Documentação do trabalho prático de PDS2

Equipe:
Allan Dayrell Marcelino
Bruno Soares e Silva
Eduardo Henrique dos Santos Correia

Link do repositório: https://github.com/edu-correia/TrabalhoPratico-PDS2

Sumário

Introdução

Problema

O problema que nos foi passado para desenvolver a solução é uma “Máquina de busca”, similar ao Google, por exemplo. Diferentemente de mecanismos tradicionais de busca, o nosso mecanismo de busca não precisa entrar na internet e pesquisar por páginas ou percorrer todos arquivos de diversas pastas, apenas percorrer os arquivos da pasta “documentos”. A nossa interface de busca é apenas um programa em execução na linha de comando, onde se é digitado a busca e é retornado todos arquivos que contém todas palavras da busca e ordenados pela quantidade de aparições das palavras nos arquivos.

Solução

Na nossa solução, haverá uma divisão em 3 subsistemas, o subsistema de Coleta onde todos arquivos disponíveis na pasta “./documentos/” serão coletados e lidos, após a coleta, dá-se início a fase de atuação do subsistema de Indexação, onde o texto lido pela fase de Coleta é separado por palavra e contando em qual e quantas vezes a palavra aparece em um determinado arquivo, assim, indexando as palavras coletadas. Por último, o subsistema de Recuperação, onde se é digitado uma sequência de palavras e esse subsistema atua de forma a localizar quais arquivos contém todas palavras e caso haja mais de um, ele os ordena mostrando os arquivos com mais ocorrências das palavras da busca primeiro.

Implementação

Parte 1: Coleta

Na fase da Coleta é usado a biblioteca ***filesystem*** (apenas disponível para C++17) para descobrir quais são os arquivos disponíveis na pasta "./documentos/", após isso, esse resultado obtido é iterado por meio de um ***for***, e a cada arquivo presente na pasta, é aberto o arquivo usando ***fopen*** e é lido o seu conteúdo, palavra a palavra, o conteúdo do arquivo é então armazenado em um ***map*** com chave sendo o nome do arquivo(***string***) e o valor sendo o conteúdo do arquivo(***string***).

Parte 2: Indexação

Funções utilizadas:

size_t split(const string &txt, vector<string> &strs, char ch): Essa função recebe uma string **txt**, um vetor de strings **strs** e um caractere **ch** como parâmetros. Ela retorna um valor do tipo **size_t**, que representa o número de palavras encontradas na string **txt**.
O objetivo da função **split** é dividir a string **txt** em palavras individuais, usando o caractere **ch** como delimitador. No exemplo específico mencionado, o caractere delimitador é o espaço em branco **' '**.
A função começa procurando a primeira ocorrência do caractere **ch** (espaço em branco) na string **txt** usando o método **find()**. Ela inicializa as variáveis **pos** e **initialPos** com os valores correspondentes. Em seguida, ela limpa o vetor **strs** utilizando o método **clear()** para garantir que esteja vazio antes de preenchê-lo com as palavras encontradas.
Dentro de um loop while, a função continua procurando o caractere **ch** na string **txt**, começando a partir da posição **initialPos**. Ela utiliza o método **substr** para extrair a substring entre as ocorrências consecutivas do caractere **ch** e adiciona essa substring ao vetor **strs** usando o método **push_back**. A função, então, atualiza o valor de **initialPos** para a próxima posição depois do caractere **ch**.
Esse processo se repete até que não haja mais ocorrências do caractere **ch** na string **txt**. No final do loop, a função adiciona a última substring (que vai da última posição encontrada até o final da string) ao vetor **strs** usando novamente o método **substr**.
Por fim, a função retorna o tamanho do vetor **strs**, que representa o número de palavras encontradas na string **txt** e armazenadas no vetor **strs**.

bool isLetter(char c): Essa função recebe um caractere como parâmetro (**c**) e retorna um valor booleano (**true** ou **false**). Ela verifica se o caractere é uma letra do alfabeto, tanto maiúscula quanto minúscula. A expressão **(c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')** é usada para fazer essa verificação. Ela compara o valor ASCII do caractere **c** com os valores ASCII correspondentes às letras maiúsculas (**A** a **Z**) e minúsculas (**a** a **z**). Se o caractere estiver dentro desses intervalos, a função retorna **true**, indicando que é uma letra. Caso contrário, retorna **false**.

char toLowercase(char c): Essa função recebe um caractere como parâmetro (**c**) e retorna o mesmo caractere em minúsculo, se for uma letra maiúscula. Caso contrário, o caractere é retornado sem modificação. A função primeiro verifica se o caractere é uma letra maiúscula, utilizando a expressão **(c >= 'A' && c <= 'Z')**. Se essa condição for

verdadeira, é realizada uma operação de deslocamento para transformar a letra maiúscula em minúscula. Isso é feito adicionando a diferença entre os valores ASCII de **'a'** e **'A'** ao valor ASCII de **c**. Por exemplo, se **c** for **'B'**, a diferença entre **'a'** e **'A'** é 32, então **c + ('a' - 'A')** resultará em **'b'**. O caractere resultante é retornado pela função.

string normalizeWord(const string &word): Essa função recebe uma string como parâmetro (**word**) e retorna uma nova string que contém apenas as letras dessa palavra em minúsculo. A função cria uma string vazia chamada **normalized**, que será utilizada para armazenar as letras normalizadas. Em seguida, itera sobre cada caractere **c** na string **word**. Se o caractere for uma letra (verificado pela função **isLetter(c)**), o caractere é convertido para minúsculo usando a função **toLowercase(c)** e é concatenado à string **normalized** usando o operador **+=**. No final da iteração, a string **normalized** contém apenas as letras da palavra original em minúsculo, e essa string é retornada pela função.

No main:

É criada uma estrutura de dados chamada **indexedWords**, que é um mapa aninhado. O mapa externo tem como chave as palavras normalizadas e como valor um mapa interno. O mapa interno tem como chave o nome do arquivo e como valor a contagem da palavra nesse arquivo.

O código itera sobre um mapa chamado **indexedFiles**, que contém informações sobre os arquivos e suas palavras. Cada elemento desse mapa representa um arquivo e suas palavras.

Para cada elemento no mapa **indexedFiles**, é realizado o seguinte:

- O nome do arquivo é extraído e armazenado na variável **file**.
- A string de palavras do arquivo é extraída e armazenada na variável **words**.
- A string de palavras é dividida em palavras individuais usando o espaço como delimitador. As palavras são armazenadas em um vetor chamado **splitWords**.

Em seguida, para cada palavra no vetor **splitWords**, é realizado o seguinte:

- A palavra é normalizada para garantir que todas as letras estejam em minúsculas e que caracteres não alfabéticos sejam removidos. A função **normalizeWord** é responsável por essa normalização, e o resultado é armazenado na variável **normalizedWord**.
- A palavra normalizada é usada como chave no mapa externo **indexedWords** para acessar o mapa interno correspondente. Se a palavra já estiver presente no mapa, o acesso é feito diretamente; caso contrário, uma nova entrada é criada para a palavra no mapa externo.
- O nome do arquivo (**file**) é usado como chave no mapa interno para acessar o valor correspondente, que representa a contagem atual da palavra nesse arquivo. Se a palavra já estiver presente no mapa interno para esse arquivo, o acesso é feito diretamente. Caso contrário, uma nova entrada é criada para a palavra no mapa interno.
- O processo se repete para todas as palavras em todos os arquivos do mapa **indexedFiles**. Dessa forma, o mapa **indexedWords** é preenchido com a contagem de palavras normalizadas em cada arquivo.

Parte 3: Recuperação

Nesse subsistema, primeiramente é perguntado ao usuário qual as palavras que ele deseja pesquisar, logo após, a pesquisa com uma linha com diversas palavras é separada a cada espaço em branco encontrado, usando a função ***split***, essas palavras que foram separadas em um ***vector*** são então iteradas, para cada palavra, é consultado se ela está no **map** do índice invertido disponibilizado pela fase de Indexação, assim, caso ela esteja, é adicionado um item em um ***map*** onde sua chave é o nome do arquivo e seu valor um vetor onde cada posição é a quantidade de vezes que a palavra sendo iterada apareceu naquele arquivo. Desse modo, ao final da iteração das palavras, os vetores presentes nesse ***map*** que tiverem de tamanho o mesmo tamanho do vetor de palavras digitadas, será um vetor, logo, um arquivo, que atende ao requisito de ter todas as palavras da pesquisa.
Tendo esse ***map*** em mãos, podemos agora "transcrevê-lo" para outro map, neste outro ***map*** apenas os registros dos arquivos que atendem ao requisito de ter todas as palavras da pesquisa serão enviados ao novo ***map***, além disso, a chave agora é a soma dos valores do vetor, resultando na quantidade de ocorrências de todas palavras da pesquisa e o valor é um vetor com os nomes dos arquivos. Por fim, vamos iterar uma última vez o ***map*** de resultados, de trás para frente, para que possamos imprimir os índices de maior valor primeiro, já que esses são os arquivos com mais ocorrências das palavras digitadas na pesquisa.

Conclusão

Por fim, concluímos que foi uma experiência prática de grande valor, pois não estamos acostumados a lidar com programação de maneira mais interativa, como foi desta vez. Sentimos que, no geral, não foi uma tarefa simples, por se tratar de algo que foi feito e implementado em conjunto, mas o fato do algoritmo não ser muito complexo tornou a tarefa mais viável e prazerosa de ser praticada, e, como singela opinião do grupo, foi uma excelente ideia que deve ser mantida. Outro detalhe importante notado foi o fato do trabalho exercitar certos conhecimentos que foram desenvolvidos durante esta matéria, como a utilização de ponteiros, vetores, makefile, map e ferramentas com uma função mais coletiva, como o git e github, que serão muito importante e de muito uso durante nossas vidas.

Como executar o código

Ambiente de testes:

- Ambiente Linux
- Ter C++17 instalado

Comando para compilar os arquivos: **make**
Comando para executar o código: **./execute_project**