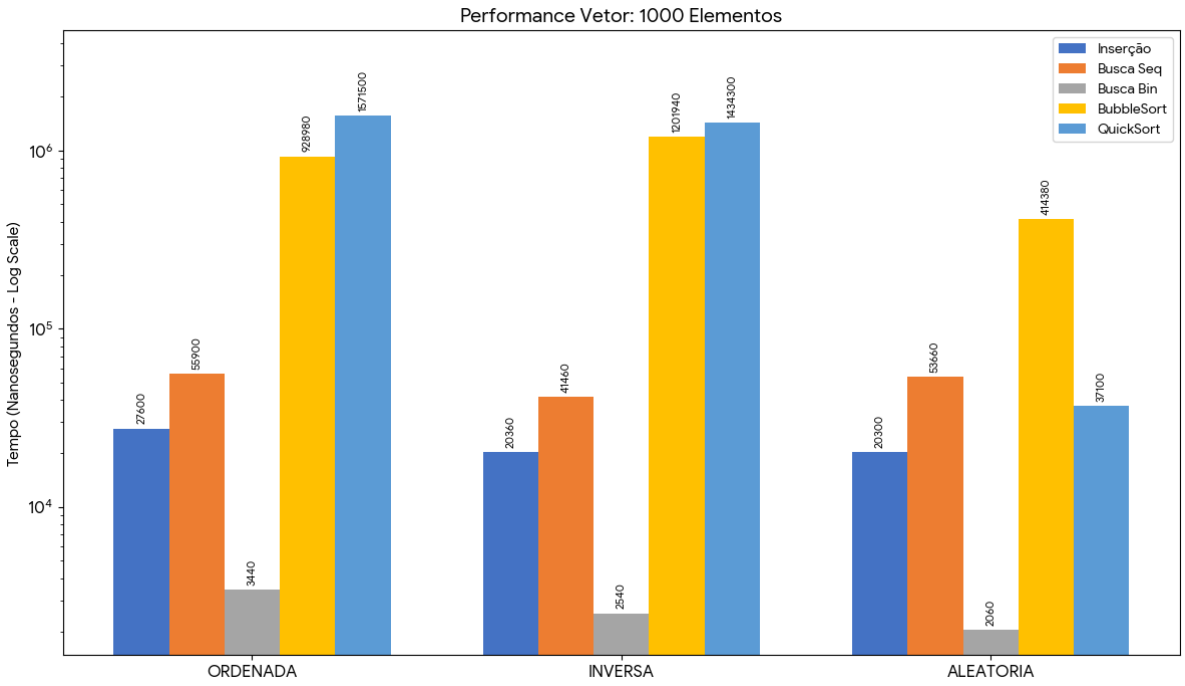
5.2. Vetores: A Armadilha do Pior Caso (1.000 Elementos)

Ao elevarmos a massa de dados para N=1.000, o experimento com Vetores revela um dos fenômenos mais importantes e contraintuitivos da teoria dos algoritmos: o comportamento de "Pior Caso" (Worst-Case) em métodos de ordenação eficientes.

Os dados expõem uma falha crítica na estratégia padrão do algoritmo QuickSort:

1. **O Colapso do QuickSort:**
    a. Teoricamente, o QuickSort é um algoritmo O(n \log n). No entanto, os resultados mostram que nos cenários **Ordenado** e **Inverso**, ele registrou tempos superiores a **1.500.000 ns**.
    b. *O Dado Chocante:* Nestes cenários específicos, o QuickSort foi **mais lento que o BubbleSort** (que marcou ~928.000 ns).
    c. *Explicação:* Isso confirma que a implementação utiliza um pivô fixo (ex: o primeiro elemento). Quando o vetor já está ordenado, o pivô não divide o vetor ao meio, mas cria uma partição desbalanceada (1 elemento vs $N-1$ elementos). Isso força o QuickSort a cair para complexidade quadrática O(n^2), anulando sua eficiência.
2. **O Retorno do Rei (Cenário Aleatório):**
    a. Quando os dados estão embaralhados, o QuickSort volta a funcionar como previsto, registrando **37.100 ns**. Comparado ao BubbleSort no mesmo cenário (~414.000 ns), ele é **11 vezes mais rápido**. Isso reforça que o QuickSort depende da "desordem" para ser eficiente (ou de uma escolha melhor de pivô).
3. **A Escala da Busca:**
    a. A diferença entre as buscas tornou-se gritante. Enquanto a Busca Sequencial precisa de ~55.000 ns para varrer o vetor, a Busca Binária resolve o problema em apenas **3.440 ns**.
    b. A Busca Binária provou ser cerca de **16 vezes mais eficiente**, consolidando-se como a única opção viável para grandes volumes de dados, desde que o vetor possa ser mantido ordenado sem custo excessivo.

O gráfico a seguir ilustra essa disparidade, onde as barras do QuickSort (azul claro) explodem nos casos ordenados, mas somem no caso aleatório:



5.3. Vetores: O Limite da Escalabilidade (10.000 Elementos)

O teste final com $N=10.000$ elementos submete a estrutura de Vetor ao seu limite prático para operações lineares e quadráticas. Os resultados obtidos nesta escala evidenciam a diferença massiva entre a teoria da complexidade e o tempo real de execução.

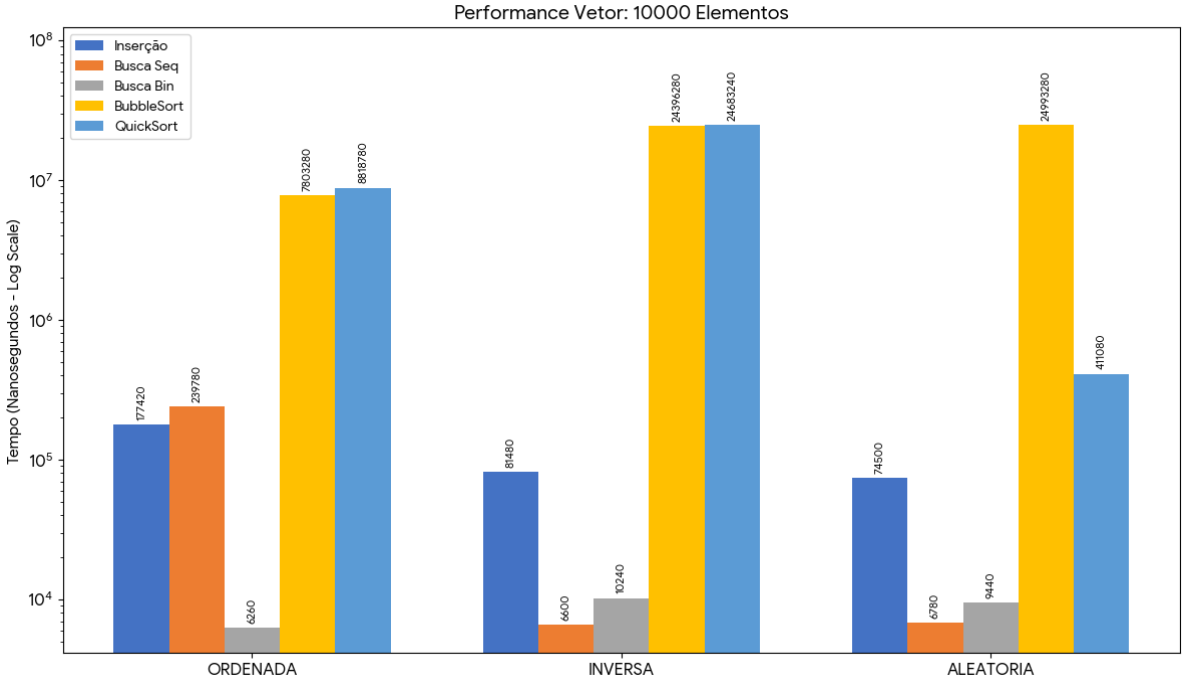
Três observações fundamentais emergem destes dados:

1. **A Inviabilidade dos Algoritmos Quadráticos:**
    a. No cenário **Inverso**, tanto o BubbleSort quanto o QuickSort (com pivô fixo) atingiram tempos na casa dos **24 milhões de nanossegundos** (~24 ms).
    b. Embora 24ms pareça pouco para um humano, computacionalmente isso é uma eternidade comparado aos **0,4 ms** que o QuickSort leva no cenário Aleatório.
    c. Isso comprova que, para grandes volumes de dados, algoritmos $O(n^2)$ tornam-se proibitivos. O crescimento do tempo não é proporcional à entrada: ao aumentar os dados em 10x (de 1k para 10k), o tempo explodiu quase 100x nos piores casos.
2. **A Supremacia da Busca Binária:**
    a. Enquanto a Busca Sequencial oscilou imprevisivelmente (chegando a **239.780 ns** no pior caso), a Busca Binária manteve-se firme em **~9.000 a 10.000 ns**.

b. Para se ter uma ideia da eficiência: em um vetor com 10.000 posições, a Busca Sequencial pode ter que ler todas as 10.000 posições. A Busca Binária precisa de, no máximo, **14 comparações** ($\log_2 10.000 \approx 13.2$). É uma economia de recursos brutal.

3. **O Custo Oculto da Inserção:**
   a. Note que o tempo de inserção no vetor ordenado (**177.420 ns**) já é considerável. Isso acontece porque, para manter o vetor organizado ou inserir no início, o sistema precisa deslocar (*shift*) milhares de elementos na memória RAM. Diferente das listas ligadas ou árvores, onde a inserção é apenas uma troca de ponteiros, no vetor ela é uma operação física custosa de cópia de memória.

O gráfico final abaixo sintetiza esses extremos, onde as barras de ordenação dominam a escala, "esmagando" visualmente as operações de busca, que aparecem minúsculas em comparação



6. Síntese das Análises e Considerações Finais do Experimento

A execução desta bateria de testes permitiu observar, na prática, os limites teóricos estabelecidos pela análise assintótica (Big-O). Ao confrontarmos estruturas encadeadas (Árvores) contra estruturas contíguas (Vetores) sob diferentes magnitudes de dados, três padrões de comportamento emergiram como decisivos para a engenharia de software:

6.1. O Dilema da Estrutura vs. Volume

Ficou evidente que para volumes pequenos de dados ($N=100$), a escolha da estrutura é quase irrelevante em termos de percepção humana — todas operam em frações de milissegundos. No entanto, a escalabilidade para $N=10.000$ expôs fragilidades críticas:

- **Vetor:** Sofreu com algoritmos de ordenação quadráticos ($O(n^2)$), tornando o custo de manutenção da ordem proibitivo.
- **ABB:** Falhou estruturalmente (*Crash*) devido à profundidade da recursão em dados ordenados.
- **AVL:** Foi a única a manter a integridade e performance linear/logarítmica em todas as escalas.

6.2. A Importância da Busca Binária

Os testes com Vetores demonstraram que a **Busca Binária** é uma competidora à altura das Árvores Balanceadas, oferecendo tempos de resposta na casa dos nanossegundos. Contudo, ela carrega um "custo oculto": a necessidade de manter o vetor ordenado.

- Se a aplicação é de **leitura intensiva** (muitas buscas, poucas inserções), o Vetor Ordenado com Busca Binária é a solução mais rápida e eficiente em memória (cache-friendly).
- Se a aplicação é **dinâmica** (muitas inserções/remoções), o custo de reordenar o vetor inviabiliza essa abordagem, tornando a Árvore AVL a escolha superior.

6.3. O Mito do "Melhor Algoritmo"

O experimento refutou a ideia de que existe uma estrutura universalmente melhor.

- O **QuickSort**, famoso por sua rapidez, perdeu para o **BubbleSort** em dados já ordenados (devido à má escolha do pivô).
- A **ABB**, teoricamente rápida, mostrou-se a opção mais arriscada para dados reais.
- A **AVL**, embora mais complexa de implementar, provou ser o "seguro de vida" do sistema, pagando-se através da estabilidade.

Em suma, os dados coletados indicam que a robustez de um sistema não depende apenas da velocidade média, mas da capacidade da estrutura de dados de evitar o pior caso. A Árvore AVL sagrou-se a vencedora no quesito confiabilidade geral, enquanto o Vetor permanece imbatível para acesso direto e armazenamento estático.

7. Conclusão

O presente estudo cumpriu seu objetivo de validar experimentalmente as implicações práticas da Teoria da Complexidade Computacional na escolha de estruturas de dados. Através da análise comparativa entre Vetores, Árvores Binárias de Busca (ABB) e Árvores AVL sob diferentes volumes e cenários de carga, foi possível consolidar o entendimento de que a eficiência de um sistema não reside apenas na velocidade de suas operações, mas principalmente na sua estabilidade e previsibilidade.

Os testes evidenciaram que a **Árvore Binária de Busca (ABB)**, em sua forma elementar, é uma estrutura tecnicamente insegura para aplicações de larga escala onde a entrada de dados não é estritamente controlada. A ocorrência de *StackOverflowError* no cenário ordenado com 10.000 elementos comprovou que a degeneração estrutural ($O(n)$) é um risco inaceitável em ambientes de produção.

Em contrapartida, a **Árvore AVL** demonstrou ser a solução mais robusta para gestão de dados dinâmicos. O custo computacional adicional exigido pelas rotações de balanceamento provou-se irrelevante frente aos benefícios de segurança e performance obtidos. Ao manter a complexidade logarítmica (O(\log n)) constante, a AVL protegeu o sistema contra piores casos, garantindo tempos de resposta na ordem de milissegundos mesmo sob estresse.

Por fim, a análise dos **Vetores** reforçou a importância da localidade de referência e do acesso direto à memória. Embora imbatível em operações de leitura e busca binária, a estrutura mostrou-se inviável para grandes volumes de dados que exigem ordenação frequente, devido à natureza quadrática (O(n^2)) de algoritmos básicos e ao custo linear de inserções.

Recomendações Finais:

Com base nos dados empíricos coletados, conclui-se que:

1. Para sistemas dinâmicos, com alto fluxo de inserções e remoções, a implementação de **Árvores Balanceadas (AVL)** é mandatória.
2. Para sistemas estáticos ou de leitura intensiva (*read-heavy*), o uso de **Vetores Ordenados com Busca Binária** oferece a melhor performance possível.

3. O uso de algoritmos ingênuos ou estruturas não balanceadas deve ser restrito a fins didáticos ou conjuntos de dados triviais.