Universidade Federal de Minas Gerais
Programação e Desenvolvimento de Software II

Trabalho Prático: Máquina de Busca

Alícia Marzola Chaves | 2022036063
Clara Garcia Tavares | 2022103380
José Gabriel Vieira de Souza | 2022035865

---

1. Introdução

O primeiro passo do trabalho foi dividir as etapas de desenvolvimento. Dessa forma, ele foi separado em:

1. Leitura de arquivos
2. Normalização do conteúdo
3. Índice Invertido
4. Função de busca
5. Chamada de funções no main
6. Elaboração do Makefile
7. Criação de exceções
8. Comentários
9. Revisão do código

A implementação deste código foi realizada seguindo as etapas descritas abaixo:

1.1 - Preparação do Ambiente: Foi feita a configuração do ambiente de desenvolvimento para todos os integrantes do grupo. Decidiu-se utilizar a IDE Visual Studio Code com configuração para o WSL (Windows Subsystem for Linux). Isso proporcionou um ambiente padronizado para facilitar o desenvolvimento do projeto. Além disso, foram baixados os arquivos de teste disponíveis no 'Moodle' para auxiliar no trabalho.

1.2 - Estrutura de Classes: O código foi dividido em três principais classes: '*Normalize'*, responsável por realizar a normalização de palavras; '*ReadFiles'*, responsável por ler os arquivos e construir o índice invertido; e '*Search'*, responsável por realizar consultas com base em palavras-chave. Essa estrutura de classes permitiu uma organização adequada do projeto.

2. Implementação Detalhada

2.1 - Leitura de Arquivos: A primeira etapa desenvolvida foi a leitura de arquivos. Para isso, foi criada a classe '*ReadFile'*. O método '*readFromFolder'* recebe o caminho da pasta a ser explorada e retorna um mapa aninhado contendo as informações de frequência de cada palavra nos diferentes arquivos. A função inicia abrindo a pasta especificada pelo parâmetro *'folder'* usando a função opendir da biblioteca <*dirent.h*>. Em seguida, itera sobre cada arquivo encontrado na pasta utilizando a função '*readdir'* para ler seus respectivos nomes. Para cada arquivo encontrado, a função cria um fluxo de arquivo '*ifstream'* para abri-lo e ler seu conteúdo. Se o arquivo for aberto corretamente, cada linha é lida utilizando '*std::getline'* e concatenada na '*string content'*. Em seguida, o conteúdo do arquivo é normalizado utilizando um objeto '*Normalize'* e a função '*normalizeContent'*, que remove números e caracteres especiais do texto, armazenando o resultado na string '*normalizedcontent'*. A função '*normalizeContent'* recebe o conteúdo bruto do arquivo e percorre cada caractere, verificando se é uma letra de A a Z por meio da função booleana '*std::isalpha'*. Se for uma letra, ela é concatenada na '*string word'* em letras minúsculas. No final, a função retorna a string '*n_content'*, que armazena todas as palavras normalizadas separadas por espaços. Em seguida, o conteúdo normalizado é rapidamente dividido em palavras utilizando a função '*std::istringstream'* da biblioteca '*<sstream>'*, em um loop para cada palavra. Se o mapa *frequency* estiver vazio, a palavra é armazenada juntamente com o arquivo em que está e o valor 1. Caso contrário, itera-se sobre o mapa e verifica-se se a palavra já está presente: se estiver, incrementa-se o valor correspondente para o arquivo atual; se não estiver, insere-se um novo par chave-valor no mapa. Ao final da iteração, o arquivo é fechado e o processo continua para o próximo arquivo encontrado na pasta.

2.2 - Busca de Palavras-Chave: Com o mapa aninhado armazenando o índice invertido de cada palavra, a implementação da função de busca tornou-se mais fácil. Utilizando a classe '*Search'*, é possível ler as palavras enviadas pelo usuário por meio do comando *getline*, que armazena as palavras de uma linha na '*string line'*. Um objeto da classe Normalize é instanciado para rapidamente normalizar a linha recebida como entrada. Após isso,, um objeto '*istringstream'* é utilizado para ler a linha como uma sequência de palavras separadas, permitindo armazená-las no atributo privado *words* do tipo '*std::vector'*. Com o vetor de palavras normalizadas, o método '*returnFiles'* recebe esse vetor e cria um mapa chamado '*files'* para armazenar as informações relevantes de cada arquivo. A chave do mapa é o nome do arquivo, e o valor é uma estrutura chamada '*File'*, que armazena o nome do arquivo, o número de ocorrências desse arquivo e a soma de seus hits em todas as palavras-chave da busca. Iterando sobre as palavras do vetor words, o método percorre o mapa '*frequency'*, recebido como parâmetro, para encontrar as ocorrências dessa palavra. Para cada ocorrência, as informações relevantes são atualizadas no mapa '*files'*, incrementando-se 'hits' (contagem de ocorrências), 'freq' (contagem de palavras encontradas no arquivo) e mantendo o nome do arquivo correspondente. Após essa iteração, o método percorre novamente o mapa criado e verifica se a contagem de palavras no arquivo (freq) é igual ao número total de palavras na busca, armazenado na variável '*len'* (tamanho do vetor '*words'*). Se for o caso, o arquivo é considerado um arquivo completo e suas informações são adicionadas no mapa '*fullFiles'*, onde a chave é o nome do arquivo e o valor é o número de

hits. Por fim, o método itera sobre o mapa ‘*fullFiles’* e copia as informações para um vetor ordenado utilizando ‘*std::sort’* e a função booleana ‘*comparePairs’* como critério de ordenação. Assim, o atributo privado ‘*sortedVector’*, que contém pares de *strings* e inteiros, ordena o mapa de forma decrescente com base no número de hits. Em caso de empate, as chaves do mapa são comparadas para decidir qual é lexicograficamente maior.

2.3 - A função main começa com várias inclusões de bibliotecas, que fornecem recursos adicionais para o programa. As bibliotecas incluídas são:

- <*iostream*>: fornece as funcionalidades básicas de entrada e saída, como std::cout e std::cin.
- <*string*>: fornece a classe std::string e suas funções associadas para manipulação de strings.
- <*map*>: fornece a classe std::map, que é uma estrutura de dados associativa que mapeia chaves para valores.
- <*sstream*>: fornece a classe std::stringstream, que permite a manipulação de strings como fluxos de entrada e saída.

Ela também inclui os arquivos de cabeçalho *“readfile.hpp"*, *"normalize.hpp"* e *"search.hpp"*.

Em seguida, há definições de macros para cores de texto, como RESET, RED, GREEN, YELLOW e BLUE. Essas macros são usadas posteriormente para formatar a saída de texto no console. Uma variável chamada ‘*folder’* é declarada para representar o diretório no qual os arquivos serão lidos. Um objeto da classe ‘*ReadFile’* é criado e atribuído à variável read. Em seguida, o método ‘*readFromFolder’* do objeto read é chamado, passando o diretório folder como argumento. Esse método retorna um objeto ‘std::map<std::string, std::map<std::string, int>>’ que contém informações sobre a frequência de palavras nos arquivos lidos. Um objeto da classe ‘*Search’* é criado e atribuído à variável *‘s’*. Uma mensagem é impressa em azul no console, solicitando que o usuário digite as palavras que deseja buscar. Em seguida, o método ‘*readWords’* do objeto é chamado, permitindo que o usuário insira as palavras de busca. O método ‘*returnFiles’* do objeto ‘*s’* é chamado, passando o objeto ‘*frequency’* como argumento. Esse método retorna um vetor de pares ‘std::vector<std::pair<std::string, int>>’ que contém os arquivos correspondentes às palavras de busca, juntamente com o número de ocorrências de cada palavra nos arquivos. Os resultados da busca são impressos no console. O laço ‘*for’* itera sobre cada par de arquivo e número de ocorrências em ‘*orderedVec’*. Se o número de ocorrências do arquivo atual for igual ao número de ocorrências do primeiro arquivo em ‘*orderedVec’*, o texto é impresso em verde usando a macro GREEN. Caso contrário, o texto é impresso em amarelo usando a macro YELLOW. A função main possui blocos ‘*catch’* para capturar e lidar com exceções lançadas durante a execução do programa. As exceções capturadas são ‘*dirNotOpenException’*, ‘*fileNotOpenException’*, ‘*searchNotFoundException’* e ‘*std::exception’*. Se qualquer uma dessas exceções for lançada, uma mensagem de erro é impressa em vermelho no console. Uma linha de asteriscos é impressa no console para indicar o final do programa. A função main retorna 0 para indicar que o programa foi executado com sucesso.

2.4 - O Makefile é usado para compilar os arquivos fonte, gerar os arquivos objeto e, por fim, criar o arquivo executável. Além disso, também possui alvos para executar o programa e limpar os arquivos gerados durante a compilação. Essas linhas definem variáveis que serão utilizadas ao longo do Makefile:

- *CC = g++*: A variável *CC* é definida como *g++*, que é o compilador *C++* utilizado para compilar o código-fonte.
- *CFLAGS = -std=c++11 -Wall*: A variável *CFLAG*S é definida com as opções de compilação. *-std=c++11* define o padrão de linguagem C++ para C++11, enquanto *-Wall* habilita avisos detalhados do compilador.
- *TARGET = main.exe:* A variável *TARGET* é definida como main.exe, que será o nome do arquivo executável gerado.
- *BUILD = ./build:* A variável *BUILD* é definida como ./build, que representa o diretório onde os arquivos objeto serão gerados.
- *SRC = ./src:* A variável *SRC* é definida como ./src, que representa o diretório onde os arquivos de código-fonte estão localizados.
- *INCLUDE = ./include:* A variável *INCLUDE* é definida como *./include*, que representa o diretório onde os arquivos de cabeçalho estão localizados.

A primeira regra de compilação é a responsável por gerar o arquivo executável:

- *${TARGET}:* É a dependência alvo desta regra, ou seja, o arquivo executável *main.exe.*
- *${BUILD}/normalize.o ${BUILD}/readfile.o ${BUILD}/search.o ${BUILD}/main.o:* São as dependências desta regra, que são os arquivos objeto gerados a partir dos arquivos de código-fonte.
- *${CC} ${CFLAGS} -o ${TARGET} ${BUILD}/*.o:* É o comando que será executado para gerar o arquivo executável. O CC indica o compilador a ser utilizado, *CFLAGS* são as opções de compilação, -o especifica o arquivo de saída, *${TARGET}* é o nome do arquivo executável e *${BUILD}/*.o são os arquivos objeto que serão vinculados.

A segunda regra de compilação é a responsável por gerar o arquivo ‘*normalize.o’:*

- *${BUILD}/normalize.o*: É a dependência alvo desta regra, ou seja, o arquivo objeto *normalize.o*.
- *${INCLUDE}/normalize.hpp ${SRC}/normalize.cpp*: São as dependências desta regra, que são o arquivo de cabeçalho normalize.hpp e o arquivo de código-fonte *normalize.cpp*.
- *${CC} ${CFLAGS} -I ${INCLUDE}/ -c ${SRC}/normalize.cpp -o ${BUILD}/normalize.o:* É o comando que será executado para gerar o arquivo objeto. -I é usado para especificar o diretório de inclusão para os arquivos de cabeçalho, -c indica que apenas a compilação deve ser feita (sem vinculação), *${SRC}/normalize.cpp* é o arquivo de código-fonte a ser compilado e *${BUILD}/normalize.o* é o arquivo objeto de saída.

As outras regras de compilação para os arquivos readfile.o, search.o e main.o seguem uma estrutura semelhante. A regra '*run'* é um alvo especial que permite executar o programa de maneira conveniente.

- *clear*: É o comando para limpar o console antes de executar o programa.
- *./${TARGET}*: É o comando para executar o arquivo executável *${TARGET}*.

A regra '*clean'* é usada para limpar o diretório de compilação.

- *rm -f ${BUILD}/**: É o comando para remover todos os arquivos presentes no diretório *${BUILD}*.
- *rm -f ${TARGET}*: É o comando para remover o arquivo executável *${TARGET}*.

```
. programa
├─ Makefile
├─ build/
|   └─ [main.o]
|   └─ [normalize.o]
|   └─ [readfile.o]
|   └─ [search.o]
├─ include/
|   └─ normalize.hpp
|   └─ readfile.hpp
|   └─ search.hpp
├─ src/
|   └─ main.cpp
|   └─ normalize.cpp
|   └─ readfile.cpp
|   └─ search.cpp
```

Hierarquia de diretórios

3. Conclusão

O desenvolvimento do projeto teve duração de aproximadamente uma semana. Nesse tempo, foi necessário aprender melhor a mexer no GitHub e a trabalhar em equipe. A maior dificuldade do código certamente foi a função de normalização das palavras de entrada e a função de busca como um todo. No final, tentamos deixar nosso código mais limpo possível, encapsulando o necessário e fortalecendo a coesão e enfraquecendo o acoplamento. Comentários foram adicionados para facilitar a leitura do código, e exceções foram inseridas para tratar os erros da forma desejada. Consultas em diversos sites foram feitas, principalmente quando se tratava de funções da biblioteca padronizada STL. No geral, cada integrante não só conseguiu colocar em prática o que foi ensinado na sala, mas também aprender funções e bibliotecas adicionais, fortalecendo suas habilidades em programação.