# Relatório

## Tempo de execução de algoritmos

# Descrição do exercício

a) Implementar um código que insere um total de P pessoas com id crescente, começando em 1, em uma
estrutura ArrayList do Java. Após a inserção, este código deve ser capaz de realizar N buscas por pessoas
com id’s aleatórios na ArrayList.
b) Implementar um código que insere um total de P pessoas com id crescente, começando em 1, em uma
estrutura HashMap do Java. Após a inserção, este código deve ser capaz de realizar N buscas por pessoas
com id’s aleatórios no HashMap.
c) Para cada implementação acima, executar testes com os valores de P e N abaixo:
i. P = 2.500.000, N = 20.000
ii. P = 5.000.000, N = 20.000
iii. P = 10.000.000, N = 20.000
d) Para cada de (c), marcar o tempo de execução incluindo a criação dos dados e o loop de busca.
e) Criar uma tabela comparando os resultados de tempo x tamanho para os resultados obtidos.
f) Analisar criticamente esta tabela, relacionado os resultados obtidos com as ordens de complexidade
teóricas conhecidas das estruturas de dados.
g) Para as implementações de (a) e (b), executar um teste com P = 2.500.000 e N = 40.000. Marcar o tempo
de execução destes testes.
h) Analisar criticamente esta último resultado, debatendo sobre as operações de criação dos dados e
busca dos dados nas estruturas utilizadas.

# Especificações do computador

**Processador:** AMD Ryzen 5 1400 3400 MHz

**Memória:** 8GB DDR4 1200mhz

**Sistema Operacional:** Windows 10

# Resultado ArrayList

| P | N | Tempo de execução |
|---|---|---|
| 2.500.000 | 20.000 | 173,5385 segundos<br>(2 minutos) |
| 5.000.000 | 20.000 | 362,7404 segundos<br>(6 minutos) |
| 10.000.000 | 20.000 | 738,6482 segundos<br>(12 minutos) |
| 2.500.000 | 40.000 | 379,3654 segundos<br>(6 minutos) |

# Análise dos resultados

O código implementado cria uma ArrayList de pessoas e preenche-a com P pessoas. Em seguida, ele executa uma série de buscas aleatórias na lista com base em N, onde N é o número de buscas a serem realizadas e P é o tamanho da lista.

O loop externo que adiciona P pessoas à lista tem complexidade O(P), pois ele itera P vezes, e cada iteração é uma operação de tempo constante para adicionar uma pessoa à lista.

O loop interno, que realiza as buscas na lista, tem complexidade O(N * P), onde N é o número de buscas e P é o tamanho da lista. Isso ocorre porque, em cada uma das N iterações, o código percorre a lista de pessoas (que tem tamanho P) para verificar se a pessoa com o ID correspondente existe. Portanto, para cada busca, o tempo necessário é proporcional ao tamanho da lista (P).

Para P = 2.500.000 e N = 20.000, o tempo de execução é de aproximadamente 173,5385 segundos (ou 2 minutos). Isso parece razoável, considerando a complexidade do código. O tempo de execução aumenta de acordo com a quantidade de buscas (N) e o tamanho da lista (P).

À medida que tanto P quanto N dobram, o tempo de execução mais do que duplica. Isso está alinhado com a complexidade O(N * P) do loop de busca.

Quando P é mantido constante (2.500.000) e N é dobrado (40.000), o tempo de execução também aumenta em aproximadamente o dobro. Isso confirma a relação linear entre o tempo de execução e o número de buscas (N).

ArrayList tem uma complexidade de tempo O(1) para adicionar elementos no final, mas acessar um elemento específico tem uma complexidade de tempo O(n), pois requer percorrer a lista até o índice desejado. Portanto, embora a inserção na lista (o primeiro loop) seja razoável, as operações de busca (o segundo loop) são mais lentas, especialmente para grandes conjuntos de dados.

O uso de um ArrayList pode não ser a melhor escolha, especialmente para operações de busca eficientes. Embora seja eficiente para inserção no final (o primeiro loop), a busca em uma lista requer percorrer os elementos, o que resulta em uma complexidade linear. Para cenários onde as buscas são frequentes, outras estruturas de dados como HashMap poderiam oferecer desempenho melhor devido à busca mais rápida.

A análise dos resultados confirma a complexidade temporal do código fornecido. A relação direta entre o aumento de P e N e o tempo de execução ilustra a natureza quadrática da complexidade (O(N * P)). Isso ressalta a importância de considerar a complexidade algorítmica ao projetar e implementar sistemas, especialmente para conjuntos de dados volumosos.

Os resultados também destacam questões de escalabilidade. À medida que tanto o tamanho da lista quanto o número de buscas aumentam, o tempo de execução aumenta significativamente. Isso pode se tornar um problema em sistemas que lidam com grandes quantidades de dados ou onde a resposta rápida é essencial.

# Resultado HashMap

| P | N | Tempo de execução |
|---|---|---|
| 2.500.000 | 20.000 | 0,6854 segundos |
| 5.000.000 | 20.000 | 1,1809 segundos |
| 10.000.000 | 20.000 | 2,2205 segundos |
| 2.500.000 | 40.000 | 0,6436 segundos |

# Análise dos resultados

Tanto a inserção quanto a recuperação em um HashMap têm uma complexidade média de O(1), ou seja, são operações de tempo constante, desde que não haja colisões excessivas.

Os resultados observados nas execuções do teste refletem essa característica em geral. O aumento do tamanho do mapa (P) tem um impacto significativo no tempo de execução, mas não de forma linear, o que está de acordo com a complexidade O(1) esperada para inserção e recuperação.

Ao dobrar o tamanho do mapa de 2.500.000 para 5.000.000 enquanto mantém o número de acessos (N) constante em 20.000, observamos que o tempo de execução quase dobra, indo de 0,6854 segundos para 1,1809 segundos. Isso está de acordo com a expectativa teórica devido à complexidade O(1).

Da mesma forma, ao dobrar novamente o tamanho do mapa de 5.000.000 para 10.000.000 mantendo N constante, vemos que o tempo de execução mais do que dobra, indo de 1,1809 segundos para 2,2205 segundos. Isso também está em conformidade com a complexidade O(1) esperada.

A mudança de N de 20.000 para 40.000, mantendo P constante em 2.500.000, resultou em uma diminuição ligeira e inesperada no tempo de execução. Isso pode ser atribuído a fatores como otimizações do Java JIT Compiler, a maneira como o gerador de números aleatórios funciona ou outras características específicas da implementação do HashMap.

Os resultados sugerem que o algoritmo implementado é eficiente para operações de inserção e recuperação em HashMap. O tempo de execução aumenta de acordo com o número de consultas e o tamanho do mapa, mas permanece razoavelmente baixo, indicando que a estrutura de dados está lidando bem com as operações realizadas.

O fato de as consultas serem aleatórias pode influenciar a eficiência, especialmente em casos onde a distribuição das chaves pode levar a colisões. No entanto, como o HashMap lida com colisões, o impacto parece ser minimizado, mas pode contribuir para variações no tempo de execução.

# Conclusão

A comparação entre o desempenho do método get() do HashMap e o ArrayList revela uma clara vantagem do HashMap. Os resultados evidenciam que, à medida que o tamanho dos conjuntos de dados aumenta, o tempo de execução das operações de busca no HashMap cresce de forma mais estável e previsível em comparação com o ArrayList.

Esta estabilidade no desempenho do HashMap, mesmo com grandes volumes de dados, indica uma melhor escalabilidade em comparação com o ArrayList. Isso é crucial em cenários onde há necessidade de lidar com grandes quantidades de dados e realizar buscas frequentes.

Em síntese, os resultados reforçam que o HashMap é a escolha mais eficiente para operações de busca, oferecendo acesso rápido aos elementos por meio de uma chave e demonstrando uma escalabilidade superior, especialmente em ambientes com grandes conjuntos de dados.