Trabalho Prático: Construção de Um Mecanismo de Indexação de Arquivos

Prof. Dr. Juliano Henrique Foleis

Objetivo

Desenvolver uma aplicação de linha de comando que simula um mecanismo de busca para documentos de texto. A aplicação deverá ter duas funcionalidades principais:

1. **Indexação:** Ler uma coleção de arquivos de texto (.txt) de um diretório, processar seu conteúdo e construir um índice invertido em memória. Ao final, este índice deve ser serializado e salvo em um arquivo no disco.
2. **Busca:** Carregar o índice previamente salvo no disco (desserialização) e permitir que o usuário realize buscas por palavras-chave, retornando de forma eficiente a lista de documentos que contêm os termos pesquisados.

Estrutura de Dados Principal

A estrutura de dados principal a ser utilizada é um **Índice Invertido**, que mapeia cada palavra para uma lista de documentos onde essa palavra aparece. A estrutura pode ser representada como um dicionário (ou mapa) onde:

- **Chave:** Uma string representando uma palavra encontrada nos documentos.
- **Valor:** Lista dos nomes dos documentos onde a palavra aparece.

Exemplo de Índice Invertido

Suponha que temos dois arquivos de texto:

- `doc1.txt`: "O gato está comendo no telhado."
- `doc2.txt`: "O cachorro está comendo no quintal."

Primeiramente , o conteúdo dos arquivos deve ser processado para remover pontuações e converter todas as palavras para minúsculas. Além disso, palavras comuns (stop words) como "o", "a", "de", "em", etc., são removidas, uma vez que não agregam valor significativo à busca.

O índice invertido resultante seria:

```
{
    "comendo": ["doc1.txt", "doc2.txt"],
    "gato": ["doc1.txt"],
```

```
    "está": ["doc1.txt", "doc2.txt"],
    "telhado": ["doc1.txt"],
    "cachorro": ["doc2.txt"],
    "quintal": ["doc2.txt"]
}
```

Na prática, seu índice invertido será diferente, pois não armazenará o nome do arquivo, mas sim identificadores numéricos (IDs) para cada documento, o que economiza espaço e melhora a eficiência. Esse mapeamento de nomes de arquivos para IDs deve ser mantido em uma estrutura eficiente separada. Veja a Seção **Eficiência** para mais detalhes.

Requisitos Funcionais

Modo de Execução

O programa deve ser executável a partir da linha de comando, aceitando argumentos para operar em modo de indexação ou de busca.

Modo Indexação

`indice construir <caminho_do_diretorio>`

Este comando deve varrer o diretório especificado, construir o índice invertido e salvá-lo em um arquivo padrão `index.dat` no diretório atual.

Modo Busca

`indice buscar <termo_de_busca>`

Este comando deve primeiro verificar se o arquivo de índice existe na pasta atual. Se não, deve informar ao usuário para executar a indexação primeiro. Se o índice existir, ele deve ser carregado em memória. A busca deve ser realizada e os nomes dos arquivos relevantes, impressos na tela.

Pré-processamento

O programa deve ser capaz de ler arquivos de texto, processar o conteúdo para remover pontuações, converter para minúsculas e eliminar stop words. A lista de *stop words* será fornecida em um arquivo `stopwords.txt`.

Lógica de Busca

A busca deve ser eficiente, retornando rapidamente os documentos que contêm o termo pesquisado. O índice deve ser armazenado em uma tabela hash. Você pode usar a biblioteca padrão do C++ (`<unordered_map>`) para implementar a tabela hash. Os parâmetros da consulta podem ser:

- Busca por uma única palavra.
- Busca por múltiplas palavras (retornando documentos que contenham todas as palavras). Neste caso, o comando de busca deve ser:

`indice buscar <termo1> <termo2> ... <termoN>`

A busca por múltiplas palavras funciona como uma operação AND. Ela pode ser implementada da seguinte forma: para cada palavra, obtenha a lista de documentos do índice invertido e faça a interseção dessas listas. A lógica de interseção deve ser eficiente, considerando que as listas podem ser grandes. **A lógica de interseção deve ser implementada manualmente, sem o uso de bibliotecas externas**.

Serialização e Desserialização

O índice invertido deve ser salvo em um arquivo binário (`index.dat`) para que possa ser carregado posteriormente. A serialização deve garantir que o índice possa ser reconstruído exatamente como era quando foi salvo. **Você não pode usar bibliotecas externas para serialização: deve implementar a lógica de serialização e desserialização manualmente**. Além do mais, o arquivo deve ser salvo em formato binário para otimizar espaço e velocidade de leitura/escrita. Além disso, o índice deve armazenar o caminho completo do diretório dos arquivos indexados, para que a busca funcione corretamente mesmo se o programa for executado em outro diretório.

Eficiência

O programa deve ser eficiente em termos de tempo e espaço. Para isso, o índice invertido deve ser armazenado de forma compacta e as operações de busca devem ser otimizadas. Algumas estratégias incluem:

- Utilizar IDs numéricos para representar documentos, ao invés de armazenar os nomes completos dos arquivos no índice. Isso reduz o espaço necessário para armazenar o índice e melhora a eficiência das operações de busca. Este mapeamento deve ser armazenado em uma estrutura separada, mas também deve ser serializado e desserializado junto com o índice.
- Utilizar uma tabela hash para armazenar o índice invertido, permitindo buscas rápidas por palavras.
- Garantir que a serialização e desserialização sejam rápidas e não consumam muita memória.
- Implementar a lógica de interseção de listas de documentos de forma eficiente, minimizando o tempo de execução.

Passos Sugeridos para Implementação em C++

O programa deverá ser implementado em `C++`, utilizando a biblioteca padrão e modularização adequada. A seguir estão os passos sugeridos para a implementação:

1. **Leitura de Arquivos:** Implemente uma função para ler todos os arquivos de texto de um diretório especificado. Utilize a biblioteca `<filesystem>` do `C++`17 para facilitar a navegação pelos arquivos.

2. **Pré-processamento de Texto:** Implemente funções para processar o conteúdo dos arquivos, removendo pontuações, convertendo para minúsculas e eliminando stop words. Carregue a lista de stop words a partir do arquivo `stopwords.txt`.

3. **Construção do Índice Invertido:** Implemente a estrutura de dados para o índice invertido usando `std::unordered_map`. Cada palavra deve mapear para uma lista de IDs de documentos.

4. **Serialização e Desserialização:** Implemente funções para salvar o índice invertido em um arquivo binário (`index.dat`) e para carregar o índice a partir desse arquivo. O arquivo deve conter tanto o índice quanto o mapeamento de IDs para nomes de arquivos.
5. **Busca:** Implemente a funcionalidade de busca, que carrega o índice do arquivo e permite ao usuário buscar por palavras-chave, retornando os nomes dos arquivos relevantes.
6. **Interseção de Listas:** Implemente a lógica para interseção de listas de documentos quando múltiplas palavras são buscadas.
7. **Interface de Linha de Comando:** Implemente a interface de linha de comando para aceitar os comandos `construir` e `buscar`, conforme especificado.

Sugestão de Estrutura de Classes

Para organizar a implementação do projeto, sugere-se uma arquitetura baseada em classes, onde cada classe possui uma responsabilidade única e bem definida. A seguir, é apresentado um diagrama que ilustra a relação entre essas classes e a descrição das suas respectivas responsabilidades.

Diagrama de Classes

O diagrama abaixo mostra a interação entre os componentes principais do sistema. A função `main` atuará como a orquestradora, criando e utilizando os objetos necessários para executar as operações de indexação e busca.

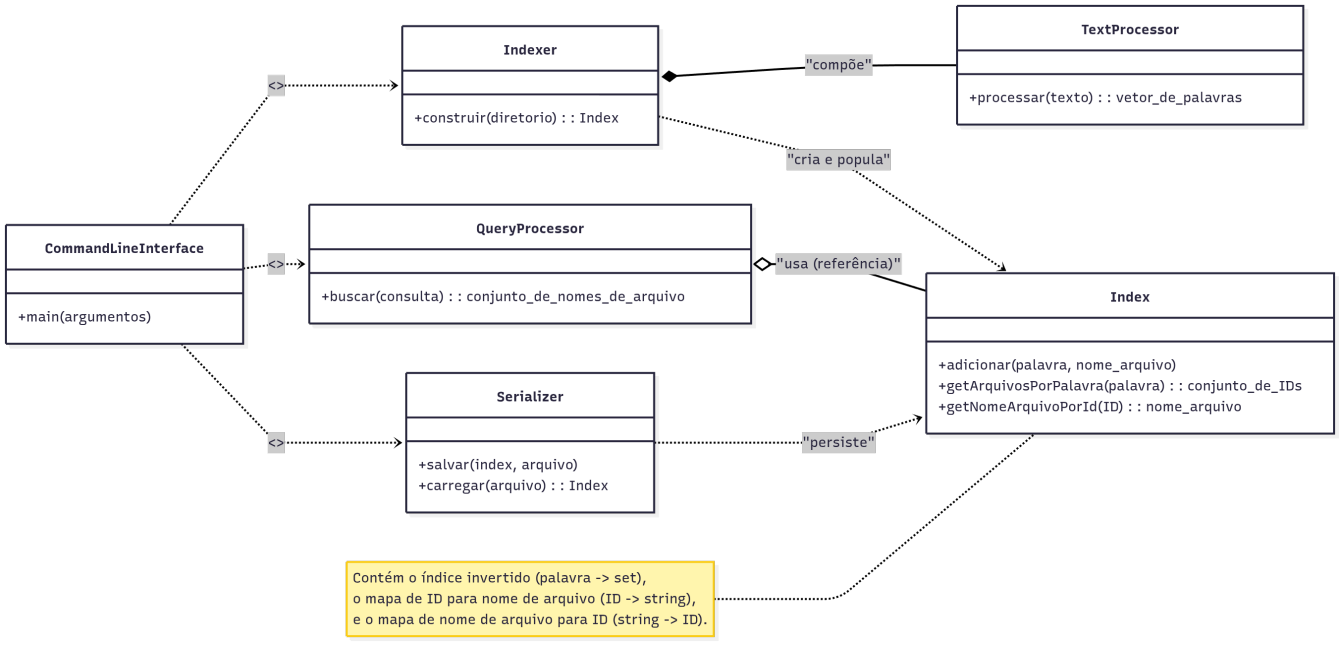

Figure 1: Diagrama de Classes

Responsabilidades das Classes

`Index` (Índice Invertido)

Responsável por encapsular toda a lógica e as estruturas de dados do índice. Suas tarefas são:

1. Manter o índice invertido principal, que mapeia uma palavra (string) para um conjunto de IDs de documentos (set).
2. Gerenciar o mapeamento de documentos, mantendo duas estruturas de dados: uma para converter um nome de arquivo em um ID único e outra para fazer o caminho inverso (ID para nome de arquivo).
3. Fornecer uma interface simples para o resto do sistema, com métodos como adicionar(palavra, nome_arquivo), que esconde a complexidade de criar IDs e atualizar as estruturas internas.

`TextProcessor` (Processador de Texto)

Responsável por receber uma string contendo texto e realizar sua normalização. Isso inclui converter o texto para letras minúsculas, remover pontuações e quebrar o texto em uma lista de palavras individuais (tokens). É um componente "utilitário" focado exclusivamente na limpeza dos dados.

`Indexer` (Indexador)

Responsável por orquestrar o processo de construção do índice. Seu objetivo é popular o objeto `Index` com dados. Ele deve:

1. Varrer o diretório especificado, lendo cada arquivo de texto.
2. Utilizar o `TextProcessor` para normalizar o conteúdo de cada arquivo.
3. Adicionar as palavras normalizadas ao índice, associando-as ao ID do documento correspondente

`Serializer` (Serializador)

Responsável por lidar com a persistência do objeto `Index`. Sua lógica agora precisa salvar e carregar todas as estruturas de dados internas da classe `Index` de/para um arquivo binário. Ele deve garantir que o índice possa ser reconstruído exatamente como era quando foi salvo.

`QueryProcessor` (Processador de Consultas)

Responsável por interpretar e executar as consultas de busca. Recebe uma referência para o objeto `Index` e fornece métodos para buscar palavras individuais ou múltiplas palavras, retornando os nomes dos arquivos relevantes. Ele deve implementar a lógica de interseção de listas de documentos quando múltiplas palavras são buscadas.

`CommandLineInterface` (Interface de Linha de Comando)

Responsável por interpretar os argumentos da linha de comando e orquestrar a execução do programa. Dependendo dos argumentos, ele deve criar e utilizar os objetos `Indexer`, `Serializer` e `QueryProcessor` para realizar as operações de indexação ou busca.

Regras

- O trabalho pode ser feito (no máximo) em duplas.
- Se o trabalho for feito em duplas, ambos alunos tem que escrever parte do código.
- O código vale 60% da nota do trabalho.
- Haverá uma avaliação escrita individual, valendo 40% da nota do trabalho. Esta avaliação abordará as estratégias de implementação, estruturas de dados utilizadas e decisões de projeto. A avaliação irá incluir questões sobre o código desenvolvido.
- O trabalho deve ser implementado em linguagem `C++`, usando modularização adequada e Makefile para compilar.
- Use apenas a biblioteca padrão do `C++`. Não é permitido o uso de bibliotecas externas.
- Você pode usar a `STL` (Standard Template Library) do `C++`, como `std::vector`, `std::unordered_map`, `std::set`. Entretanto, a lógica de serialização, desserialização e interseção de listas deve ser implementada manualmente, sem o uso de bibliotecas externas.
- Trabalhos plagiados uns dos outros ou da internet serão totalmente desconsiderados, sem possibilidade de recuperação.
- Comente o código nos pontos mais importantes. Não é necessário fazer textão :)
- O código deve ser compilável e executável em uma máquina Linux (Ubuntu 20.04 ou superior). Se você usar apenas a biblioteca padrão do `C++`, isso não será um problema.
- O código deve ser bem estruturado, modularizado e seguir boas práticas de programação.

Entrega

- Um arquivo .zip (ou .tar.gz) contendo todo o código do trabalho. Inclua também um arquivo alunos.txt, com os nomes dos alunos da equipe, e um LEIAME.txt com informações adicionais para compilação e execução.
- Prazo máximo para entrega: **24/11/2025 às 23:59**, via Moodle. Ambos integrantes da equipe precisam enviar o trabalho.
- A avaliação escrita será durante a aula, no dia **27/11/2025**.

Bom Trabalho!