Tempo de execucao de algoritmos - Analise dos resultados

Nome: Guilherme Lage da Costa

Matrícula: 792939

Data: 19 de fevereiro de 2024

Especificações do ambiente de execução

- JDK: Java 17
- Processador: AMD Ryzen 5 3600, 4.2 Ghz, 6 cores e 12 threads, 32mb de cache
- RAM: 16GB, 3000Ghz
- Sistema Operacional: Windows 11
- IDE: IntelliJ Ultimate

Resultados obtidos

Estrutura armazenamento	Quantidade pessoas	Quantidade buscas	Tempo percorrido (Seg)
ARRAY_LIST			
Cenário 01	2.500.000,00	20.000,00	0,555193225
Cenário 02	5.000.000,00	20.000,00	0,978161675
Cenário 03	10.000.000,00	20.000,00	1,86601815
Cenário 04	2.500.000,00	40.000,00	0,542301925
		Média	0,985418744
HASH_MAP			
Cenário 01	2.500.000,00	20.000,00	0,6314886
Cenário 02	5.000.000,00	20.000,00	1,141933075
Cenário 03	10.000.000,00	20.000,00	2,43808885
Cenário 04	2.500.000,00	40.000,00	0,570067
		Média	1,195394381

Base de testes utilizada

Estrutura armazenamento	Quantidade pessoas	Quantidade buscas	Tempo percorrido (Seg)	Tempo percorrido (Ms)
ARRAY_LIST	2.500.000,00	20.000,00	0,57418	574,18410
ARRAY_LIST	5.000.000,00	20.000,00	1,03762	1037,62380
ARRAY_LIST	10.000.000,00	20.000,00	1,84493	1844,93090
ARRAY_LIST	2.500.000,00	40.000,00	0,55190	551,89660
HASH_MAP	2.500.000,00	20.000,00	0,60075	600,74770
HASH_MAP	5.000.000,00	20.000,00	1,04979	1049,78900
HASH_MAP	10.000.000,00	20.000,00	2,38048	2380,48160
HASH_MAP	2.500.000,00	40.000,00	0,48576	485,76420
ARRAY_LIST	2.500.000,00	20.000,00	0,54069	540,69130
ARRAY_LIST	5.000.000,00	20.000,00	0,98640	986,40490
ARRAY_LIST	10.000.000,00	20.000,00	1,89096	1890,96130
ARRAY_LIST	2.500.000,00	40.000,00	0,54567	545,67120
HASH_MAP	2.500.000,00	20.000,00	0,60401	604,01410
HASH_MAP	5.000.000,00	20.000,00	1,27466	1274,66070
HASH_MAP	10.000.000,00	20.000,00	2,49918	2499,18490
HASH_MAP	2.500.000,00	40.000,00	0,62324	623,24470
ARRAY_LIST	2.500.000,00	20.000,00	0,55314	553,13920
ARRAY_LIST	5.000.000,00	20.000,00	0,97222	972,22040
ARRAY_LIST	10.000.000,00	20.000,00	1,77380	1773,79600
ARRAY_LIST	2.500.000,00	40.000,00	0,53285	532,84670
HASH_MAP	2.500.000,00	20.000,00	0,69926	699,26170
HASH_MAP	5.000.000,00	20.000,00	1,03090	1030,89870
HASH_MAP	10.000.000,00	20.000,00	2,39474	2394,74030
HASH_MAP	2.500.000,00	40.000,00	0,60083	600,82600
ARRAY_LIST	2.500.000,00	20.000,00	0,55276	552,75830
ARRAY_LIST	5.000.000,00	20.000,00	0,91640	916,39760
ARRAY_LIST	10.000.000,00	20.000,00	1,95438	1954,38440
ARRAY_LIST	2.500.000,00	40.000,00	0,53879	538,79320
HASH_MAP	2.500.000,00	20.000,00	0,62193	621,93090
HASH_MAP	5.000.000,00	20.000,00	1,21238	1212,38390
HASH_MAP	10.000.000,00	20.000,00	2,47795	2477,94860
HASH_MAP	2.500.000,00	40.000,00	0,57043	570,43310

Análise dos resultados obtidos

De modo geral, o tempos de execução para realizar as operações descritas no exercício foram semelhantes em ambos os casos (Array List e HashMap). De toda forma, comparando os valores médios finais de cada uma das estruturas, foi possível observar que as operações com Array List apresentaram um tempo de execução médio (0,9854seg) 21% inferior às realizadas com HashMap (1,1953 seg).

A seguir, será feito uma análise mais detalhada acerca dos resultados obtidos.

Complexidade da estrutura Array List

```
private static void realizarTestesArrayList() {

        List<Pessoa> listaPessoas = new ArrayList<>();

    cenarios.forEach((integer, cenario) -> {

            int qtdePessoas = cenario.qtdePessoas();
        int qtdeBuscas = cenario.qtdeBuscas();

        for (int i = 1; i <= qtdePessoas; i++)
                listaPessoas.add(new Pessoa(i, format("Pessoa
%s")));

                for (int i = 1; i <= qtdeBuscas; i++)
                listaPessoas.get(random.nextInt((qtdePessoas)));
        }
}
```

Observação: apenas para fins de análise, foram removidas no trecho de código acima algumas estruturas adicionas do Java, como tratamento de exceção e geração de logs.

Para analisar essa estrutura, vou analisar os dois métodos que foram empregados no código, sendo eles:

- add() → adicionar um novo elemento à lista
- get() → retorna um elemento da lista

Como estamos falando de uma lista, ou seja, um elemento é posicionado após o outro, o método de `add()` depende diretamente da posição de um dado elemento da lista. Ou seja, ele precisa saber qual a última posição na qual existe um elemento, para adicionar um outro na próxima posição. Por conta disso, a complexidade desse método é da ordem `O(n)`.

O método `get()`, por sua vez, recebe como parâmetro o index (posição) do elemento que está sendo buscado. Por conta disso, esse método não depende da posição de outros elementos da lista, e consequentemente, possui complexidade da ordem `O(1)`.

Complexidade da estrutura HashMap

```
private static void realizarTestesHashMap() {

        HashMap<Integer, Pessoa> hashMapPessoas = new HashMap<>();

    cenarios.forEach((integer, cenario) -> {

            int qtdePessoas = cenario.qtdePessoas();
        int qtdeBuscas = cenario.qtdeBuscas();

        for (int i = 1; i <= qtdePessoas; i++)
                        hashMapPessoas.put(i, new Pessoa(i,
format("Pessoa %s", i)));

                for (int i = 1; i <= qtdeBuscas; i++)

hashMapPessoas.get(random.nextInt((qtdePessoas)));
        }
}
```

Observação: apenas para fins de análise, foram removidas no trecho de código acima algumas estruturas adicionas do Java, como tratamento de

exceção e geração de logs.

Para analisar essa estrutura, vou analisar os dois métodos que foram empregados no código, sendo eles:

- put() → adicionar um novo elemento ao mapa
- get() → retorna um elemento do mapa

Diferente do método de lista, analisado anteriormente, o método de `put()`, que tem função semelhante ao `add()`, mas para estruturas em mapas, não necessita de referências de outras posições para saber onde será inserido no mapa. Isso faz com que a operação de inserção seja da ordem de `O(1)`, o que é mais rápido em relação às listas.

Um adendo com relação a essa análise, é que em colisões durante a inserção de um elemento, é possível que existam colisões entre os elementos, ou seja, tentar inserir um elemento em uma posição que já está ocupada. Nesses casos, o método irá buscar a próxima posição vaga para que a inserção seja feita, chegando em um cenário semelhante ao `add()` da lista, de complexidade `O(n)`. Segundo informações do Baeldung , nas novas versões do JDK, esse tempo de complexidade foi reduzido para `O(log(n))`, em razão de otimizações realizadas.

De toda forma, como a classe `Pessoa` implementada no código possui implementação do método `equals()`, é improvável que tenham ocorrido muitas colisões durante as inserções no mapa.

O método de `get()` para a estrutura de mapa recebe como parâmetro um `Object key`, ou seja, uma chave que referencia um item único no mapa. Por conta disso, a busca é feita diretamente no index de um determinado elemento, de modo que possui complexidade da ordem `O(1)`.

Conclusões

Com base nas análises realizadas, eu esperaria que os resultados obtidos mostrassem que as operações que utilizam estrutura de HashMap tiveram resultados superiores ao ArrayList, o que não aconteceu. Na prática, como

demonstrado no primeiro tópico, a estrutura de lista apresentou na média tempo de exeução 21% superior ao mapa.

Inicialmente, avaliei se a estrutura de mapa poderia estar consumindo mais recursos de computador, no entanto, os resultados também foram semelhantes. Com a estrutura de lista, o consumo máximo de CPU e RAM, respectivamente, ficaram em torno de 38 e 76%, enquanto com a estrutura de mapa ficou em torno de 78 e 80%, o que eu não acredito que poderia gerar toda essa diferença na performance, haja vista que esses valores ainda não causarão redução da performance.

Um comportamento que percebi, é que a implementação para salvar o mapa na linha: `hashMapPessoas.put(i, new Pessoa(i, format("Pessoa %s", i)))`, o fato de colocar a chave a partir da variável `i`, causa perda de performance considerável. Em alguns testes que realizei à parte, trocando `i` por um valor constante igual a `ZERO`, percebi ganho de performance próxima a 20%, o que explicaria a diferença entre as duas estruturas nos testes.

Se apenas essa mudança causou a disparidade dos valores esperados e os obtidos, acredito que a estrutura de mapa apresentaria uma vantagem maior em cenários em que fossem realizadas mais operações de inserção e busca, uma vez que as implementações dessas rotinas são da ordem de `O(1)` na maioria dos casos.