INTRODUÇÃO

Link do repositório: https://github.com/AnneIsabelleRodrigues/maquina-busca

O trabalho prático: Máquina de Busca foi implementado em C++ e neste foi aplicada grande parte da carga teórica dada em sala de aula. O projeto propunha a construção de um índice invertido, a formação de estruturas de dados de armazenamento, a formação de um cosine ranking a partir dos dados adquiridos e um teste de unidade para o método implantado.

A partir do apresentado, houve a necessidade de utilizar conteúdos de programação que envolvem armazenamento de dados, tipos abstratos de dados e testes de unidade.

IMPLEMENTAÇÃO

A implementação da Máquina de Busca foi separada em quatro documentos significativos:

busca.h: Header que declara a Classe Termo. A classe é usada como estrutura de composição do Indice Invertido. Permite organizar uma string (querie) junto à um map que indica o número de ocorrências da string em um dado documento.

busca.cpp: Código de implementação das funções da Classe Termo e do Struct Conteudo.

main.cpp: Implementa a entrada de dados, o cálculo de coordenadas e o *cosine ranking* em um único bloco.

busca_teste.cpp: Declara e implementa os testes referentes às funções da Classe Termo.

ENTRADA DE DADOS

O documento consulta.txt, o qual é sempre acessado, introduz ao programa quais devem ser os documentos de entrada que vão ser utilizados para o banco de dados de consultas. A máquina de busca recebe a expressão de busca a partir do terminal no seguinte formato:

```
Entre com sua pesquisa: Camisa estampada de flamingo 0
```

Cada palavra deve ser separada por espaço e, para finalizar a querie, deve ser inserida uma string zero (zero seguido de espaço e enter).

ESTRUTURAS DE DADOS PARA ARMAZENAMENTO

O Índice Invertido foi implementado a partir de uma List <Termo>. Esta armazena todas as palavras juntamente com a identificação do documento e o número de ocorrências.

A utilização de outras estruturas de dados também foram necessárias. As coordenadas W(d,p) foram armazenadas em um vector de 2 dimensões, e as coordenadas das palavras de pesquisa W(q,p) em um vector de 1 dimensão.

Termo palavra_ = "camisa" docs_ = <1,2> , <3,1> 0	*Termo* palavra_ = "tem" docs_ = <1,1> 1	*Termo* palavra_ = "flamingo" docs_ = <1,2> , <2,1> 2	*Termo* palavra_ = "calvin" docs_ = <4,2> , <3,1> 3	*Termo* palavra_ = "klein" docs_ = <1,1> , <4,1> 4

Figura 1 - Representação do Índice Invertido.

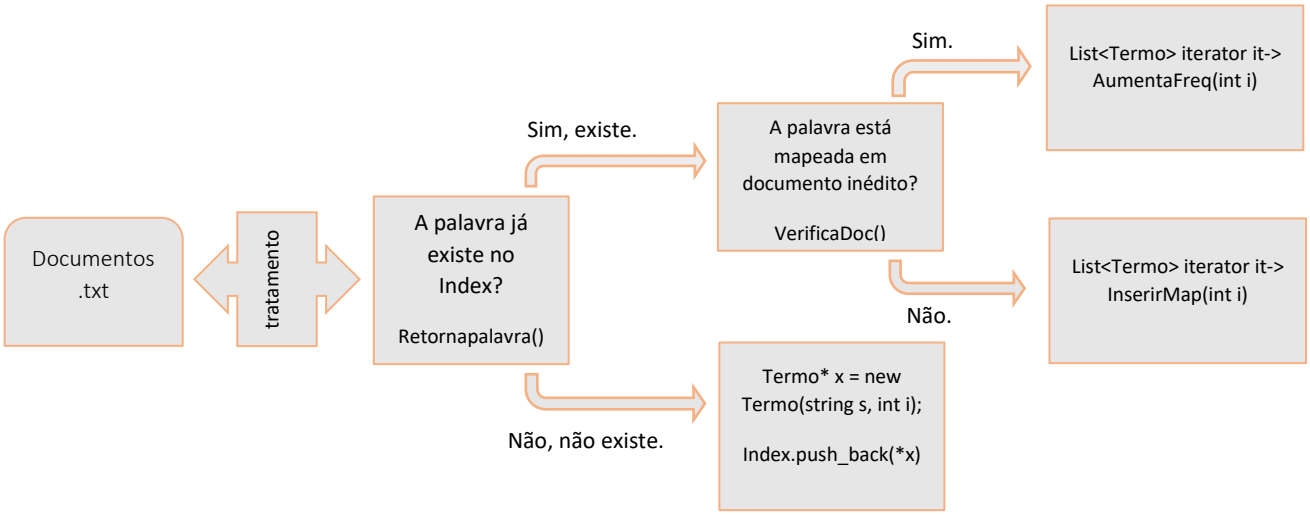
String	Documento 1	Documento 2	Documento 3	Documento 4
camisa	$W(D_1,p_1)$	$W(D_2,p_1)$	$W(D_3,p_1)$	$W(D_4,p_1)$
tem	$W(D_1,p_2)$	$W(D_2,p_2)$	$W(D_3,p_2)$	$W(D_4,p_2)$
flamingo	$W(D_1,p_3)$	$W(D_2,p_3)$	$W(D_3,p_3)$	$W(D_4,p_3)$
calvin	$W(D_1,p_4)$	$W(D_2,p_4)$	$W(D_3,p_4)$	$W(D_4,p_4)$
klein	$W(D_1,p_5)$	$W(D_2,p_5)$	$W(D_3,p_5)$	$W(D_4,p_5)$

Figura 2- Representação do Vector de coordenadas de documento.

A list *Index* tem sempre o tamanho do número de palavras não repetidas introduzidas no banco de dados para consulta. A matriz *coordenada* de documentos tem dimensão Index.size() x Número de documentos; o vector *search* referente à coordenada de busca tem tamanho Index.size(), porém, é repleta de zeros, já que apenas uma pequena parcela das palavras do banco de dados são usadas como expressão de busca por vez. Em tempo, o vector *similaridade* tem o tamanho igual ao número de documentos de consulta.

FUNCIONAMENTO

A construção do índice invertido acontece simultâneamente à entrada dos dados de consulta. Para cada palavra de cada documento realiza-se o tratamento (retirada de símbolos e letras maiúsculas passam a ser minísculas) e verifica se a nova palavra já existe no índice. Se sim, cria-se um map para representar a ocorrêcia da palavra (valor mapeado) em um documento inédito (key) ou aumenta o valor mapeado de um map já existente. Se não existir, um novo Termo é criado e adicionado ao STL list Index.



Em seguida, é calculado as coordenadas do documento no eixo de cada palavra. Para cada palavra é adquirida a sua frequência em cada documento (correspondente ao valor mapeado) e a importancia desta no documento, dados que podem ser encontrados no índice invertido que foi criado. Os resultados são armazenados no vector *coordenadas*.

É necessário calcular ainda, as coordenadas da expressão de busca (querie) no eixo de cada palavra. Os mesmos cálculos são feitos, mas em relação à string *q,* aonde foram armazenadas as palavras de busca. Os resultados são armazenados no vector *search.*

Com os vectors *coordenadas* e *search*, podemos calcular facilmente a similaridade entre os documentos e a querie. Multiplicamos os valores das coordenadas do documento e os valores das coordenadas da pesquisa para cada ocorrência da expressão de busca, em seguida somamos os resultados. Esse valor é armazenado na variável *acumsup*. O quadrado dos valores de cada ocorrência em *coordenadas* e *search* que devem ser somados são armazenados nas variáveis *sumD* e *sumQ*, respectivamente. No momento em que todas as palavras são contempladas no cálculo, a raíz quadrada de *sumD* e raíz quadrada de *sumQ* são multiplicados e formam a variável *acuminf*. A similaridade de cada documento corresponde a acumsup/acuminf. Um resumo da lógica dos cálculos implementados podem ser vistos abaixo:

$$acumsup = \sum_{i=0}^{tamq} (coordenadas[posição\ da\ palavra][documento] * search[posição\ da\ palavra])$$

$$sumD = \sqrt{\sum_{i=0}^{tamq} (coordenadas[posição\ da\ palavra][documento])^2}$$

$$sumQ = \sqrt{\sum_{i=0}^{tamq} (search[posição\ da\ palavra])^2}$$

$$acuminf = \sqrt{sumD} * \sqrt{sumQ} \qquad similaridade[documento] = \frac{acumsup}{acuminf}$$

O somatório é feito com base em um for que percorre cada palavra da list *Index* a procura da posição das palavras que compõem a expressão de busca. Recalculamos os valores para cada documento e armazena-se os dados no vector *similaridade*.

Dessa forma, os valores do vector *similaridade* permitem a construção de um ranking de importância em ordem crescente e priorizando a ordem de entrada dos documentos, caso a similaridade seja idêntica.

CONCLUSÃO

O trabalho foi um desafio no sentido de organização do código e de desbug. Com certeza a inexperiência é um aspecto que prejudica o processo de desbug e, no caso da organização e construção do código, posteriormente, ficam claras outras ideias que podem ser impregadas com a finalidade de tornar o código superior.

De certo, o projeto foi uma ótima forma de entender o funcionamento e aplicação de vários dos tópicos de estudo que nos foram apresentados até o momento e aprimorá-los. Ademais, o trabalho permitiu a exteriorização dos pontos fracos que precisam ser trabalhados, em relação à um mau-comportamento de programação ou à um conhecimento insuficiente em determinado conteúdo.