Centro Universitário FAMINAS - Muriaé
Curso de Análise e Desenvolvimento de Sistemas

Lucas Picanço Santos Ferreira Silva

RELATÓRIO TÉCNICO

ANÁLISE DE DESEMPENHO DE ESTRUTURAS DE DADOS EM JAVA

Link do repositório:
LucasPicancoo/Trabalho---Estruturas-de-Dados-em-Java

1. Metodologia

Os experimentos foram realizados com o objetivo de comparar o desempenho de diferentes estruturas de dados e algoritmos de ordenação. Todos os testes seguiram os mesmos princípios:

1.1 Geração dos Dados

Para cada tamanho **100**, **1000** e **10000**, foram gerados três conjuntos de dados:

- **Crescente**
- **Decrescente**
- **Aleatório**

Esses conjuntos foram utilizados nos testes de inserção em Vetor, ABB e AVL, e como entrada para os algoritmos de ordenação.

1.2 Medição do Tempo

A medição foi feita utilizando uma classe utilitária chamada **Timer**, com base em **System.nanoTime().** A metodologia foi:

1. **Aquecimento da JVM**
   Antes dos testes, um “warm-up” foi executado para reduzir os efeitos do JIT Compiler.
2. **Execução repetida**
   Cada operação foi repetida 5 vezes e o tempo médio foi registrado.
3. **Operações analisadas**
   - **Inserção**
     Vetor, Árvore Binária de Busca (ABB) e Árvore AVL
   - **Busca**
     Busca Sequencial, Busca Binária, ABB e AVL
   - **Ordenação (Vetores)**
     Bubble Sort e QuickSort

1.3 Hardware e Condições

Os testes foram executados:

- Máquina local
  - Processador: Intel Core i7-14700K
  - Memória Ram: 16GB DDR5 6000MHz
  - Sistema Operacional: Windows 11

- Nenhuma outra aplicação pesada rodando
- Cada teste realizado individualmente

2. Tabelas de Resultados

Obs: Como se tratam de nanosegundos é comum haver um pouco de ruído entre os resultados.

Inserção - Vetor			
Tamanho	Crescente(ns)	Decrescente(ns)	Aleatório(ns)
100	220 ns	260 ns	260 ns
1000	320 ns	280 ns	280 ns
10000	3120 ns	3060 ns	3120 ns

A inserção final do vetor possui complexidade média **O(1))**. O leve aumento no tempo observado conforme o tamanho cresce deve-se a fatores de otimização dos métodos, e não à complexidade algorítmica.

Inserção - ABB			
Tamanho	Crescente(ns)	Decrescente(ns)	Aleatório(ns)
100	37640 ns	32060 ns	4340 ns
1000	1919960 ns	3598160 ns	48340 ns
10000	226155440 ns	419119940 ns	595380 ns

Quando os dados estão ordenados, a ABB **degenera em uma lista encadeada**, pois cada novo elemento sempre é inserido no mesmo lado da árvore (sempre à direita ou sempre à esquerda).

Isso produz uma árvore com altura **O(n)** e, portanto, um tempo de inserção também **O(n)**.

Os casos **crescente** e **decrescente** são *muito mais lentos*, com tempos centenas de vezes maiores.

A inserção **aleatória** mantém a árvore mais equilibrada por acaso, aproximando-se de uma altura **O(log n)**, resultando em um tempo muito menor.

Os resultados confirmam claramente o pior caso **O(n)** da ABB para dados ordenados.

Inserção - AVL			
Tamanho	Crescente(ns)	Decrescente(ns)	Aleatório(ns)
100	41760 ns	34320 ns	20020 ns
1000	155660 ns	119280 ns	78960 ns
10000	395660 ns	447820 ns	838440 ns

A AVL realiza rotações automáticas sempre que necessário para manter a árvore **estritamente balanceada**, com altura garantida em **O(log n)**.

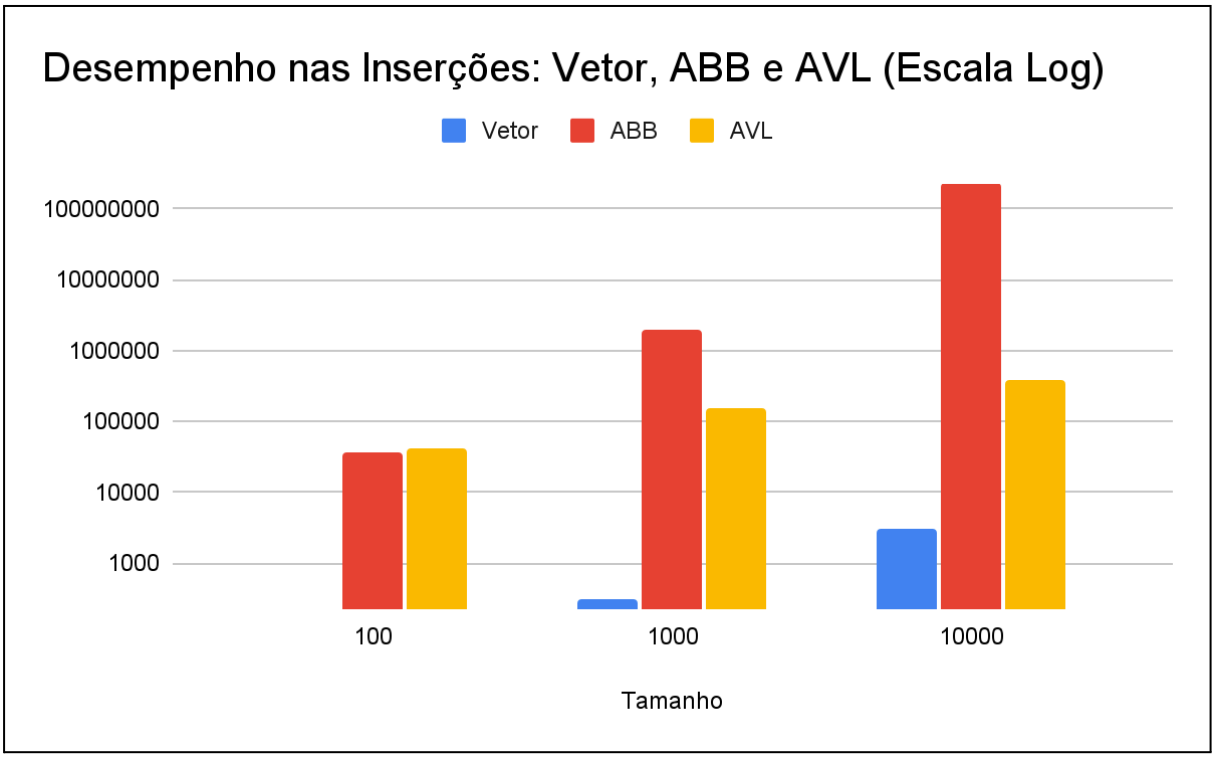
Por isso, diferentemente da ABB, ela **não se degrada** mesmo quando os dados já vêm ordenados.

A diferença entre as três ordens existe, mas é **pequena**.

Inserções aleatórias às vezes exigem **mais rotações**, por isso podem ficar mais lentas que crescente/decrescente.

Mesmo no pior caso prático, o tempo continua extremamente menor que os casos degenerados da ABB.

Os resultados confirmam o comportamento esperado da AVL, mantendo sempre altura **O(log n)** e evitando degradação, mesmo com dados ordenados.



Obs: Os vetores usados nos testes de busca foram gerados em ordem crescente.

Busca Sequencial - Vetor							
Tamanho	Primeiro elemento	Último elemento	Elemento do meio	Aleatório 01	Aleatório 02	Aleatório 03	Inexistente
100	120 ns	760 ns	440 ns	680 ns	240 ns	220 ns	600 ns
1000	100 ns	5780 ns	2960 ns	1700 ns	5820 ns	4360 ns	5860 ns
10000	100 ns	55780 ns	11440 ns	13240 ns	10700 ns	8320 ns	13720 ns

A busca sequencial tem complexidade linear O(N), já que ela verifica cada posição do vetor até encontrar o valor desejado. Isso faz com que o tempo de execução cresça proporcionalmente ao tamanho da entrada. Nos testes que fiz, esse comportamento ficou claro nos três cenários analisados.

1. Pior caso – crescimento linear (O(N))
O pior caso acontece quando o valor está na última posição, obrigando o algoritmo a percorrer o vetor inteiro. Os resultados mostram isso bem:

- Tamanho 1.000: ~5.780 ns
- Tamanho 10.000: ~55.780 ns

O vetor aumentou 10 vezes e o tempo cresceu quase na mesma proporção (cerca de 9,6 vezes). Isso confirma a ideia de linearidade: quanto maior a entrada, maior o tempo.

2. Melhor caso – tempo constante (O(1))
Quando o valor está logo no começo, a busca encontra na primeira comparação. Por isso, o tempo praticamente não muda, independentemente do tamanho do vetor. Nos testes com 1.000 e 10.000 elementos, o tempo ficou sempre por volta de 100 ns. Esse é o comportamento típico do melhor caso.

3. Caso médio
No caso médio, espera-se que a busca percorra metade do vetor. Os resultados do vetor de tamanho 1.000 mostram isso direitinho:

- Último elemento (N): 5.780 ns
- Elemento do meio (N/2): 2.960 ns

O tempo do elemento do meio é praticamente metade do tempo do último (5.780 / 2 ≈ 2.890), o que confirma bem a expectativa teórica do caso médio.

Conclusão
A busca sequencial funciona sem problemas quando o vetor é pequeno. Porém, como o tempo cresce de forma linear, ela acaba ficando pouco eficiente em vetores maiores, especialmente quando comparada a algoritmos mais rápidos, como aqueles que têm desempenho logarítmico.

Busca Binária - Vetor							
Tamanho	Primeiro elemento	Último elemento	Elemento do meio	Aleatório 01	Aleatório 02	Aleatório 03	Inexistente
100	600 ns	220 ns	240 ns	240 ns	240 ns	240 ns	240 ns
1000	1400 ns	200 ns	180 ns	900 ns	200 ns	2700 ns	220 ns
10000	1260 ns	1300 ns	1240 ns	1240 ns	1220 ns	1240 ns	1620 ns

A Busca Binária segue a complexidade logarítmica O(logN). Diferente da busca sequencial, o tempo cresce bem devagar mesmo quando o tamanho do vetor aumenta bastante.

1. Escalabilidade (O(log N))
A estabilidade do algoritmo fica evidente quando comparamos os valores maiores. Mesmo aumentando o vetor de 100 para 10.000 elementos (100 vezes maior), o tempo médio de busca subiu só cerca de 6 vezes, de mais ou menos 200 ns para cerca de 1.250 ns. Isso faz sentido, já que o número de comparações também só dobra.

2. Comparação com a Busca Sequencial
Nos testes com vetor de 10.000 elementos, a diferença entre os dois algoritmos fica muito clara. Quando o valor não existe no vetor (pior caso), a Busca Binária levou cerca de 1.620 ns, enquanto a Sequencial precisou de algo próximo de 13720 ns.

3. Diferença entre posições testadas

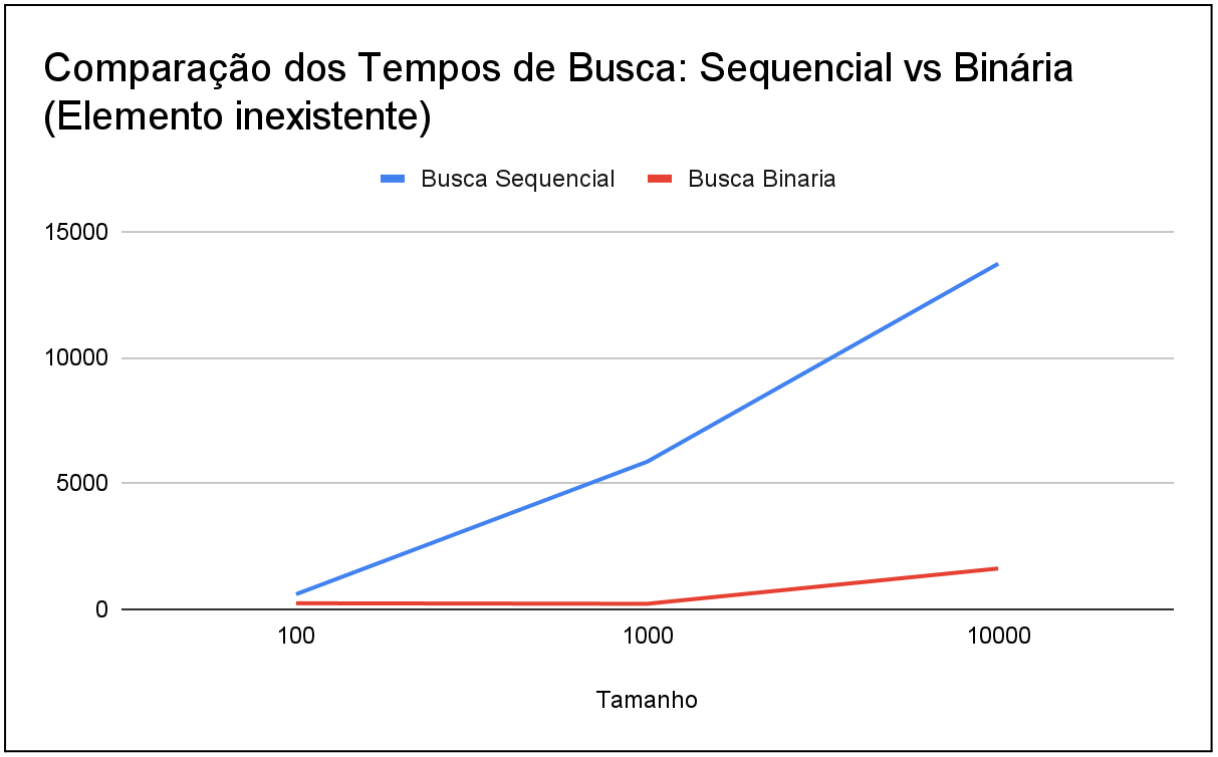
- **Elemento do meio:** Foi sempre o mais rápido (por exemplo, 180 ns com tamanho 1.000), já que representa o melhor caso da busca binária, o valor é encontrado logo na primeira comparação.

- **Primeiro e último elementos:** Ao contrário da busca sequencial, aqui as extremidades não são encontradas de imediato. A busca binária precisa

passar por várias divisões até chegar nelas, então os tempos acabam ficando próximos do caso médio/pior caso, cerca de 1.300 ns.

Conclusão

Os resultados mostram que a Busca Binária, comparada com a Busca Sequencial, é a melhor opção para vetores ordenados. Mesmo com vetores grandes, os tempos continuam baixos e estáveis, o que comprova a vantagem da sua escalabilidade em relação à busca sequencial. Em contrapartida, ela exige que o vetor esteja previamente ordenado.



Busca - Árvore Binária							
Tamanho	Primeiro elemento	Último elemento	Elemento do meio	Aleatório 01	Aleatório 02	Aleatório 03	Inexistente
100	120 ns	6480 ns	2800 ns	4940 ns	1880 ns	1260 ns	6100 ns
1000	100 ns	11140 ns	4800 ns	1300 ns	5960 ns	4900 ns	10380 ns
10000	120 ns	44560 ns	22100 ns	17300 ns	38160 ns	7960 ns	44600 ns

Os testes de busca na ABB foram feitos inserindo os dados em ordem crescente. Esse é justamente o pior cenário possível para esse tipo de árvore, porque, sem qualquer balanceamento, ela acaba virando praticamente uma lista encadeada inclinada para a direita.

1. Linearidade observada (N vs N/2)
Isso ficou bem evidente quando comparei o tempo para buscar o último elemento com o tempo para buscar o elemento do meio. Em uma estrutura degenerada, chegar até o meio deveria levar mais ou menos metade do tempo de chegar até o final, e foi exatamente o que aconteceu.
No teste com 10.000 elementos:

- Último elemento (N): 44.560 ns
- Elemento do meio (N/2): 22.100 ns

Isso mostra que a busca está acontecendo nó a nó, como em uma lista, com comportamento O(N).

2. Tempo de busca para elemento inexistente
Também busquei um valor maior que todos os presentes na árvore. Isso força o algoritmo a percorrer toda a estrutura até chegar a um ponteiro nulo.

- Tempo para elemento inexistente (10.000): 44.600 ns

Esse tempo é praticamente igual ao do último elemento, o que confirma que, nesse cenário, a árvore precisa visitar todos os nós para concluir que o valor não está presente.

3. Melhor caso (O(1))
A busca pelo primeiro elemento sempre foi rápida, entre 100 ns e 120 ns. Como os valores foram inseridos em ordem crescente, o primeiro elemento é justamente a raiz da árvore. Por isso, esse caso é sempre imediato, independentemente da profundidade que a árvore acaba ganhando para a direita.

Conclusão
Os resultados mostram que, sem algum tipo de balanceamento, a ABB se torna muito ineficiente quando recebe dados ordenados. Nessas condições, a busca cresce de forma linear com o tamanho da árvore, funcionando na prática como uma busca sequencial O(N) e perdendo totalmente o comportamento logarítmico O(logN) esperado de uma árvore binária bem estruturada.

Busca - Árvore AVL							
Tamanho	Primeiro elemento	Último elemento	Elemento do meio	Aleatório 01	Aleatório 02	Aleatório 03	Inexistente
100	1160 ns	520 ns	440 ns	520 ns	460 ns	380 ns	480 ns
1000	1000 ns	660 ns	680 ns	560 ns	1800 ns	640 ns	640 ns
10000	1260 ns	1040 ns	840 ns	2360 ns	720 ns	860 ns	1000 ns

Os testes feitos na AVL mostraram bem como o balanceamento automático faz diferença. Mesmo inserindo os dados em ordem crescente, o mesmo cenário que bagunçou totalmente a ABB, a AVL continuou organizada e manteve um desempenho estável.

1. Estabilidade e comportamento O(logN)
O que mais chama atenção nos resultados é que os tempos de busca ficaram muito próximos uns dos outros. No teste com 10.000 elementos, por exemplo:

- Elemento inexistente: 1.000 ns
- Último elemento: 1.040 ns
- Primeiro elemento: 1.260 ns

Todos ficaram na casa de 1 microsegundo. Isso mostra que a altura da árvore está realmente controlada, dá por volta de 14 níveis. Ou seja, não existe caminho muito longo nem muito curto tudo fica equilibrado.

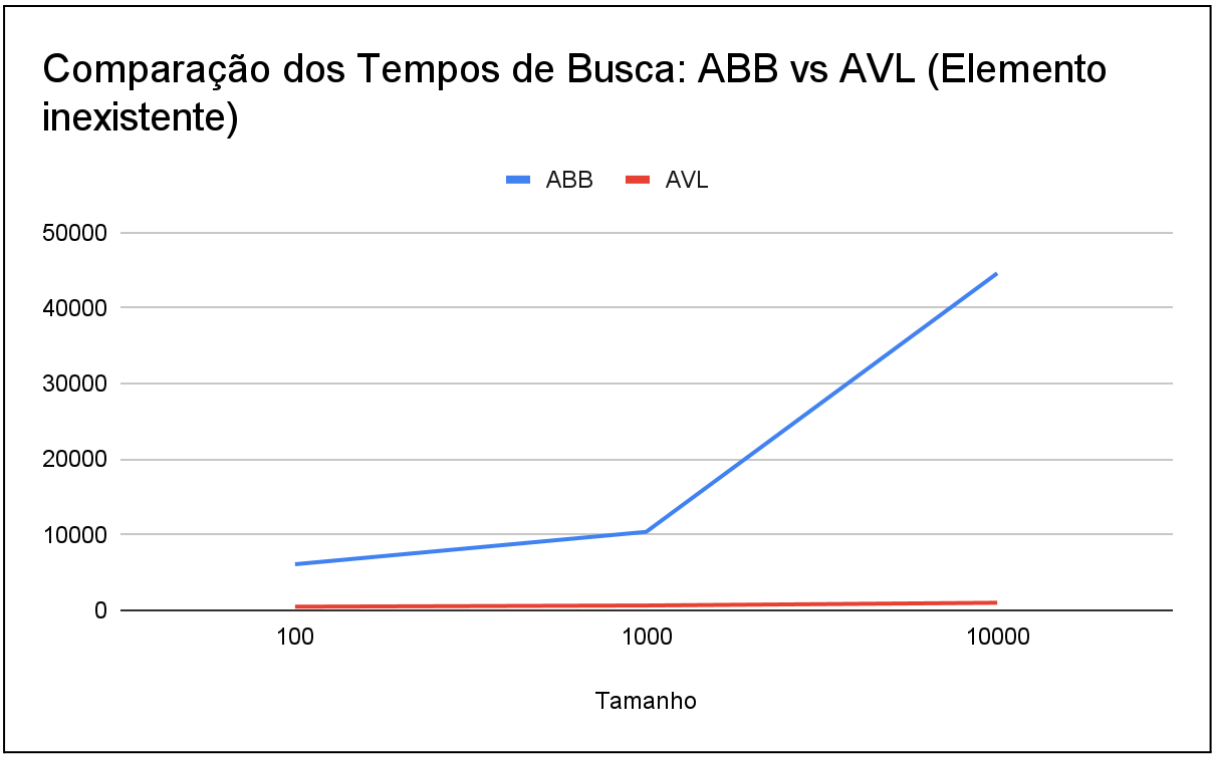
2. Escalabilidade
Comparando os tamanhos dos vetores, o comportamento logarítmico ficou bem evidente (Ultimo elemento):

- Tamanho 100 → tempos perto de 500 ns
- Tamanho 10.000 → tempos perto de 1.000 ns

Mesmo aumentando a entrada em 100 vezes, o tempo de busca só dobrou. É o oposto do que acontece numa busca linear ou numa ABB degenerada, onde o tempo cresceria na mesma proporção do tamanho da entrada.

3. Sobre o tempo do “Primeiro Elemento”
Um ponto interessante é que buscar o primeiro elemento (1.260 ns) foi um pouco mais lento que buscar o elemento do meio (840 ns). Isso acontece porque, com as rotações de balanceamento, o menor valor acaba sendo empurrado para baixo e vira uma folha na subárvore esquerda. Já o elemento do meio tende a ficar mais próximo da raiz depois das rotações. Isso confirma que a AVL está se reestruturando conforme a inserção, diferente da ABB, onde o primeiro elemento ficou preso na raiz.

Conclusão
Os resultados deixam claro que a AVL evita completamente o pior caso causado por dados ordenados, se o objetivo é garantir um desempenho consistente independentemente da ordem de inserção, a AVL entre as duas é a melhor escolha.



Bubble Sort			
Tamanho	Crescente(ns)	Decrescente(ns)	Aleatório(ns)
100	24060 ns	44720 ns	2520 ns
1000	108420 ns	256440 ns	448480 ns
10000	5899380 ns	18164920 ns	38231280 ns

O Bubble Sort realmente tem comportamento quadrático O(n²). Conforme o tamanho do vetor aumenta, o tempo de execução cresce de forma muito rápida, o que bate com a teoria. Além disso, alguns resultados chamaram atenção por causa da forma como o processador lida com as comparações e trocas.

1. Crescimento Geral O(n²)

Comparar 1.000 com 10.000 elementos mostra bem como o Bubble Sort não escala.
No cenário Aleatório, o tempo foi de cerca de 0,45 ms para mais de 38 ms.
Isso significa um aumento de aproximadamente 85 vezes, mesmo com a entrada sendo apenas 10 vezes maior, algo bem próximo do comportamento quadrático esperado ($10^2 = 100$).

2. Análise dos Cenários

Crescente (melhor caso "prático")
Esse foi o mais rápido: ~5,9 ms para 10.000 elementos.
Como já estava ordenado, o algoritmo não precisou fazer nenhuma troca, o que reduz muito o custo de memória. Mesmo assim, o tempo não foi linear, porque minha implementação ainda percorre todos os laços aninhados, só que sem trocar nada.

Decrescente vs. Aleatório
Pela teoria, o vetor Decrescente deveria ser o pior caso, porque exige o maior número de trocas. Mas, nos testes, o Aleatório (38 ms) ficou bem mais lento que o Decrescente (18 ms).

Creio eu que a causa é a predição de desvio (branch prediction) do processador:

- **No Decrescente:** a condição if (v[j] > v[j+1]) é sempre verdadeira. Como o padrão é constante, o processador acerta quase sempre a predição e consegue manter o pipeline cheio, executando tudo mais rápido.

- **No Aleatório:** a comparação é praticamente imprevisível (metade das vezes dá verdadeiro, metade falso). Isso gera muitos erros de predição, obrigando o processador a limpar o pipeline várias vezes, um custo maior que o próprio custo das trocas no caso decrescente.

Conclusão

Os resultados mostram que o Bubble Sort não é uma boa opção quando o volume de dados cresce. Mesmo no melhor cenário (Crescente), 10.000 elementos levaram quase 6 ms, enquanto no Aleatório esse valor subiu para 38 ms.

O ponto mais interessante é perceber que, em hardware moderno, nem sempre o número de operações de troca é o fator mais pesado. A previsibilidade das comparações pode impactar o desempenho tanto quanto, ou até mais do que escrever valores na memória.

Quick Sort			
Tamanho	Crescente(ns)	Decrescente(ns)	Aleatório(ns)
100	1180 ns	2380 ns	840 ns
1000	68060 ns	157520 ns	15980 ns
10000	5566500 ns	14597760 ns	431240 ns

Os testes que fiz mostram claramente o ponto forte e o ponto fraco do QuickSort. Quando os dados estão em uma ordem “normal”, ele é extremamente rápido. Mas, dependendo da forma como o pivô é escolhido, ele pode degradar bastante, e na minha implementação o pivô é sempre o último elemento, o que influencia diretamente nos resultados.

1. Caso Aleatório - desempenho ideal O(NlogN)

No cenário aleatório, o QuickSort se saiu muito bem.
Para ordenar 10.000 elementos, o tempo ficou por volta de **0,43 ms**, o que é bem baixo.

O Bubble Sort que testei antes levou cerca de **38 ms** para o mesmo tamanho, então o QuickSort foi mais ou menos **88 vezes mais rápido**.

Esse bom desempenho acontece porque, com dados aleatórios, o último elemento (que é o pivô) costuma cair numa posição mais “central”, dividindo o vetor em partes mais equilibradas. Isso mantém a profundidade da recursão pequena e o algoritmo trabalha dentro do padrão.

2. Pior Caso - quando o vetor já está ordenado O(n²)

Nos cenários Crescente e Decrescente o desempenho caiu bastante.
Isso acontece porque escolher sempre o último elemento como pivô não funciona bem quando o vetor já vem ordenado. Nessas situações, o pivô acaba sendo sempre o maior ou o menor valor, e o algoritmo não consegue dividir o problema em partes menores de forma eficiente.

O resultado é que o QuickSort deixa de dividir o vetor em duas partes, separa só o pivô, e passa quase o vetor inteiro para a próxima chamada recursiva. Isso faz o algoritmo descer para o comportamento quadrático, igual ao de algoritmos bem mais lentos.

3. Prova prática dessa queda de desempenho

Os números mostram isso claramente:

- Para **1.000 elementos** no cenário decrescente: ~0,15 ms
- Para **10.000 elementos**: ~14,6 ms

Ou seja, aumentando o tamanho do vetor em 10 vezes, o tempo aumentou cerca de **93 vezes**, que é praticamente o padrão de um algoritmo **O(n²)**

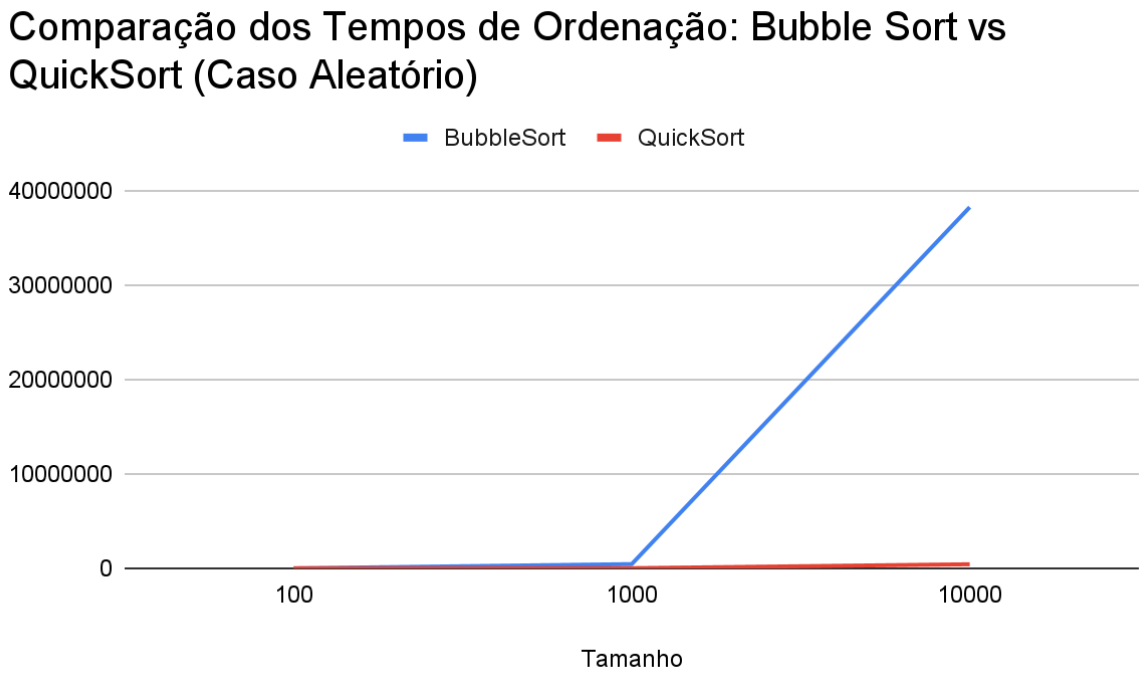
4. Comparação direta (para 10.000 elementos)

- **Aleatório:** 0,43 ms
- **Crescente:** 5,56 ms (cerca de 12 vezes mais lento que o aleatório)
- **Decrescente:** 14,6 ms (aproximadamente 34 vezes mais lento)

O pior cenário foi o Decrescente. Além da recursão profunda, ele ainda faz o máximo possível de movimentações para reposicionar elementos em relação ao pivô.

Conclusão

Os testes deixam bem claro que o QuickSort é excelente para dados gerais, mas, com um pivô fixo no final, ele fica muito vulnerável quando o vetor já está ordenado. Se o algoritmo for usado em situações onde isso pode acontecer, o ideal seria trocar a estratégia de pivô, por exemplo, usando **mediana de três** ou pivô aleatório para evitar cair no pior caso e manter o desempenho próximo de O(NlogN).



Considerações Finais

A realização deste trabalho permitiu comparar, na prática, o comportamento de várias estruturas de dados e algoritmos de ordenação em Java. Os testes ajudaram a confirmar, com resultados reais, aquilo que normalmente vemos apenas na teoria sobre complexidade.

ABB vs. AVL

Os testes mostraram claramente o quanto a ABB é sensível à ordem dos dados. Quando os valores foram inseridos em ordem crescente, a ABB praticamente virou uma lista encadeada, fazendo com que os tempos de inserção e busca ficassem muito altos.

Já a AVL lidou muito bem com esse mesmo cenário. Por causa do balanceamento automático, ela manteve a altura pequena e os tempos de execução estáveis, independentemente da ordem de entrada. Na prática, ficou nítido que a AVL é uma opção bem mais segura quando se quer garantir desempenho consistente.

Algoritmos de Busca

A comparação entre busca sequencial e busca binária reforçou o que já era esperado: a sequencial cresce de forma linear e rapidamente se torna lenta conforme o vetor aumenta, enquanto a busca binária manteve tempos bem baixos. A única exigência foi vetor estar ordenado e é um custo pequeno perto do ganho de desempenho.

Bubble Sort e o impacto do hardware

Os testes com Bubble Sort chamaram atenção por um detalhe curioso: o vetor aleatório foi mais lento que o vetor em ordem decrescente, mesmo o decrescente sendo o “pior caso clássico”. Foi interessante perceber como o hardware também influencia os resultados.

QuickSort e a escolha do pivô

O QuickSort teve ótimo desempenho com dados aleatórios, confirmando seu caso médio O(NlogN). Porém, usando o último elemento como pivô, o algoritmo quebrou completamente nos casos crescente e decrescente, chegando ao comportamento quadrático. Isso deixou claro que, em implementações reais, é obrigatório usar estratégias melhores, como pivô aleatório ou mediana de três, para evitar justamente esse tipo de problema.

Conclusão Final

Em geral, o trabalho mostrou que a escolha da estrutura ou do algoritmo ideal depende tanto da natureza dos dados quanto da própria forma como o hardware executa as operações. A teoria ajuda a entender o comportamento esperado, mas só os testes práticos revelam como tudo realmente se comporta na máquina.