

# Relatório Verificador de Similaridades

**Nome:** Giovanna Borges Coelho - **RA:** 10756784

**Nome:** Luís Fernando de Mesquita Pereira - **RA:** 10410686

**Nome:** Gabriel Leal Leone - **RA:** 10402494

## Introdução

O aumento de documentos digitais exige ferramentas capazes de identificar similaridade entre textos. Este projeto implementa um verificador de similaridade usando estruturas de dados desenvolvidas manualmente, conforme solicitado no PDF do trabalho.

O sistema faz:

- leitura e normalização de texto,
- remoção de pontuação e stopwords (ex.: lista ptbr.txt),
- armazenamento de frequências em **Hash Table implementada manualmente**,
- cálculo da similaridade usando **métrica do cosseno**,
- Organização dos resultados em uma **Árvore AVL**

## Metodologia

### 2.1 Processamento de Texto

A classe Documento implementada realiza:

- leitura de arquivo,
- conversão para minúsculas,
- remoção de pontuação,
- tokenização,
- remoção de stopwords,
- inclusão das palavras em uma HashTable de frequências.

### 2.2 Tabela Hash (implementação manual)

A HashTable<K,V> implementada realiza:

- encadeamento separado,
- classe interna Entry,

- funções put, get, remove,
- duas funções de hash selecionáveis,
- método getKeys() para permitir varredura (necessário para similaridade).

A estrutura de dados desenvolvida é uma Tabela Hash (Hash Table) genérica, projetada para armazenar pares de chave-valor ( $\langle K, V \rangle$ ) para resolver conflitos de mapeamento (quando duas chaves distintas resultam no mesmo índice do vetor), foi adotada a estratégia de **Endereçamento Fechado (Closed Addressing)**.

- **Estrutura Interna:** A tabela é composta por um array de listas encadeadas (`LinkedList<Entry<K, V>>[]`).
- **Funcionamento:** Cada posição do array funciona como o cabeçalho de uma lista. Quando ocorre uma colisão, o novo par chave-valor é simplesmente adicionado ao final da lista correspondente àquele índice.
- **Justificativa:** Essa abordagem permite que a tabela armazene um número de elementos superior à sua capacidade física (Fator de Carga  $> 1.0$ ) sem falhar, degradando a performance apenas nos buckets superlotados.

### Funções de Dispersão

A implementação oferece duas funções de hash distintas para fins de análise comparativa de entropia e distribuição. A escolha da função é definida no construtor da classe. O mapeamento final para o índice do vetor é feito através da operação de módulo: `Math.abs(hash) % capacity`.

- **Função Polinomial:** Ela calcula o hash iterando sobre os caracteres da chave (String) e aplicando a fórmula polinomial: `hash = (31 * hash + char)`. O número 31 é um primo ímpar, o que reduz chances de cancelamento na operação de módulo. Além disso, essa função considera a posição dos caracteres. Logo, anagramas como "roma" e "amor" produzem hashes totalmente diferentes, minimizando colisões em textos reais.
- **Função de Soma Simples:** Implementada estritamente para comparação (baseline), esta função apenas soma os valores ASCII dos caracteres.

## 2.3 Métrica de Similaridade

A classe Similaridade implementada utilizando cosseno entre vetores de frequência. Essa métrica foi escolhida, pois ela normaliza automaticamente o tamanho dos documentos, permitindo comparar de forma justa textos curtos com textos longos, focando na composição das palavras e não somente na contagem bruta das palavras como outras técnicas.

Isso permite detectar plágio ou similaridades mesmo quando um documento é um subconjunto ou uma versão estendida da outra.

## 2.4 Estrutura de Armazenamento (AVL)

A classe AVLTree foi implementada como uma extensão de uma Árvore Binária de Busca (BST), herdando a lógica de inserção básica e adicionando a camada de auto balanceamento. O objetivo principal desta estrutura é garantir que a altura da árvore permaneça logarítmica, assegurando eficiência nas operações de busca e inserção, mesmo no pior caso.

- **Balanceamento:** Após cada inserção recursiva (no método `inserirRecursivo`), a altura do nó ancestral é atualizada e o método `balancear(Nó)` é invocado. Este método verifica se o nó violou a regra da AVL.
- **Classificação das Rotações:** A decisão de qual rotação aplicar depende não apenas do nó desbalanceado, mas também do sinal do FB do seu filho (o lado "pesado"). A implementação trata os quatro casos clássicos:
  - **Rotações Simples:** Ocorrem quando o desbalanceamento é linear (sinais iguais).
    - **Caso Esquerda-Esquerda (LL):** O nó está pesado para a esquerda ( $FB > 1$ ) e seu filho esquerdo também ( $FB \geq 0$ ). Aplicamos uma `rotacaoDireita`.
    - **Caso Direita-Direita (RR):** O nó está pesado para a direita ( $FB < -1$ ) e seu filho direito também ( $FB \leq 0$ ). Aplicamos uma `rotacaoEsquerda`.
  - **Rotações Duplas:** Ocorrem quando o desbalanceamento é em "zigue-zague" (sinais opostos).
    - **Caso Esquerda-Direita (LR):** O nó é pesado para a esquerda ( $FB > 1$ ), mas o filho tende à direita ( $FB < 0$ ). Primeiro rotacionamos o filho à esquerda, depois o nó atual à direita.
    - **Caso Direita-Esquerda (RL):** O nó é pesado para a direita ( $FB < -1$ ), mas o filho tende à esquerda ( $FB > 0$ ). Primeiro rotacionamos o filho à direita, depois o nó atual à esquerda.
- **Contagem das Rotações:** A contagem das rotações é feita utilizando dois contadores globais (`totalRotacoesSimples` e `totalRotacoesDuplas`), eles são incrementados dentro

do método balancear (na lógica de decisão if/else) e não dentro dos métodos de rotação em si.

## Descrição do Algoritmo

- **Etapa 1: Validação e Configuração Inicial**

- O programa inicia validando os argumentos recebidos via linha de comando (args).
- Verificação de Argumentos: Garante que foram passados no mínimo 3 parâmetros (diretório, limiar, modo).
- Validação de Diretório: Checa se o caminho informado existe e é uma pasta válida.
- Conversão de Dados: Transforma o argumento de limiar (String) para double. Caso falhe, encerra com erro.
- Instanciação: Cria o objeto Configuracao para carregar os parâmetros.

- **Etapa 2: Carregamento de Recursos (Pré-processamento)**

- Antes de processar os textos, o sistema prepara o ambiente.
- Stop Words: Lê o arquivo ptbr.txt, remove espaços, converte para minúsculas e armazena em um Set<String> (para busca rápida O(1)).
- Identificação de Arquivos: Varre o diretório informado buscando apenas arquivos terminados em .txt.
- Objetos Documento: Instancia objetos da classe Documento para cada arquivo encontrado, mas ainda sem ler o conteúdo (Lazy loading).

- **Etapa 3: Roteamento de Execução (Modos)**

- Um estrutura switch decide qual fluxo seguir com base no argumento <modo>:
- lista: Compara todos contra todos e exibe pares acima do limiar.
- topK: Similar ao lista, mas limita a exibição aos K pares mais similares (exige 4º argumento).
- busca: Compara apenas dois arquivos específicos informados nos argumentos extras.

- **Etapa 4: Processamento e Comparação (Núcleo)**

- O método `processarDocumentos` orquestra a lógica pesada:
- **Processamento Individual:** Cada documento lê seu texto, normaliza, remove stop words e popula sua Tabela Hash interna de frequências.
- **Formação de Pares:** O método `criarParesDocumentos` gera todas as combinações possíveis de documentos dois a dois (Combinação Simples).
- **Cálculo de Similaridade:** O `ComparadorDeDocumentos` usa a métrica (Cosseno) para calcular a similaridade entre os pares gerados.
- **Inserção na AVL:** Cada resultado (par de documentos + similaridade) é inserido na Árvore AVL, usando a similaridade como chave de ordenação.

- **Etapa 5: Geração de Saída (Output)**

- Finaliza a execução apresentando os dados.
- **Consulta na AVL:** O método `processarResultados` percorre a árvore para recuperar os pares ordenados (maiores e menores similaridades) respeitando o limiar e o TopK.
- **Formatação:** Monta uma String formatada com estatísticas (total processado, colisões da hash, pares encontrados).
- **Persistência:** Exibe o relatório no console e grava o arquivo `resultado.txt` no disco.

## Análise Experimental

- **Rotações da árvore AVL:** durante a execução do algoritmo para um conjunto de dados inicial de 5 documentos, a Árvore AVL registrou um total de 2 rotações simples e 0 rotações duplas. A ocorrência exclusiva de rotações simples indica que os desbalanceamentos encontrados foram do tipo "Esquerda-Esquerda" ou "Direita-Direita". Isso sugere que, em determinados momentos, os dados foram inseridos em uma ordem parcialmente ordenada (crescente ou decrescente). Como também, a ausência de rotações duplas demonstra que não houve casos onde um nó inserido causou um desbalanceamento interno oposto ao do pai (casos "Esquerda-Direita" ou "Direita-Esquerda"). Isso implica que a sequência de entrada não apresentou oscilações agudas imediatas nos valores de similaridade durante a construção da estrutura.
  - Conforme o resultado da similaridade calculada:
    - Pares com similaridade  $\geq 0,70$
    - -----
    - doc4.txt <-> doc5.txt = 1,00
    - -----
    - Pares com menor similaridade
    - -----
    - doc1.txt <-> doc2.txt = 0,56
    - doc1.txt <-> doc3.txt = 0,37
    - doc2.txt <-> doc3.txt = 0,32
    - doc1.txt <-> doc4.txt = 0,15
    - doc1.txt <-> doc5.txt = 0,15
    - doc3.txt <-> doc4.txt = 0,13
    - doc3.txt <-> doc5.txt = 0,13
    - doc2.txt <-> doc4.txt = 0,12
    - doc2.txt <-> doc5.txt = 0,12

## Análise das funções de hash implementadas

Função Hash	Documento	Número de Buckets	Elementos armazenados	Fator de carga	Buckets vazios	Buckets usados	Colisões
Polinomial	doc1.txt	931	325	0,3491	660	271	54
	doc2.txt	172	99	0,5756	96	76	23
	doc3.txt	126	73	0,5794	74	52	21
	doc4.txt	114	63	0,5526	62	52	11
	doc5.txt	114	63	0,5526	62	52	11
Soma simples dos caracteres	doc1.txt	931	325	0,3491	679	252	73
	doc2.txt	172	99	0,5756	101	71	28
	doc3.txt	126	73	0,5794	71	55	18
	doc4.txt	114	63	0,5526	73	41	22
	doc5.txt	114	63	0,5526	73	41	22

## Conclusões

O principal desafio conceitual na Tabela Hash foi a compreensão profunda do impacto das colisões. Implementar o tratamento por Endereçamento Fechado (Listas Encadeadas) exigiu um cuidado extra para evitar `NullPointerExceptions` na inicialização dos buckets.

Embora a lógica de rotações (simples e duplas) seja complexa devido à manipulação precisa de ponteiros, o maior desafio técnico do projeto revelou-se na implementação dos métodos de travessia condicional, especificamente o `exibirMaiores`. Diferente de uma travessia padrão (In-Order), este método exigia uma lógica de "Filtragem + Ordenação Reversa + Limitação" (Top K). Para corrigir o erro, alterei a estratégia de controle de parada. Em vez de depender de um contador decrescente passado por parâmetro, passei a verificar o tamanho da lista de resultados acumulada (que é um objeto mutável, `ArrayList`, passado por referência). A condição de parada passou a ser:

```
if (pares.size() < topK) {  
    // continua a busca recursiva  
}
```

Dessa forma, todas as instâncias da recursão "enxergam" a mesma lista e sabem exatamente quando parar de adicionar novos elementos, resolvendo o bug e garantindo a eficiência da busca.