Indexação e Busca Eficiente com HashMap e TreeMap

Objetivo

Nesta atividade, você irá explorar **estruturas de indexação de conteúdo** utilizando o `HashMap` e o `TreeMap` da **Java Collections Framework (JCF)**.

Ao final do tutorial, você será capaz de:

- Entender como o Java utiliza funções de *hash* e ordenação natural para armazenar e buscar informações.
- Medir diferenças de performance entre `HashMap` e `TreeMap`.
- Implementar uma indexação simples de texto, como fazem mecanismos de busca.
- Utilizar ferramentas de **debug** do IntelliJ IDEA para observar o funcionamento interno das estruturas.

Conceitos Envolvidos

- Estruturas de indexação: `HashMap` (baseada em *hashing*) e `TreeMap` (baseada em árvores).
- Armazenamento de pares **chave-valor** (`Map<K, V>`).
- Função `getOrDefault()` e o laço `for-each` com `Map.Entry`.
- Análise de tempo e depuração.

Materiais Necessários

- **IntelliJ IDEA** (Community Edition ou superior).
- **JDK 17** (ou mais recente).
- Um **arquivo de texto (.txt)** para ser indexado — pode conter qualquer parágrafo ou texto de sua escolha.

Passo a Passo

Etapa 1 – Crie o projeto

1. Abra o **IntelliJ IDEA**.
2. Crie um novo projeto Java e nomeie como `Atividade_Indexacao`.
3. Crie um novo arquivo chamado **`WordIndexer.java`** dentro da pasta `src`.
4. Na raiz do projeto (mesmo diretório de `src`), crie um arquivo chamado **`texto.txt`** com qualquer conteúdo (ex: uma pequena notícia ou trecho de livro).

Exemplo de conteúdo do arquivo:

```
A inteligência artificial está transformando o mundo.
A análise de dados é parte fundamental dessa revolução tecnológica.
```

Etapa 2 – Copie o código abaixo

Cole o código no arquivo `WordIndexer.java`.

```
import java.util.*;
import java.io.*;

/**
 * Exemplo prático de indexação de conteúdo usando HashMap e
TreeMap.
 * O programa lê um texto, separa as palavras e conta a frequência
de cada uma.
 */
public class WordIndexer {

    public static void main(String[] args) {
        // Caminho do arquivo de texto que será indexado
        String filePath = "texto.txt";  // Altere se necessário

        // Estruturas de indexação
        Map<String, Integer> hashIndex = new HashMap<>();
        Map<String, Integer> treeIndex = new TreeMap<>();

        System.out.println("=== INDEXAÇÃO DE CONTEÚDO ===\n");

        // Indexação e medição de tempo usando HashMap
        long start = System.nanoTime();
        indexFile(filePath, hashIndex);
        long end = System.nanoTime();
```

```
        System.out.println("Tempo HashMap: " + (end - start) /
1_000_000.0 + " ms");

        // Indexação e medição de tempo usando TreeMap
        start = System.nanoTime();
        indexFile(filePath, treeIndex);
        end = System.nanoTime();
        System.out.println("Tempo TreeMap: " + (end - start) /
1_000_000.0 + " ms\n");

        // Exibição de parte do resultado
        System.out.println("Palavras indexadas (HashMap): " +
hashIndex.size());
        System.out.println("Palavras indexadas (TreeMap): " +
treeIndex.size() + "\n");

        // Exemplo de saída ordenada (TreeMap)
        System.out.println("Exemplo de indexação ordenada
(TreeMap):");
        int count = 0;
        for (Map.Entry<String, Integer> entry :
treeIndex.entrySet()) {
            System.out.println(entry.getKey() + " -> " +
entry.getValue());
            if (++count >= 10) break; // limita a 10 itens
        }
    }

    /**
     * Função que lê um arquivo e conta a frequência de cada
palavra.
     * @param path Caminho do arquivo
     * @param map Estrutura de indexação (HashMap ou TreeMap)
     */
    private static void indexFile(String path, Map<String, Integer>
map) {
        try (BufferedReader reader = new BufferedReader(new
FileReader(path))) {
            String line;
            while ((line = reader.readLine()) != null) {
                // Divide as linhas em palavras, ignorando pontuação
```

```
                String[] words = line.toLowerCase()
                        .replaceAll("[^a-zà-ú0-9 ]", "")
                        .split("\\s+");

                for (String word : words) {
                    if (word.isEmpty()) continue;
                    // Incrementa a contagem da palavra
                    map.put(word, map.getOrDefault(word, 0) + 1);
                }
            }
        } catch (IOException e) {
            System.err.println("Erro ao ler o arquivo: " +
e.getMessage());
        }
    }
}
```

Etapa 3 – Execute o programa

1. Clique com o botão direito sobre o arquivo `WordIndexer.java` → **Run 'WordIndexer.main()'**.

2. Observe no console os resultados:

    - O tempo total de execução para `HashMap` e `TreeMap`.

    - Quantas palavras foram indexadas.

    - Algumas palavras com suas frequências.

> *Dica:* O `HashMap` geralmente é mais rápido, mas o `TreeMap` mantém as chaves ordenadas alfabeticamente.

Etapa 4 – Depure o código (modo Debug)

Coloque um **breakpoint** na linha abaixo dentro do método `indexFile`:

```
map.put(word, map.getOrDefault(word, 0) + 1);
```

1.
2. Execute o programa em modo **Debug (Shift + F9)**.

3. Observe no painel **Variables**:
    - O conteúdo de `map` (as palavras que estão sendo indexadas).
    - O valor de `word` e sua contagem.
4. Continue a execução linha a linha para perceber o comportamento da estrutura.

Etapa 5 – Experimente e compare

Tente as variações abaixo e registre os resultados:

Estrutura	Tempo (ms)	Mantém ordem?	Tipo de ordenação
HashMap		❌	—
TreeMap		✅	Alfabética
LinkedHashMap		✅	Ordem de inserção

1. Substitua `new HashMap<>()` por `new LinkedHashMap<>()`.
2. Execute novamente e observe que a ordem de inserção agora é preservada.
3. Compare os tempos de execução e discuta os resultados.

Etapa 6 – Desafio opcional

Modifique o programa para exibir as **10 palavras mais frequentes** no texto.

> Dica: use o método `entrySet()` e o método `stream()` para ordenar as entradas por valor (frequência).

Exemplo de saída esperada:

```
Palavras mais frequentes:
a -> 15
de -> 12
dados -> 8
...
```

Entrega

Entregue um **relatório curto (1 a 2 páginas)** contendo:

1. **Print da tela de depuração** mostrando o `map` sendo preenchido.
2. **Tabela comparando os tempos** de execução entre `HashMap`, `TreeMap` e `LinkedHashMap`.
3. **Conclusão**, respondendo:
    - Qual estrutura é mais rápida?
    - Qual mantém os dados ordenados?
    - Qual seria mais adequada para implementar um **índice de busca textual**?