# Trabalho Prático 3
# Máquina de Busca Avançada

**Igor Lacerda Faria da Silva**

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG) - Belo Horizonte - MG - Brasil

igorlfs@ufmg.br

## 1 Introdução

O problema proposto foi implementar um **indexador de memória** e um **processador de consultas**, que a partir do indexador e de uma busca produz um *ranking* com os documentos mais relevantes do indexador. O indexador, por sua vez, é construído a partir de um *corpus* (um diretório passado como parâmetro, contendo os documentos) e um arquivo de *stopwords*, ambos passados como parâmetros na execução do programa. *Stopwords* são palavras muito comuns que em nada agregam numa busca, como *the*, *of*, *a* e etc. As consultas são realizadas com o uso do Modelo Espaço Vetorial, que usa princípios de álgebra linear para criar uma classificação consistente e robusta. Foi usada uma variação desse modelo que normaliza as frequências das palavras nos documentos. Além disso, tanto o *ranking* como a *query* também eram passadas como parâmetros.

Esta documentação tem como proposta explicar como se deu essa implementação, desde questões mais ligadas ao funcionamento do programa (Seção 2) como análises de complexidade (Seção 3) e análises experimentais (Seção 6), as estratégias de robustez adotadas (Seção 4) e os testes realizados (Seção 5). Ao final do texto, encontra-se uma conclusão (contendo o aprendizado pessoal do autor com o trabalho), bibliografias e, por último, as instruções para compilação e execução.

## 2 Método

O programa foi desenvolvido em C++ e compilado utilizando o g++, do GCC. A máquina que foi usada durante o desenvolvimento conta com 12Gi de memória RAM, e processador AMD Ryzen 7 5700U, e roda o sistema operacional GNU/Linux (versão do kernel: 5.16.9).

A formatação do código fonte (**incluindo a indentação**) **foi feita automaticamente usando a ferramenta clang-format**. Foi usado um arquivo

customizado para isso, que se encontra na raiz do projeto, com o nome de *.clang-format*. É um arquivo bem curto, baseado em preferências pessoais do autor, mas que **garante a consistência da formatação do projeto**.

## 2.1 Organização do código e detalhes de implementação

O projeto atende à especificação no que diz respeito à organização do código de forma geral. Em particular, a única divergência é que os *headers* usam a extensão `.hpp` e não `.h`. Os diversos arquivos de código estão organizados da seguinte maneira: um arquivo que executa o programa principal, uma definição e implementação de uma classe `InverseIndex`, uma implementação de listas *linkadas* (`cell.hpp`, `linearList.hpp`, `linkedList.hpp` e `linkedList.cpp`) com base no TP 1 e dois *headers* que definem ordenações com *templates* em C++ (devido a idiossincrasias de C++ com *templates*, esses arquivos permaneceram como *headers*). Os arquivos compiláveis se encontram na pasta *src*, os *headers* na pasta *lib*, os objetos em *obj* e o binário após a *linkagem* na própria raiz do projeto.

A classe `InverseIndex` sem dúvidas é o maior alicerce deste programa. Com mais de 250 linhas, ela é responsável tanto por gerar o índice como por processá-lo (considerou-se adequado trabalhar com apenas uma classe nessa instância, visto que todas as operações permeiam o índice). Ademais, ela conta com um "hash", com sua própria função de *hashing* e manejamento de colisões. O arquivo que executa o programa principal apenas lê os argumentos e invoca as funções de criação e processamento da classe `InverseIndex`.

Outra questão de fundamental importância foi a implementação do tratamento de arquivos. Atendendo à especificação, não foram considerados caracteres não ASCII (letras com acento, cê-cedilha e afins). De fato, o tratamento, implementado em `clearFile()` consistiu em reescrever o arquivo da seguinte maneira: lê-se o arquivo caractere a caractere, se o caractere for alfabético (pela função `isalpha()`), ele é reescrito como minúsculo, e caso não o seja, ele é reescrito como um espaço (*ie*, ' '). O arquivo resultante dessa transformação contém apenas uma linha, e é simples de ser lido usando a *stream* do arquivo. Esse tratamento levou a pequenas divergências nos resultados dos testes que foram disponibilizados no *moodle*.

Com respeito às boas práticas, foi usado *camelCase* nos nomes das variáveis e métodos (fora nomes tradicionais e possíveis exceções com nomes tradicionalmente em maiúsculas, como em *IDs*).

## 2.2 Estruturas de Dados, TADs e métodos

A estrutura de dados canônica deste TP é o *hash*, que é implementado como módulo da classe `InverseIndex`. Além disso, foram usadas listas encadeadas, baseadas na implementação do TP 1, para simplificar alguns métodos (é especialmente prático usar listas encadeadas quando não se sabe o tamanho de um dado *array a priori*). Também foram implementados **dois** algoritmos de ordenação, um *QuickSort* para ordenar os nomes dos arquivos do *corpus* (que

não precisa ser estável) e uma modificação do *MergeSort* que recebe dois *arrays* e é estável. Preferiu-se fazer uso ao máximo de *templates* para facilitar o reuso.

### 2.2.1 *Hash*

Optou-se por não fazer uma classe dedicada para o *hash*, uma vez que o membro `index` da classe `InverseIndex` é usado diversas vezes. Outras escolhas de desenvolvimento dessa estrutura são: alocação estática a partir de um inteiro grande arbitrário (100003) e manejamento de colisões por endereçamento aberto. Escolheu-se o número 100003 pois ele não é grande ao ponto de não caber na memória (mas razoavelmente grande) e por ser primo (o menor primo maior que 100000). Uma discussão do por quê é interessante usar números primos para esse fim é dada `aqui`. A implementação da função de *hashing* teve como base `esse` artigo.

O manejamento de colisões, implementado em `handleCollisions()`, faz uso de uma propriedade da implementação de listas encadeadas: normalmente, a célula cabeça não recebe nenhum conteúdo (o que simplifica algumas operações). Desse modo, a cabeça foi usada para armazenar o termo correspondente à lista encadeada na primeira inserção de cada lista. Para realizar inserções subsequentes, checava-se se o termo a ser *hasheado* "batia" com a cabeça da lista esperada, e caso não batesse, incrementava-se a posição até que fosse encontrada uma posição adequada.

### 2.2.2 Índice Invertido

A classe que implementa tanto a **criação** do índice como seu **processamento** é a `InverseIndex`. Para realizar essas tarefas, ela possui 13 métodos: um *getter* para recuperar a frequência de uma dada palavra numa lista (`getFrequency()`), 3 *setters* para identificadores, *stopwords*, busca e documentos, as duas funções de *hashing* e a de limpeza de arquivos discutidas anteriormente, e funções que ou implementam as principais operações ou são auxiliares (`calculateNormalizers()` e `print()`) a elas. Em particular, funções como `isInList()` e `incrementInList()` só existem porque é usada a estrutura `std::pair` de C++.

### 2.2.3 Algoritmos de Ordenação

Foram implementados dois algoritmos de ordenação, usados em etapas diferentes do programa: um *QuickSort* que ordena os números dos nomes dos arquivos (que não precisa ser estável pois os nomes de arquivos não podem se repetir) e um *MergeSort* para gerar o ranking final (que precisa ser estável para atender à especificação). Além disso, o MergeSort recebe um parâmetro adicional (um segundo *array*). O *QuickSort* seguiu praticamente a mesma implementação do TP anterior e o *MergeSort* foi inspirado na implementação do *GeeksForGeeks* (que aliás, possui um vazamento de memória).

# 3 Análise de Complexidade

A seguir, são analisadas as complexidades de tempo e de espaço dos principais métodos do buscador. Presume-se que algumas funções padrão de C++ são $\Theta(1)$. Alguns métodos simples serão omitidos, outros receberão comentários mais breves. De fato, somente os métodos da classe `InverseIndex` serão analisados em detalhes.

Foi realizada uma análise detalhada do *QuickSort* no TP 2, e então nessa instância será assumida sua complexidade de caso médio – $\Theta(n \log n)$ no número de elementos. Além disso, o *MergeSort* também não vai receber atenção particular: não é um algoritmo adaptável, é sempre $\Theta(n \log n)$ no número de elementos.

- `getFrequency()`

  **Tempo:** a lista é percorrida sequencialmente, ou seja, existem diferentes casos. No melhor caso, o primeiro item da lista é encontrado e a função retorna imediatamente, então obtém-se $\Theta(1)$. Já no pior caso a lista inteira é percorrida, chegando-se no último elemento. Portanto, o pior caso é $\Theta(n)$ no comprimento da lista.

  **Espaço:** a variável alocada não depende do tamanho da lista e os parâmetros são passados por referência. Essa função portanto é $\Theta(1)$ em complexidade assintótica de espaço.

- `clearFile()`

  **Tempo:** essa função lê todo um arquivo e o sanitiza, conforme descrito na seção 2. Considerando que a leitura e a escrita de caracteres, assim como outras operações relacionadas de manipulação de arquivos possuem, sua complexidade assintótica de tempo depende linearmente do tamanho do arquivo.

  **Espaço:** as alocações não dependem do tamanho do arquivo ($\Theta(1)$).

- `setFile()`

  **Tempo:** essa função chama `clearFile()` e lê sequencialmente o arquivo resultante, "palavra a palavra". Como `clearFile()` faz uma leitura caractere a caractere e essa função faz uma leitura por palavras, podemos dizer sua complexidade assintótica de tempo é $\Theta(m + n)$, que $m$ é o número de caracteres e $n$ o número de palavras.

  **Espaço:** cada elemento que é inserido na lista precisa precisa de memória adicional, e como não existem outra alocações relevantes, essa função é $\Theta(n)$ no número de palavras do arquivo em complexidade assintótica de tempo.

- `setDocuments()`

  **Tempo:** essa função calcula, em tempo linear, o número de documentos no diretório (usando o método `std::distance()`. Em seguida, lê o nome dos arquivos (com possíveis extensões), os ordena com base nos identificadores, e por fim *setta* o *path* dos documentos. Essas operações são, respectivamente, $\Theta(n), \Theta(n \log n)$ e $\Theta(n)$, portanto esse método é $\Theta(n \log n)$ em complexidade assintótica de tempo.

**Espaço:** essa função aloca algumas variáveis que dependem do número de documentos no diretório e portanto é $\Theta(n)$ em complexidade de espaço.

- `setIDs()`

  **Tempo:** essa função percorre uma lista removendo o desnecessário de cada documento do *corpus* (mantendo apenas os identificadores numéricos). Faz algumas operações com *strings*, mas assumindo-se $\Theta(1)$ para todas, conclui que essa função é linear em complexidade assintótica de tempo.

  **Espaço:** as alocações não dependem do número de documentos ($\Theta(1)$).

- `hash()`

  **Tempo:** essa função percorre uma *string*, caractere a caractere, aplicando algumas operações matemáticas de custo constante e portanto sua complexidade assintótica de tempo é linear.

  **Espaço:** as alocações não dependem do tamanho da *string* ($\Theta(1)$).

- `handleCollisions()`

  **Tempo:** se o termo em questão não está colidindo, obtém-se o melhor caso: $\Theta(1)$. Por outro lado, se o índice está cheio, percorre-se todo o seu comprimento. No entanto, esse comprimento é fixo (o tamanho do *array* `index`), então de certa forma, faz sentido em dizer que o pior caso é também $\Theta(1)$, porém isso não é muito descritivo. Então, pode-se afirmar que o pior caso é $\Theta(n)$ no tamanho do `index`. O caso médio é difícil de ser estimado, pois conforme o índice vai enchendo, as colisões vão ficando mais frequentes.

  **Espaço:** as alocações não dependem do tamanho dos parâmetros ($\Theta(1)$).

- `isInList()`

  **Tempo:** a lista é percorrida sequencialmente, ou seja, existem diferentes casos. No melhor caso, o primeiro item da lista é encontrado e a função retorna imediatamente, então obtém-se $\Theta(1)$. Já no pior caso a lista inteira é percorrida e o item procurado não é encontrado. Portanto, o pior caso é $\Theta(n)$ no comprimento da lista.

  **Espaço:** a variável alocada não depende do tamanho da lista e os parâmetros são passados por referência. Essa função portanto é $\Theta(1)$ em complexidade assintótica de espaço.

- `incrementInList()`

  **Tempo:** no melhor caso, o primeiro item da lista é encontrado, seu número de vezes incrementado e a função retorna imediatamente, então obtém-se $\Theta(1)$. Já no pior caso (assume-se que essa busca sempre é possível incrementar), a lista inteira é percorrida, chegando-se no último elemento. Portanto, o pior caso é $\Theta(n)$ no comprimento da lista.

  **Espaço:** a variável alocada não depende do tamanho da lista e os parâmetros são passados por referência. Essa função portanto é $\Theta(1)$ em complexidade assintótica de espaço.

- `calculateNormalizers()`

**Tempo:** esse método é notavelmente mais complexo (e mais complicado de analisar) que os anteriores. Essa função faz um laço entre os $D$ documentos. Dentro desse laço principal, há um outro laço, que percorre todas as $M$ posições do *hash*. Se a lista associada a um termo estiver vazia, o laço é continuado. Caso contrário, tenta-se recuperar a frequência associada ao documento em questão para aquela lista pelo método `getFrequency()`, que como discutido anteriormente, é $O(k)$ no pior caso. O restante do laço não contribui significativamente para a análise assintótica. No melhor caso, quando o índice está vazio, o método `getFrequency()` nunca é chamado e a complexidade da função é $\Omega(DM)$. Já no pior caso, o método sempre é chamado. Assumindo que todas as listas tem um tamanho máximo $k$, conclui-se que a complexidade do pior caso é $O(DMk)$, ou seja, é cúbica.

**Espaço:** esse método não aloca memória com base no tamanho dos parâmetros que recebe (passados por referência), então é $\Theta(1)$.

- `print()`

  **Tempo:** essa função apenas imprime no arquivo de saída o "top" 10 documentos mais relevantes, se existirem (se não é impresso até os documentos acabarem). Como seu laço é executado no máximo 10 vezes, sua complexidade assintótica de tempo é $\Theta(1)$.

  **Espaço:** essa função não aloca memória a depender de seus parâmetros e por isso é $\Theta(1)$ em complexidade assintótica de tempo.

- `createIndex()`

  **Tempo:** esse método é também complicado de ser analisado. A primeira operação relevante é o `setFile()` para o arquivo de *stopwords*. Seja $a$ o número de caracteres no arquivo e $b$ o número de palavras, o método começa com $\Theta(a + b)$. Em seguida, os documentos são configurados em um custo $\Theta(n \log n)$ pela análise da função `setDocuments()`. O passo seguinte é um laço no número $n$ de documentos. A primeira etapa é limpar os arquivos, em tempo $\Theta(k)$ no número de caracteres máximo. A segunda etapa é um laço no número $l$ de palavras no máximo nos documentos. Primeiro, confere-se se a palavra é uma *stopword* em tamanho linear na lista de *stopwords*. Assumindo que não seja *stopword*, o *hashing* é realizado em tempo linear no tamanho $m$ de cada palavra. Em seguida, as colisões são tratadas em no mínimo tempo constante e no máximo $O(M)$, tamanho total do `index`. Por fim, é feita uma busca na lista cuja posição foi tratada que é $O(p)$ no pior caso e existem dois casos: se a busca retornar falso, o item é inserido na lista, assim como possivelmente o tratamento para colisões (em tempo constante). Caso contrário, a lista é mais uma vez buscada para que se incremente a posição correta, novamente em tempo $O(p)$. Juntando todas essas operações, conclui-se que este método é $O(a + b) + O(n \log n) + O(n(k + l(b + m + M + p)))$.

  **Espaço:** a complexidade assintótica de espaço, em contra partida, não é tão difícil de ser analisada. A maioria das variáveis declaradas não depende dos parâmetros passados. No entanto, as funções auxiliares

`setFile()` e `setDocuments()` alocam memória para suas respectivas listas. Sendo $n$ o número de palavras no arquivo e sejam $k$ documentos, esse método possui complexidade assintótica de espaço $O(n + k)$.

- `process()`

  **Tempo:** o último método é também o mais difícil de analisar. A *query* é definida em $\Theta(a)$, no número de palavras, depois os documentos em tempo $O(D)$ e os pesos dos documentos em $O(DMk)$ (adicionavelmente é usada a função `memset` para zerar todos os valores). A etapa seguinte é fazer um laço nas $a$ palavras da *query*, *hasheando* com base no tamanho $l$ da string e tratando colisões, na pior das hipóteses, em tempo $O(M)$. Se a palavra está ausente, ela é pulada. Se está presente, é feito um laço nos $D$ documentos, onde se busca a frequência pela lista de tamanho $p$ e são realizadas algumas operações de custo constante. No terceiro bloco de código, a *query* é normalizada em $\Theta(D)$, os vetores ordenados em $\Theta(D \log D)$ e o resultado impresso em tempo constante. Juntando todas essas informações e simplificando um pouco, conclui-se que o pior caso é $O(DMk) + O(D \log D) + O(a(l + M + Dp))$.

  **Espaço:** são feitas diversas alocações que dependem do número de documentos $D$. Além disso, também é alocado espaço para a *query*. Logo, a complexidade assintótica de tempo desse método é $\Theta(D + n)$.

A complexidade do programa como um todo é a soma da complexidade da função `process()` e da função `createIndex()` (em termos de complexidade assintótica de tempo). Em espaço deve ser considerado também a memória alocada ao se declarar o objeto `I` do tipo `InverseIndex`, que é constante mas não é barata em termos computacionais.

# 4 Estratégias de Robustez

Não foram implementadas muitas estratégias de robustez, somente o necessário para garantir o funcionamento do programa. Isso inclui abortar o programa quando houver falha ao alocar memória (usando o operador `new`) na classe `LinkedList` e constante tratamentos de arquivos: sempre é conferido se o arquivo foi aberto, escrito (ou lido) e fechado corretamente. Todas essas condições são checadas se usando o `msgassert`, biblioteca inspirada na implementação do Wagner Meira Júnior para tratar esses tipos de erro.

Adicionalmente são tratados algumas outras exceções: ao se tratar as colisões de *hash*, é possível que o índice fique cheio. Tendo isso em mente, a função `handleCollisions()` possui um contador para averiguar se o índice encheu e, caso afirmativo, abortar o programa. Outra exceção está no tratamento do *path* do corpus: a função `setIDs()` espera receber que o *path* contenha o caractere '/' (como é definido na função `setDocuments()`).

Os parâmetros do programa também são checados para averiguar se algum ficou vazio (o que não deve ocorrer). Como a função `stoi()` é usada em `setDocuments()`, o *path* "`.`" para o `corpus` é proibido.

# 5 Testes Realizados

Foram realizados 4 tipos de teste do **programa completo**: o caso "base" da especificação (com as palavras casa, maca e lua), uma modificação para testar a validade do tratamento de colisões ("collide") e os casos do *moodle*, tanto para o *corpus small* como para o *corpus full* ("advanced"). Não foram realizados testes de unidade, porque a maioria das funções usadas são privadas (talvez isso seja um erro de design).

Os casos, como descrito anteriormente, se encontram no diretório `test`. Como o programa sobrescreve os arquivos de entrada, cada diretório conta com um subdiretório `copy` que é uma cópia do diretório pai, exceto por si mesmo e por umas pasta com as saídas: `outputs`. A exceção a essa regra são os casos do *moodle*, cuja pasta `advanced` é separada de uma pasta `moodle` que contém os dados vindos diretamente do *moodle*.

A saída do caso base está igual à esperada, como é possível conferir em seu respectivo diretório. O mesmo vale para a checagem de colisões. Mais especificamente, no teste de colisões, "lua" vira *moon*, "maca" vira *silver* e "casa" vira *dimension* (foi testado separadamente que *silver* e *dimension* colidem dada a função de *hash* usada). Além disso, as *stopwords* foram trocadas para correspondentes no inglês. Mas o resultado foi exatamente igual ao caso anterior, como esperado.

Foi ao realizar os testes do *moodle* que a situação complicou. Como apontado por alguns alunos no fórum, os resultados obtidos diferem um pouco das saídas esperada, no geral sendo permutações destas. Acredita-se, no entanto, que essas diferenças sejam resultantes de tratamentos razoavelmente diferentes para se trabalhar apenas com caracteres alfabéticos em ASCII. De fato, o problema de verdade está relacionado à performance: a execução do programa para o *corpus full* demora cerca de 13 minutos na máquina em que foi testada. Investigou-se fazer uso de *multithreading* para melhorar a performance, o que acabou se provando muito complicado. Em função dessas dificuldades, somente a primeira consulta foi testada no *corpus full*.

# 6 Análise Experimental

*Por uma questão de tempo essa análise não foi realizada.*

# 7 Conclusões

Neste trabalho foi implementado um buscador de termos em arquivos de texto, usando o Modelo Espaço Vetorial com normalização. Foi usado um *hash* para se criar um índice invertido que é processado, gerando um *ranking* com os arquivos mais promissores.

### 7.1 Aprendizado Pessoal

Esse trabalho foi razoavelmente mais complicado de implementar que o anterior. Foi difícil entender os requerimentos. Como *a priori* não sabia se podia usar STL, implementei parte do programa usando as bibliotecas `<map>` e `<list>`, o que facilitou bastante a implementação de verdade, sem STL. De fato, foi como realizar uma *tradução* da versão inicial. Finalmente usei *templates*, o que ficou até bacana, mas não gostei de deixar implementações em *headers*. Foi bom usar a lista encadeada que tinha feito no TP 1. Outra questão de implementação que não foi de encontro ao meu agrado foi fazer uma classe **massiva** (o arquivo de *source* possui cerca de 300 linhas!) para lidar com o indexador. Acho que poderia ter fracionado melhor isso (até por uma questão de testes de unidade), mas o tempo estava meio apertado.

As análises de complexidade desse TP foram bem mais complicadas que as anteriores. Muitas variáveis sendo utilizadas (talvez eu devesse ter feito algumas simplificações). Não achei eficiente em termos de custo fazer uma análise experimental: pouco tempo e poucos pontos para muito trabalho.

Ficam aqui meus agradecimentos aos professores e monitores pela disciplina.

## 8 Bibliografia

1. MERGE Sort. [S. l.], 10 jan. 2022. Disponível em: `https://www.geeksforgeeks.org/merge-sort/`. Acesso em: 20 fev. 2022.
2. STRING Hashing. [S. l.], 16 jan. 2022. Disponível em: `https://cp-algorithms.com/string/string-hashing.html`. Acesso em: 20 fev. 2022.

# Instruções

## Compilação

Você pode compilar o programa da seguinte maneira:

1. Utilize o comando `cd` para mudar de diretório para a localização da raiz do projeto;
2. Utilize o comando `make`.

## Execução

Você pode rodar o programa da seguinte maneira:

1. Utilize o comando `cd` para mudar de diretório para a localização da raiz do projeto;
2. Insira o comando `./binary -i <consulta> -o <saída> -c <corpus> -s <stopwords>`. Todos os parâmetros são obrigatórios.