UNIVERSIDADE FEDERAL DE MINAS GERAIS

PROGRAMAÇÃO E DESENVOLVIMENTO DE SOFTWARE II

TRABALHO PRÁTICO

Máquina de Busca

Eduardo Renan da Silva Gomes

Samuel Fischer de Oliveira Lage

Belo Horizonte, 2018

1. INTRODUÇÃO

Este trabalho tem como objetivo a implementação de um índice invertido, que é basicamente uma estrutura onde uma palavra é correlacionada a uma lista de documentos que a contém. Dada uma quantidade qualquer de documentos, o nosso algoritmo deve ter a capacidade de realizar a leitura de todas as palavras existentes dentro desses documentos e criar um índice onde cada uma dessas palavras apresentará uma lista de documentos nos quais está contida. Para realizar essa tarefa utilizamos a maioria dos conhecimentos adquiridos durante o semestre de curso da matéria Programação e Desenvolvimento de Software II, dentre estes podemos destacar a implementação de TAD, classes, estruturas de dados, testes de unidade e tratamento de exceções. É importante ressaltar que a IDE usada para o desenvolvimento do projeto foi o Microsoft Visual Studio 2017.

2. IMPLEMENTAÇÃO

Para realizarmos a implementação, utilizamos a estrutura de um TAD. Criamos um projeto onde temos arquivos *.cpp* e *.h* que se interconectam e nos possibilitam a divisão e organização de todos os componentes necessários para a implementação geral do código. Iremos explicar como cada um desses arquivos funcionam e as relações de interdependência que cada um destes tem um com o outro. Interessante observar que a ordem de explicação de cada um desses arquivos na documentação será feita de forma a contemplar as relações de interdependência, já citadas, de acordo com a configuração do projeto.

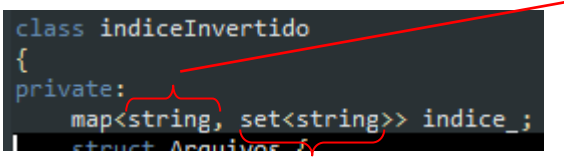
a. *indiceInvertido.h*

Este arquivo é um arquivo do tipo *header* onde realizamos a definição da classe *indiceInvertido*, com seus membros privados e públicos. Essa classe foi construída exatamente com o objetivo de contemplar todas as funcionalidades e abstrações necessárias para que o índice invertido fosse implementado de acordo com as orientações do professor.

- Membros da parte Private da classe:

  - Estrutura do índice invertido:

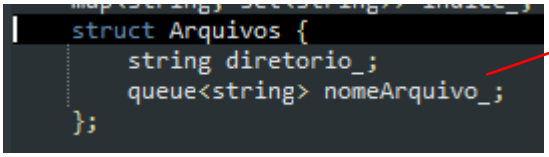
    Essa é a parte da classe onde foi concebida a construção das entidades que compõem a nossa classe. Dentre elas temos a variável *índice_* do tipo *map* (dicionário), que é uma classe da biblioteca básica do C++. Essa variável foi construída utilizando a tecnologia de templates, a qual que nos possibilitou criá-la com a abstração sugerida pelo professor, como indicado abaixo.



Termo: *String* utilizada para armazenar uma palavra presente em algum dos documentos que serão lidos durante a execução do programa. Utilizado como "chave" do *map* (dicionário).

Incidência: Tipo *set* de *string's (presente na* biblioteca básica de C++) usado para armazenar a lista de documentos nos quais o termo (chave) está presente. Representa o componente "valor" da abstração da

Além disso, também criamos uma estrutura chamada *Arquivos* que é uma estratégia para podermos armazenar o diretório e nome dos arquivos contidos nele para podermos utilizar em outras funções.



Variável do tipo *queue*, que é uma classe (biblioteca básica de C++) própria para ser implementada para criação de filas. No nosso caso, então, criamos uma fila de strings pela utilização da tecnologia de template disponível na classe.

- Funções:

  atualizaTermo : Função que recebe duas strings, sendo elas de nome *termo* e *documento*, e atualiza ou adiciona a incidência do termo no índice invertido. Sua execução confere se o termo recebido já existe dentro do índice. Caso o termo já exista, a função atualiza a incidência do mesmo, adicionando o documento em questão. Caso o termo não exista, a função adiciona o termo ao índice juntamente com o documento em questão.

  selecionaDir : Função que recebe um diretório (string) e retorna uma estrutura do tipo *Arquivos* com o diretório buscado e os arquivos que estão presentes dentro do mesmo. Seu funcionamento baseia-se, pricipalmente, na utilização de ponteiros do tipo *DIR* e *dirent* que foram importados da biblioteca *dirent.h*. Um laço while testa se existem arquivos dentro de um diretório, apontando-os um a um e, no caso de existir, esses arquivos são adicionados à fila *nomeArquivo_* da estrutura.

  checaTipoTXT : Função criada para testar se a extensão do arquivo é do tipo ".txt". Ela faz isso checando se o arquivo tem pelo menos 5 caracteres e analisando os últimos 4 caracteres do nome do arquivo, checando se eles são ".txt".

❖ Membros da parte Public da classe:

- Funções:

  reinicializa : Função que realiza a limpeza do índice. Ela faz isso utilizando a função *clear* presente na biblioteca da classe *map*. Além disso, também é realizado um teste para saber se a limpeza foi feita com sucesso.

  atualizaIndice : Função responsável por atualizar o índice invertido de uma forma mais geral. É uma das funções mais importantes de todo o projeto, pois ela é chamada na função *main* do nosso projeto para a atualização do índice a partir da inserção de um diretório pelo usuário. Primeiro ela recebe o diretório (tipo string) e chama a função *selecionaDir* para criar uma variável *filaArquivos* do tipo *Arquivos* e, a medida em que a função lê os termos presentes nos arquivos, um por um, a função *atualizaTermo* é chamada para poder inserir um novo termo no índice ou atualizar as incidências de um já existente. Essa leitura de arquivo por arquivo é realizada utilizando funções como *empty* (responsável por checar se a fila de arquivos não está vazia), *front* (retorna o primeiro arquivo presente na fila de arquivos) e *pop* (remove o arquivo já lido da lista de arquivos), sendo todas essas pertencentes à classe *queue* incluída no arquivo. Toda essa leitura de arquivos e de termos presentes dentro deles é feita enquanto não existirem mais arquivos na fila de arquivos da *filaArquivos*. É importante realizar a observação que dentro dessa função foi feito o tratamento de string requerido pelo professor na parte de identificar os termos presentes em cada arquivo de leitura, que é ler e armazenar todos os termos em letras minúsculas, detectando, substituindo e removendo acentos, letras

maiúsculas, espaços e outras coisas desnecessárias de acordo com a especificação.

incidencia : Função que retorna um *set* de *string's* com as incidências do termo recebido pela mesma. Ela faz isso utilizando um *iterator* do tipo *map* que recebe o valor de retorno da função *find* da classe *map (endereço de memória do termo, se achado)*. A partir do momento que o *iterator* recebe esse valor, é conferido se o termo em questão existe (se não existir é retornado o endereço de memória do fim do índice). Caso o termo exista, a função adiciona o arquivo ao qual o termo é pertencente ao *set* de *string's* que contém as incidências do termo, utilizando a função *insert* da classe *map*.

vocabulário : Função que retorna um *set* de *string's* com todos os termos presentes no índice. São utilizadas estratégias de implementação bem parecidas com a função *incidencia* para poder fazer isso.

- Funções para testes unitários:

atualizaTermoTest : Essa função é responsável por fazer o teste de unidade da função *atualizaTermo*. Ela faz isso criando uma variável *teste* do tipo da classe *indiceInvertido*, adicionando uma chave e um valor qualquer á essa variável. Depois ela faz a checagem se a chave e valor realmente foram inseridos, insere um novo valor à chave e checa se esse novo valor inserido foi atualizado.

selecionaDirTest : Função responsável por fazer o teste de unidade da função *selecionaDir*. Ela faz isso tentando abrir um arquivo, que ela mesma cria, no diretório *C:\ProgramData*\ da máquina. Caso ela não consiga, ela retorna falso. Importante lembrar que para que essa função funcione, é necessário que a máquina do usuário tenha um diretório *C:* e que um cliente *Windows* esteja instalado nesta partição (de modo que o diretório C:\ProgramData\ exista).

reinicializaTest : Função responsável por fazer o teste de unidade da função reinicializa. Ela faz isso criando uma variável test do tipo da classe *indiceInvertido*. Ela tenta fazer a limpeza do índice de teste após adicionar algo ao mesmo. Caso a limpeza tenha sucesso, a função retorna valor verdadeiro.

incidenciaTest : Função responsável por fazer o teste de unidade da função *incidencia*. Ela cria uma fila de arquivos de teste associada a um índice de teste, adiciona dois termos ao índice relacionados a essa fila e realiza então uma lógica para averiguar a equalidade entre a fila conhecida e a retornada pela função.

vocabularioTest : Função responsável por fazer o teste de unidade da função *vocabulario*. Assim como na *incidenciaTest* ela cria índice e fila de arquivos de teste, adiciona elementos à essa fila e ao índice e procura se o vocabulário retornado é igual ao conhecido. Caso não forem, a função retorna valor falso

checaTipoTXTTest : Função responsável por fazer o teste de unidade da função *checaTipoTXT*. Ela passa um nome correto de arquivo com

extensão ".txt" para a função *checaTipoTXT*. Caso a função *checaTipoTXT* retorne falso a função de teste retorna erro.

b. *indiceInvertido.cpp*

Arquivo que contém a implementação de todas as funções presentes no arquivo *indiceInvertido.h*. É nele onde o arquivo *indiceInvertido.h* puxa o código de todas as suas funções. Não é necessário comentar uma ou mais funções contidas nesse arquivo, devido ao fato de que isso já foi feito durante a descrição do arquivo *indiceInvertido.h*.

c. *utilidades.h*

Arquivo que contém as declarações dos nomes das funções utilizadas na função *main* do arquivo *MaquinaDeBusca.cpp* para rodar testes unitários de uma forma prática, para imprimir um índice invertido e para imprimir as incidências de um termo.

d. *utilidades.cpp*

Arquivo que contém o código escrito das funções presentes no arquivo *utilidades.h*. Cada uma delas será descrita e explicada abaixo.

- Funções:

  testeGeral : Função que executa todos os testes unitários de uma vez só. Permite que o desenvolvedor possa visualizar se o programa que ele vai utilizar apresenta algum erro.

  imprimeIndice : Função que recebe o índice e imprime todos os seus termos seguidos de suas incidências de forma ordenada e organizada.

  imprimeIncidencia : Função que recebe um *set* de *string's* com as incidências e as imprime na tela do usuário. Faz isso utilizando membros e funções da classe *set*.

e. *MaquinaDeBusca.cpp*

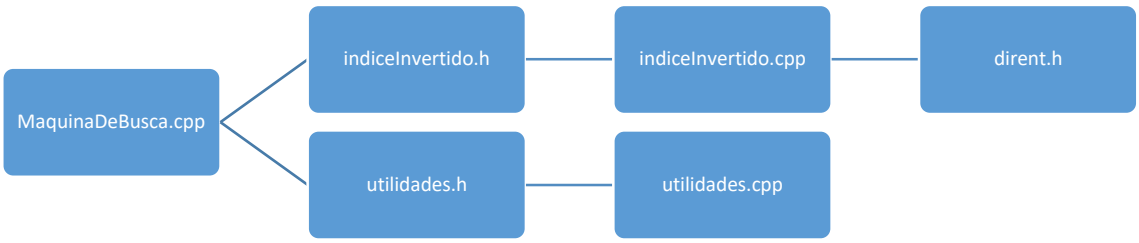
Este é o arquivo que contém a função *main* do nosso projeto. Portanto é nele onde o usuário realiza a real interface com o programa. Sua função *main* tem algumas particularidades que serão mostradas abaixo.

- Funções:

  main : Função que chama todas as outras funções do TAD implementado. Ambiente onde o usuário realmente interage e utiliza das ferramentas do programa desenvolvido. Algumas de suas particularidades tem de ser notadas. Primeiramente nos deparamos com a execução da função *testeGeral*, para realizar todos os testes de unidade antes da execução do programa. Depois temos a parte em que o programa pede para que o usuário entre com o diretório para que a função *atualizaIndice* seja chamada posteriormente, leia os arquivos desse diretório e atualize o índice invertido. Temos outra parte, onde o

programa proporciona a opção para que o usuário possa realizar a impressão de todo o índice na tela. E, finalmente, temos a parte onde o usuário pode digitar um termo específico para que o programa retorne a lista de incidências de arquivos nos quais o termo em questão está presente, dando a opção de realizar outras consultas antes da finalização da execução do programa. É importante lembrar que componentes da parte de tratamento de exceções estão presentes nessa parte e estes serão explicados logo á frente na nossa documentação.

Agora, depois de descrever todos os arquivos e seus conteúdos, vemos que é interessante mostrar a forma como eles se interconectam. Abaixo temos um descritivo de como vemos a estrutura de arquitetura e organização desses arquivos.



Como indicado acima, podemos visualizar a interdependência dos arquivos de projeto. Onde temos o arquivo *MaquinaDeBusca.cpp* que importa bibliotecas dos arquivos *indiceInvertido.h* e *utilidades.h*. Em sequência temos o arquivo *indiceInvertido.h* que importa todos os códigos de funções do arquivo *indiceInvertido.cpp*. Temos o arquivo *indiceInvertido.cpp* que está conectado ao arquivo *dirent.h*, pois importa várias estruturas dele. Por outro lado, temos o arquivo *utilidades.h* que importa todos os códigos das suas funções que estão contidos no arquivo *utilidades.cpp*. Podemos notar que existe uma relação de interdependência entre os arquivos que ora é obrigatória e ora não, dependendo de qual função, classe, estrutura que estamos utilizando no programa durante o processo de compilação.

Agora, explica-se como foram concebidos os testes de exceção realizados durante a implementação do código. Fizemos da forma como foi ensinado na aula sobre assunto ministrada durante o nosso curso, utilizando o método que usa os comandos *try* e *catch*. Abaixo mostraremos a tabela de códigos de erro (do tipo int) lançados nos tratamentos de exceção.

- Codificação dos testes de exceção:

  1 = Índice não foi limpo corretamente.

  2 = Diretório não está no padrão (string com tamanho menor que 2)

  3 = Diretório não pode ser aberto

  4 = Diretório aberto está vazio

  5 = Arquivo não foi aberto

6 = Arquivo não foi fechado

7 = Termo não existe no índice

8 = O índice está vazio e impossibilita a execução da *vocabulario*

10 = O índice está vazio e não existem termos

Com a aplicação desses tratamentos de exceção no nosso projeto podemos realizar uma ótima utilização desses códigos, tanto para comunicar ao usuário algum tipo de ocorrência indevida durante a execução ou para realizar o bloqueio ou interrupção de alguma parte do código durante a execução.

3. OBSERVAÇÕES

É necessário observar que todo o projeto foi feito utilizando a IDE Microsoft Visual Studio 2017 v15.9.1 (Conjunto de Ferramentas Visual Studio v141 e SDK do Windows 10.0.17134.0), desenvolvida utilizando o SO Windows 10.

Para fins de registro, tentamos migrar para o CodeBlocks nas etapas finais do desenvolvimento com o objetivo de disponibilizar o projeto no compilador GCC, porém não obtivemos sucesso. Ao passar o código do Visual Studio ao CodeBlocks, foram encontrados vários erros do tipo de caracteres que estavam sendo utilizados nas linhas de código (o CodeBlocks apenas aceita linhas de código escritas estritamente com caracteres ASCII). Após recriar todo o projeto e fazer toda a conversão do código para ASCII, encontramos um problema de que o código não funcionava da mesma maneira, ocorrendo erros provavelmente no tratamento de exceções (análise feita através da observação da depuração do projeto).

Conversamos com o corretor do TP (Marcos) e ele nos disse que não existe problema em disponibilizar o projeto da forma como o desenvolvemos, em Visual Studio desde que a descrição acima feita fosse documentada.

4. CONCLUSÃO

Com o término da execução e implementação desse projeto é possível entender que os conceitos adquiridos durante o decorrer da disciplina foram todos colocados em prática. Nota-se que diversas coisas ditas e reforçadas pelo professor se tornaram muito mais importantes durante o processo de desenvolvimento desse trabalho. A medida que ele se tornava mais e mais complexo, viu-se que as estratégias de organização do programa foram cruciais para que o projeto fosse bem executado. A implementação do TAD com a separação de bibliotecas específicas usando arquivos *.c* e *.h,* fez com que o projeto se organizasse de uma forma muito mais dinâmica e técnica. Poder manipular esses arquivos separadamente e abstraí-los como bibliotecas específicas para utilizações específicas se mostrou algo muito útil. Utilizar classes e obedecer os critérios de abstração também foi de muita utilidade, principalmente no quesito de reutilização de funções. A parte de testes unitários foi uma das maiores surpresas, pois, assim que estavam prontos, proporcionaram imensa agilidade no desenvolvimento final do projeto. E, por fim, os tratamentos de exceção, bem parecidos com os de unidade nesse aspecto, também garantiram muito mais avanço rápido no desenvolvimento do projeto e proporcionam conforto para a entrega do trabalho. Assim, conclui-se que esse trabalho foi de grande valia para a conclusão do nosso curso de programação e desenvolvimento de software.