RELATÓRIO DO PROJETO

ALUNO: Erick Rafael Ferreira Nunes

- **“Você deve produzir um relatório explicando as escolhas feitas durante o desenvolvimento, incluindo a definição do tamanho m da tabela hash, e uma análise de desempenho utilizando o método System.nanoTime() para medir o tempo de execução das operações (inserção de um novo número e busca por um número). Além disso, realize comentários sobre a complexidade assintótica das operações, utilizando a notação big-O, com base no entendimento de como Java implementa internamente o ArrayList, o LinkedList e o TreeSet.”**

➢ Explicação inicial:

Durante o Desenvolvimento do projeto descrito, utilizei apenas 2 classes Java para o executar, a Main e TabelaHash. A classe Main, como de praxe, ficou responsável pela execução do que foi solicitado. Já a classe TabelaHash foi implementada com base nas características das 3 estruturas de dados solicitadas: ArrayList, LinkedList e TreeSet, seguindo a utilização da 4° estrutura que intitulou a classe, a TabelaHash.

Nesse sentido, optei por usar apenas 1 classe para a implementação, dessa forma, quando cada objeto tabela foi criado, o que diferiu uma das outras foi o construtor utilizado já que a criação de cada tinha parâmetros específicos a serem seguidos.

A partir disso, cada uma das 3 estruturas em TabelaHash possuía 1 atributo do formato lista para si (para cumprir o sentido de “Tabela”), 1 método Insert e 1 método Search que foram solicitados.

O tamanho “m” utilizado para todas as tabelas foi definida com base na característica da própria tabela, pois quando um elemento “k” está para ser colocado

na tabela, se utiliza a notação: h(k) = k % m (resto da divisão do elemento por o tamanho da tabela). Dessa maneira, é indicado que um número escolhido para ser o "m" seja um número primo, pois assim ele é um número maior que 1, que só pode ser divisível sem sobra por 1 e por ele mesmo. Seguindo esse pensamento, escolhi o número primo 211, pois estatisticamente, ele garantia cerca de 2.370 elementos em cada "linha" da tabela, deixando assim ela equilibrada.

Portanto, após a parte lógica ficar pronta, realizei a situação solicitada e seus respectivos testes. Desse modo, criei uma cartela de Bingo de 500.000 números aleatórios e sem repetição, e realizei a inserção da mesma sequência de valores em cada uma das 3 tabelas hash. Logo após isso, seguindo nessa mesma lógica, criei uma lista de 2.000.000 de números, também aleatórios e sem repetição, e realizei procuras dessa mesma sequência de valores em cada uma das 3 tabelas.

Os resultados em tempo decorrido (segundos) estão a seguir:

➔ INSERT

 # TABELA HASH - INSERT ARRAYLIST

-> Tempo total em segundos: 0,0343405000

-> Tempo médio em segundos: 0,0000000687

 # TABELA HASH - INSERT LINKEDLIST

-> Tempo total em segundos: 0,0869709000

-> Tempo médio em segundos: 0,0000001739

 # TABELA HASH - INSERT TREESET

-> Tempo total em segundos: 0,3422820000

-> Tempo médio em segundos: 0,0000006846

➔ SEARCH

 # TABELA HASH - SEARCH ARRAYLIST

-> Tempo total em segundos: 13,1495761001

-> Tempo médio em segundos: 0,0000065748

 # TABELA HASH - SEARCH LINKEDLIST

-> Tempo total em segundos: 36,6112060000

-> Tempo médio em segundos: 0,0000183056

 # TABELA HASH - SEARCH TREESET

-> Tempo total em segundos: 1,6527774000

-> Tempo médio em segundos: 0,0000008264

➢ Comentário sobre os custos de Insert e Search

- **Nas estruturas ArrayList e LinkedList utilizei o método .add para inserir os números, em ambas as estruturas esse método tem custo O(1), pois elas guardam a referência do final da lista e colocam o elemento lá. Porém é interessante reparar em uma diferença mínima de tempo entre as duas execuções:**

  # TABELA HASH - INSERT ARRAYLIST

  -> Tempo médio em segundos: 0,0000000687

  # TABELA HASH - INSERT LINKEDLIST

  -> Tempo médio em segundos: 0,0000001739

  Essa diferença se dá pela forma de registro na inserção, onde na ArrayList é inserido direto na posição, mas na LinkedList acontece uma troca, que é a forma de implementação da lista encadeada, que justifica essa diferença mínima.

- **Já na estrutura TreeSet, onde também utilizei o método .add para a inserção de elementos, tem custo médio O(log n), pois leva em consideração a estrutura árvore binária de busca, porém, no pior caso, a inserção pode ter custo O(n), por causa da necessidade de balanceamento da árvore. Esses custos, médio e pior, justifica essa estrutura ser a mais lenta em termo de inserção:**
  # TABELA HASH - INSERT TREESET

  -> Tempo médio em segundos: 0,0000006846

- **Indo agora para os método utilizados para realizar o Search, nas estruturas ArrayList e LinkedList, utilizei o método .indexOf, que retorna a posição do elemento procurado na lista, e se não encontrar retorna -1. Esse método tem custo O(n), pois procura na lista inteira para verificar se tem ou não o número.**
  # TABELA HASH - SEARCH ARRAYLIST

  -> Tempo médio em segundos: 0,0000065748

  # TABELA HASH - SEARCH LINKEDLIST

  -> Tempo médio em segundos: 0,0000183056

- **Já para procurar na estrutura TreeSet, utilizei o método .contains, que retorna um boleano true se tiver o elemento na árvore, e false se não. Como dito anteriormente, por ser uma BST (Árvore Binária de Busca), o custo dessa procura é O(log n), e por ser uma árvore balanceada durante a inserção, não ocorre o pior caso de O(n).**
  Dessa forma, fica claro o quanto a TreeSet é superior as outras estruturas utilizadas no quesito busca:
  # TABELA HASH - SEARCH TREESET

  -> Tempo médio em segundos: 0,0000008264