Programação e Desenvolvimento de Software
Trabalho Prático
Máquina de Busca

Emerson Alves da Silva
Jean Santos Costa

I. Introdução

O presente trabalho descreve a implementação de uma *Máquina de Busca* utilizando um índice invertido.

A partir de uma base de dados formada por arquivos de texto, o software deve ser capaz de armazenar todas as palavras presentes nos mesmos e indicar em quais arquivos cada palavra pode ser encontrada. Além disso, o software deve permitir que o usuário pesquise por uma palavra qualquer e obtenha o nome dos arquivos que contém essa chave de busca.

A figura 1 mostra o diagrama de casos de uso relacionado ao software com os dois *stakeholders* envolvidos.

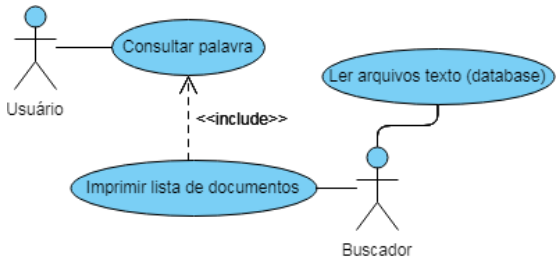

Figura 1: Diagrama de casos de uso.

O índice invertido implementado utiliza como chave apenas palavras com letras minúsculas e caracteres alfa-numéricos, ou seja, cada palavra lida da coleção de documentos foi convertida antes de ser adicionada ao índice.

II. Implementação

A. Estruturas Map e Set

Foram utilizadas os contêineres associativos *map* e *set* da *Standard Template Library (STL)* do C++, tal como sugerido na descrição do trabalho. A *STL* define componentes reutilizáveis baseados em *template* que implementam estruturas de dados e algoritmos comumente utilizados por programadores. Reforçando assim o conceito de reutilização de código.

As estruturas de dados *map* e *set* são contêineres associativos que permitem acesso direto aos seus elementos via *chaves de pesquisa*, tanto para armazenamento quanto para recuperação. Um *map* armazena um conjunto de *pares chave/valor*, enquanto que um *set* armazena um conjunto de *valores*. Ambos não permitem o armazenamento de elementos repetidos (*chaves* no caso do *map* e *valores* no caso do *set*) e os armazena de forma ordenada. Caso haja a tentativa de se inserir um elemento repetido no conjunto, ele será ignorado. O armazenamento é feito ordenando-se as *chaves* ou os *valores* do menor para o maior (*default*).

A seção II-C mostra como esses contêineres foram utilizados como membros de dados da classe *Indice Invertido*.

B. Leitura de arquivos

Essa parte consistiu na leitura do conteúdo dos arquivos em `.txt` e do processamento das palavras. Esse processamento foi responsável pela remoção de todos os caracteres que não fossem letras (a-z e A-Z) e números (0-9), além de transformar todas as letras maiúsculas em minúsculas.

Como essa parte não tem relação direta com a estrutura de dados do índice invertido, ela foi implementada separadamente da classe principal, nos arquivos `Useful.h` e `Useful.cpp`. No arquivo `Useful.h` foram colocados os protótipos das funções de leitura e processamento, com comentários sobre suas funcionalidades, e no arquivo `Useful.cpp` colocou-se a implementação dessas funções.

As três funções implementadas nesses arquivos para executar a leitura e o processamento dos caracteres foram: *CaractereIndesejado*, *AjustaString* e *LeArquivo*.

A função *CaractereIndesejado* recebe como parâmetro um único caractere, verifica se ele é uma letra (maiúscula ou minúscula) ou um número, utilizando a função *isalnum*. A função retorna *false* caso ele seja uma letra ou número e *true*, caso não seja. Indicando assim se ele é ou não um caractere que deve ser removido.

Já a função *AjustaString*, é responsável por remover os caracteres indesejados e transformar as letras de maiúsculas para minúsculas. Ela recebe como parâmetro de entrada uma palavra do tipo *string* por referência, e não retorna nada. Ela utiliza as funções *erase*, que é uma função membro definida para a classe *string* do C++ e a função *remove_if*, para remover os caracteres indesejados e a função *tolower* para transformar as letras em minúsculas. Devido a passagem por referência, as modificações realizadas foram feitas na palavra original, e não em uma cópia dela. Assim, após o término da função, a palavra estará devidamente ajustada para ser incluída no índice.

Por fim, a função *LeArquivo* é responsável por fazer a leitura do arquivo `.txt`, um de cada vez, e chamar a função *inserir* da classe índice invertido (explicada na seção II-C), a partir de um objeto dessa classe, passado como parâmetro (por referência constante). Assim, a função recebe dois parâmetros, uma *string* com o nome completo do arquivo e sua extensão, e um objeto da classe índice invertido. A função não retorna nada. Antes de chamar a função *inserir*, a função *AjustaString* é executada. Após isso, cria-se um par *palavra/arquivo*, com a palavra ajustada e o nome do arquivo do qual ela foi lida, que

é então passado como parâmetro de entrada para a função *inserir*. O diagrama de sequência da figura 2 exemplifica o processo de leitura de um arquivo para inserir no índice invertido um novo item.

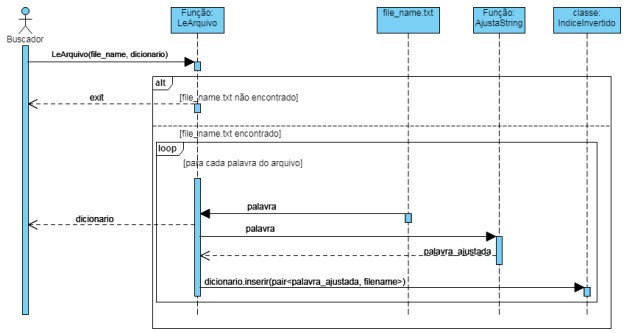

Figura 2: Diagrama de sequência.

C. Estrutura de dados do IndiceInvertido

A estrutura índice invertido foi implementada como um classe, chamada *IndiceInvertido*. Ela possui como variável membro um *map* de um *par palavra/documentos*, no qual a palavra é do tipo *string* e os documentos representam um *set* de *strings*, associados à essa palavra (*map< string, set<string> >*). Ela possui ainda um construtor que inicializa um índice vazio e as funções membro: *inserir*, *remover*, *vazio*, *tamanho*, *pertence* e *buscar*. Tal como pode ser visto no diagrama de classe UML na Figura 3.

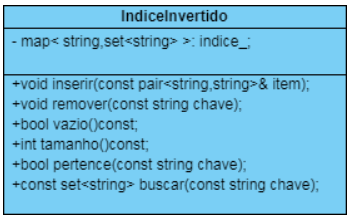

Figura 3: Diagrama da classe índice invertido.

A interface da classe, mostrada no diagrama de classe da Figura 3, foi implementada no arquivo `IndiceInvertido.h`. Sua implementação foi feita no arquivo `IndiceInvertido.cpp`.

A função *inserir* é responsável pela inserção de uma nova palavra e dos nomes dos arquivos associados à ela. Ela recebe como parâmetro, por referência constante, um par de *strings chave/arquivo*. Antes de inserir o novo par no índice, ela verifica se a palavra que estamos tentando inserir já pertence ao índice, por meio da função *pertence* da classe. Caso a palavra seja nova (ainda não pertença ao índice), um novo conjunto (*set*) é criado para armazenar os nomes dos arquivos associados a essa palavra. O nome do arquivo é então inserido nesse conjunto, por meio da função *insert*, função membro do contêiner *set*. Em seguida um par *palavra/conjunto de arquivos* é criado e passado como parâmetro para a função membro *insert* do contêiner *map*, realizando assim a inserção de uma nova palavra e dos arquivos associados a ela no índice. Caso contrário, se a palavra que estamos tentando inserir já estiver no índice, a função tentará apenas inserir no conjunto de arquivos associado à aquela palavra o nome do arquivo recebido. Caso o nome do arquivo também já faça parte do conjunto de arquivos dessa palavra, sua inserção será ignorada, pois o conjunto *(set)* não aceita elementos repetidos.

A função *remover* realiza a remoção de um par *palavra/conjunto de arquivos*. Para isso, ela recebe como parâmetro de entrada uma palavra (chave) do tipo *string*. Caso essa palavra pertença ao índice (indicada pela função *pertence* da classe), um iterador do tipo *map< string, set<string> >* (mesmo tipo de *indice_*) é declarado e inicializado com o valor de retorno da função membro *find* do contêiner *map*. Essa função recebe como parâmetro a chave (palavra do tipo *string*) que estamos buscando e retorna um iterador (uma referência do mesmo tipo do objeto que a chamou) para ela no índice. Essa referência é então passada para a função membro *erase* do contêiner *map*, que removerá a chave e o conjunto de valores associados a ela do índice. Caso a palavra não faça parte do índice, a função remover não é executada.

A função constante *vazio* não recebe parâmetros e retorna uma variável do tipo *bool* que indica se o índice está vazio (*true*) ou o não (*false*). Para tanto, ela utiliza a função membro *empty* definida para o contêiner *map*.

A função *tamanho* também não recebe nenhum parâmetro. Ela retorna uma variável do tipo *int* que representa o número de chaves armazenadas no índice. Utilizando a função membro *size* do contêiner *map*.

Já a função *pertence*, é responsável verificar se uma palavra, passada como parâmetro, pertence ao índice. Caso ela seja encontrada no índice, a função retorna o valor *true*, caso contrário retorna *false*. Ela percorre o índice invertido declarando um iterador do mesmo tipo do índice e inicializando seu valor com o retorno da função *find*. Caso a referência retornada por *find* não for uma posição além do último elemento do índice (indicado pela função *end*), então a palavra foi encontrada no índice.

Por fim, a função *buscar* da classe é responsável por realizar a busca de uma palavra no índice e retornar o conjunto com os nomes dos arquivos associados à ela. Ela recebe como parâmetro a palavra que se deseja pesquisar, do tipo *string*. Assim como nas funções *pertence* e *remover*, utiliza a função *find* para retornar uma referência para a palavra no índice. Essa referência é então retornada pela função caso a palavra tenha sido encontrada. Caso ela não pertença ao índice, um conjunto de strings vazio é retornado.

III. Testes

Testes de unidade são testes aplicados à funções ou procedimentos individuais do software que buscam atestar o seu funcionamento isolado. São aplicados às menores partes testáveis do código. Eles simplificam a debugação e integração das partes, reduzindo as incertezas nas unidades testadas.

Inicialmente tentou-se utilizar um *framework* para realização dos testes de unidades, o *CUnit*, por meio da integração com a *IDE NETBeans*. Contudo, a instalação e integração do *framework* à *IDE* não funcionou como o esperado, e não houve tempo hábil para tentar solucionar

o problema. Assim, os testes foram feitos por meio dos métodos de uma classe de testes criada especificamente para testar a máquina de busca.

Essa classe foi chamada de *TestesDeUnidade* e sua interface foi implementada separada da classe principal do índice invertido, no arquivo `TestesDeUnidade.h`. Sua implementação foi feita no arquivo `TestesDeUnidade.cpp`.

A classe de testes contém funções que buscam testar isoladamente as principais funcionalidades da máquina de busca, são elas: *TestaInserir*, *TestaRemover*, *TestaBusca* e *TestaAjustaString*. As duas primeiras funções de teste, *TestaInserir* e *TestaRemover*, foram sobrecarregadas para testar tanto a inserção/remoção de uma palavra e seus arquivos associados, quanto a inserção/remoção de várias palavras e seus arquivos associados no índice.

Para que a classe de testes tivesse acesso aos dados *private* da classe *IndiceInvertido*, ou seja, tivesse acesso ao próprio índice, ela foi declarada como uma classe *friend*. Essa declaração foi feita no interior da definição da classe *IndiceInvertido* e por uma *forward declaration* de classe *TestesDeUnidade* antes do início da definição da classe *IndiceInvertido*.

A realização dos testes contou com a utilização de entradas simples. Para os testes da função *inserir*, foram utilizadas palavras já ajustadas (sem caracteres especiais e todas minúsculas), e nomes de arquivos do tipo `"teste2.txt"`. Para testar a função remover, utilizou-se um índice não vazio e uma palavra que pertencia a ele. Já o teste da função *AjustaString* recebeu uma palavra com caracteres especiais, além de letras e números, e letras maiúsculas, e foi testado se após a execução da função, a palavra retornada foi igual a sua versão esperada. Por fim, o teste da função de busca foi realizado tanto para o caso em que o índice não estava vazio e a palavra fazia parte dele, quanto para o caso do índice estar vazio, ou seja, a busca deveria retornar um conjunto vazio. Na Figura 4 são mostradas os resultados dos testes de unidade executados.

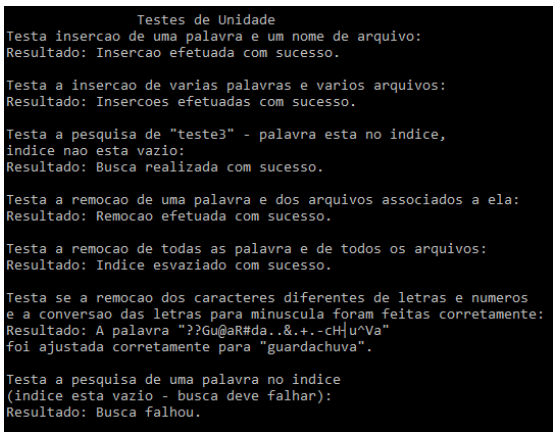

Figura 4: Resultado da execução dos testes de unidades para inserção, remoção, busca e ajuste de *strings*.

IV. Tratamento de exceções

A inexistência de um arquivo, a ser lido e incluído no índice, no diretório informado, foi tratada como uma exceção. Ou seja, caso a *string* contendo o nome completo da localização do arquivo apresente algum erro que impeça a sua leitura, o programa tratará essa situação como uma exceção.

Para isso, foi implementada uma classe de exceção chamada *ArquivoNaoEncontrado*. A interface e implementação dessa classe foi feita no arquivo `ArquivoNaoEncontrado.h`, com funções implementadas *inline*. Essa classe tem um construtor padrão que inicializa a variável *private mensagem_* como uma *string* vazia. O segundo construtor da classe inicializa essa variável com o conteúdo da *string* passada como parâmetro. Além desses construtores, há também a função *getMensagem* que retorna o conteúdo armazenado na variável *mensagem_* da classe.

Na função *LeArquivo*, implementada em `Useful.cpp`, após a abertura do arquivo, cujo nome e extensão foram passados como parâmetro, é testado se a variável do tipo *file input stream* que aponta para o arquivo aberto, não está apontando para *NULL* (posição inválida de memória). Caso não esteja, a execução da função continua normalmente. Caso essa variável esteja apontando para *NULL*, então uma exceção é lançada pelo comando *throw*. Isso é feito declarando-se e lançando um objeto da classe *ArquivoNaoEncontrado*. Nesse momento, a execução da função *LeArquivo* é interrompida e a execução volta para a função que a chamou, nesse caso a função *main*.

Na função *main*, o bloco *try*, no qual foi inserida essa função de leitura de arquivo, identifica que uma exceção foi lançada e interrompe a execução dos comandos que se seguem à chamada da função dentro desse bloco, passando imediatamente para o bloco *catch*. O bloco *catch*, inserido logo após o bloco *try*, identifica que uma execção do tipo *ArquivoNaoEncontrado* foi lançada e aplica as ações necessárias para correção do problema. Nesse caso, as ações tomadas foram a de avisar o usuário que o arquivo informado não foi encontrado e que por isso o programa seria encerrado. Na Figura 5 é mostrado um exemplo de funcionamento da classe de exceção.

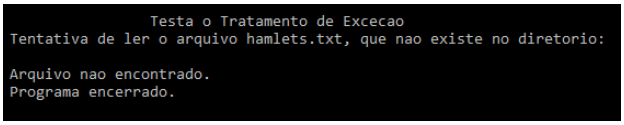

Figura 5: Tratamento de exceção: Arquivo não encontrado.

V. Resultados

Foram utilizados cinco arquivos como base de dados para o software de busca: *hamlet.txt*, *kinglear.txt*, *macbeth.txt*, *othello.txt* e *romeoandjuliet.txt*. As figuras 6, 7, 8, 9 e 10 mostram os resultados da busca por algumas palavras.

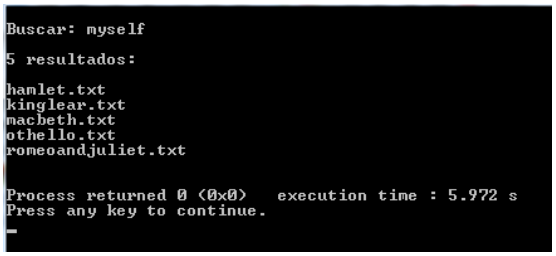

Figura 6: Resultado da busca pela palavra ”myself”, 5 resultados foram encontrados.

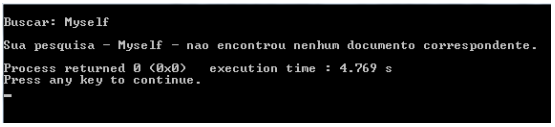

Figura 7: Teste de busca por palavra com letra maiúscula, nenhum resultado foi encontrado.

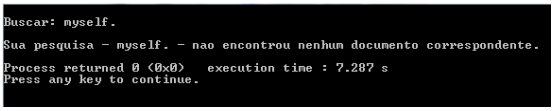

Figura 8: Teste de busca por palavra com caractere de pontuação, nenhum resultado foi encontrado.

Figura 9: Teste de busca com caractere numérico, dois resultados foram encontrados.

Figura 10: Resultado da busca pela palavra ”horse”, 4 resultados foram encontrados.

VI. Considerações finais

Nesse trabalho foi possível aplicar vários dos conceitos estudados na disciplina, tais como: controle de versão de código, *code review*, encapsulamento, reutilização de código e estilo de programação.

Para o controle de verão foi utilizada a plataforma *GitHub*. Seu uso proporcionou maior controle sobre as mudanças realizadas no código ao logo de seu desenvolvimento, bem como a revisão de código (*code review*) entre os membros do grupo.

O encapsulamento de informações foi aplicado separando-se a interface das classes e os protótipos das funções de suas respectivas implementações. Seguindo o conceito de que o usuário não deve e não precisa saber como uma função é executada, mas sim o que ela faz, o que recebe como entrada e o que retorna para o usuário ou o programa.

A reutilização de código foi exercitada por meio da utilização da biblioteca padrão do c++, a *Standard Template Library*. A qual forneceu estruturas suficientes para a implementação do índice invertido, além de funções auxiliares pelas quais foi possível armazenar, percorrer e remover elementos dessas estruturas.

Por fim, tentou-se seguir o guia de programação de estilos do *Google (Google Style Guide)*. Além de evitar comentários excessivos no código. Os comentários foram adicionados, em sua grande maioria, na interface das classes e nos protótipos das funções. Com o objetivo de explicar seu uso e necessidade no programa.