Trabalho prático 2 Banco de Dados

Isabella A. Macedo Daniel[1]**,Lucas do Nascimento Silva**[1]

[1]Instituto de Computação – Universidade Federal do Amazonas (UFAM)
Av. Gen. Rodrigo Octávio, 6200, Coroado I
69080-900 – Manaus – AM

{ isabella.macedo, lucas.nascimento}@icomp.ufam.edu.br

Resumo. *Este projeto envolve a criação de programas para o armazenamento e consulta de dados que estão em memória secundária, utilizando estruturas de arquivos de dados e índices conforme estudadas em aula. Os programas devem permitir operações de inserção e oferecer diferentes estratégias de busca, alinhando-se com as técnicas de organização e indexação de arquivos apresentadas no curso.*

Estrutura dos Arquivos de Dados e Índices

Nesta seção da documentação, serão detalhadas as estruturas, as decisões de projeto e o formato utilizado para cada uma das estruturas implementadas.

1. Arquivos e Suas Estruturas

1.1. bloco.cpp - Responsável: Isabella e Lucas

Este arquivo lida com blocos que armazenam registros. Cada bloco é composto de:

- **HeaderBlock**: Contém metadados do bloco, com:
  - `espacoDisponivel`: Quantidade de espaço livre em bytes.
  - `quantidadeRegistros`: Número de registros armazenados no bloco.
  - `offsetsRegistros`: Lista de offsets, onde cada offset indica a posição de um registro dentro do bloco.
- **Bloco**: Representa um bloco de dados que contém:
  - `header`: Cabeçalho do bloco (HeaderBlock) com metadados.
  - `dadosBloco`: Array de bytes para armazenar registros compactados.

1.1.1. Fontes Incluídas

- `<iostream>`: Usado para exibir informações, como na função `imprimirBloco`.
- `<vector>`: Utilizado para armazenar os offsets de registros dentro de cada bloco de forma dinâmica.
- `<cstring>`: Fornece funções de manipulação de memória, como `memcpy`.

1.1.2. Funções Contidas e Seus Papéis

1. `void sizeRegistro(Registro* registro)`:
    - **Papel**: Calcula e atualiza o tamanho total do registro (`tamanhoRegistro`), somando o tamanho de cada campo, incluindo os tamanhos das strings e o caractere nulo (`'\0'`).
2. `void printRegistro(const Registro& registro)`:
    - **Papel**: Exibe o conteúdo de um `Registro` no console em formato legível, incluindo campos como `id`, `ano`, `titulo`, `autores`, etc.
3. `Registro createRegistro(...)`:
    - **Papel**: Inicializa um novo `Registro` com valores padrão ou especificados. Chama `sizeRegistro` para calcular e atualizar o campo `tamanhoRegistro`.
4. `bool compareRegistroById(const Registro& a, const Registro& b)`:
    - **Papel**: Compara dois registros com base no campo `id`, retornando `true` se o `id` do primeiro registro for menor, útil para ordenações ou buscas.
5. `Bloco criarBloco()`:
    - **Papel**: Inicializa um novo bloco vazio, configurando `espacoDisponivel` e `quantidadeRegistros` no cabeçalho e zerando o conteúdo de `dadosBloco`.
6. `Bloco extrairHeader(const char* bytesBloco)`:
    - **Papel**: Extrai o cabeçalho (`HeaderBlock`) de um bloco, copiando `espacoDisponivel`, `quantidadeRegistros`, e `offsetsRegistros` de um array de bytes para uma estrutura `Bloco`.
7. `void imprimirBloco(const Bloco* bloco)`:
    - **Papel**: Exibe os metadados do bloco, como espaço disponível, quantidade de registros e offsets dos registros, facilitando a depuração e verificação do conteúdo.
8. `bool inserirRegistroNoBloco(Bloco* bloco, const Registro& registro)`:
    - **Papel**: Insere um `Registro` no bloco se houver espaço disponível.
    - **Detalhes**:
        - Verifica se há espaço para o registro.
        - Calcula o `enderecoInsercao`, inserindo o registro no final ou antes do último registro.
        - Copia cada campo do registro para `dadosBloco`.
        - Atualiza `espacoDisponivel` e `quantidadeRegistros` no cabeçalho e adiciona o novo offset aos `offsetsRegistros`.
        - Retorna `true` se a inserção foi bem-sucedida ou `false` caso contrário.
9. `Registro* buscarRegistroNoBloco(const Bloco* bloco, int idRegistro)`:
    - **Papel**: Busca um registro pelo `id` no bloco.
    - **Detalhes**:

- Percorre os `offsetsRegistros` e lê o `id` do registro na posição do offset.
- Se o `id` corresponde ao `idRegistro` buscado, carrega os dados restantes do registro.
- Retorna o `Registro` encontrado ou `nullptr` se o registro não for encontrado.

1.2. bucket.cpp - Responsável: Isabella e Lucas

1.2.1. Estrutura do Arquivo de Dados e Índices

O código define a estrutura de um `bucket`, que contém um array de índices de blocos (`indicesBlocos`). Cada índice aponta para um bloco específico, e os buckets são usados para armazenar e gerenciar grupos de blocos dentro de um sistema de armazenamento baseado em hashing.

- **Bucket**:
  - Contém o array `indicesBlocos` com `BUCKET_SIZE` posições. Cada índice aponta para um bloco no arquivo de dados.

1.2.2. Fontes Incluídas

- `"bloco.hpp"`: Permite o uso da estrutura `Bloco` e de funções associadas, como `criarBloco`.
- `<fstream>`: Suporta operações de leitura e escrita em arquivos binários.
- `<cstring>`: Usado para operações de manipulação de memória, como `memcpy`, para copiar dados entre buffers e estruturas.

1.2.3. Funções Contidas e Seus Papéis

1. `Bucket* criarBucket(std::ofstream& arquivoBinario)`:
   - **Papel**: Cria um bucket e inicializa blocos vazios no arquivo binário.
   - **Detalhes**:
     - Cria uma nova instância `Bucket`.
     - Inicializa um buffer (`buffer`) para armazenar os dados de cada bloco.
     - Para cada posição em `indicesBlocos`:
       - Define o índice do bloco no bucket.
       - Usa `memcpy` para copiar o cabeçalho do bloco e os dados para o buffer.
       - Escreve o buffer no arquivo binário para armazenar o bloco inicializado.
     - Retorna o bucket inicializado.
2. `Bloco* carregarBloco(int enderecoBloco, std::ifstream& arquivoBinario)`:
   - **Papel**: Carrega um bloco específico de um arquivo binário para a memória.
   - **Detalhes**:

        - Move o ponteiro do arquivo para `enderecoBloco`.
        - Lê os dados do bloco e armazena em `dadosBloco`.
        - Extrai o cabeçalho (`HeaderBlock`) usando `extrairHeader` e o armazena no `header` do bloco.
        - Retorna o bloco carregado.
3. `bool escreverRegistroNoBucket(int enderecoBloco, Bloco* bloco, const Registro& registro, std::ofstream& arquivoEscrita)`:
    - **Papel**: Escreve um registro no bloco correspondente dentro de um bucket, com base em uma função hash.
    - **Detalhes**:
        - Usa `inserirRegistroNoBloco` para adicionar o registro ao bloco.
        - Move o ponteiro de escrita para `enderecoBloco` no arquivo.
        - Escreve o conteúdo atualizado do bloco (`dadosBloco`) de volta no arquivo.
        - Retorna `true` para indicar sucesso.

1.3. btree.cpp - Responsável: Isabella e Lucas

1.3.1. Estrutura do Arquivo de Dados e Índices

Este código implementa uma estrutura de `árvore B+` usada para armazenar índices de forma hierárquica, facilitando a busca eficiente de dados. Ele define dois tipos principais de nós:

- **No**: Representa um nó da árvore B+ na memória principal.
- **NoDisco**: Estrutura para armazenar nós em disco, com offsets para chaves e ponteiros.

1.3.2. Fontes Incluídas

- `<iostream>`: Suporta operações de saída para exibir informações no console.
- `<cstdio>`, `<cstdlib>`, `<cstring>`: Usadas para manipulação de arquivos, alocação de memória, e cópia de dados em memória.
- `<queue>`: Fornece uma fila para percorrer a árvore.
- `"bloco.hpp"`: Inclui a estrutura `BlocoOffset`, que armazena o offset de um bloco de dados.

1.3.3. Estruturas de Dados

1. **BlocoOffset**: Armazena o offset de um registro no arquivo.
2. **No**: Representa um nó da árvore B+ em memória com as seguintes propriedades:
    - `ehFolha`: Indica se o nó é uma folha.
    - `ponteiros`: Array de ponteiros para filhos (em nós internos) ou registros (em folhas).
    - `chaves`: Array de chaves no nó.

- `pai`: Ponteiro para o nó pai.
- `numChaves`: Número de chaves armazenadas no nó.

3. **NoDisco**: Representa um nó armazenado em disco:
   - `ehFolha`: Indica se o nó é uma folha.
   - `ponteiros`: Array de offsets para filhos ou registros.
   - `chaves`: Array de chaves armazenadas no nó.
   - `pai`: Offset para o nó pai no arquivo.
   - `numChaves`: Número de chaves armazenadas no nó.

1.3.4. Funções Contidas e Seus Papéis

1. `int calcularPontoDivisao(int tamanho)`:
   - **Papel**: Calcula o ponto de divisão (split) de um nó, retornando o ponto onde o nó será dividido.
2. `No* criarNo()`:
   - **Papel**: Cria e inicializa um nó vazio para a `árvore B+`.
3. `No* criarFolha()`:
   - **Papel**: Cria e inicializa um nó folha.
4. `No* inicializarArvore(int chave, BlocoOffset* bloco)`:
   - **Papel**: Inicializa uma nova árvore B+ com uma chave e seu bloco correspondente, criando uma raiz que é uma folha.
5. `BlocoOffset* criarBlocoOffset(int offset)`:
   - **Papel**: Cria um bloco para armazenar o offset de uma chave.
6. `No* encontrarFolha(No* raiz, int chave)`:
   - **Papel**: Percorre a árvore a partir da raiz até encontrar a folha onde uma chave deve ser inserida.
7. `No* inserirEmFolha(No* folha, int chave, BlocoOffset* bloco)`:
   - **Papel**: Insere uma chave e um bloco em uma folha com espaço disponível.
8. `No* inserirNovaRaiz(No* esquerda, int chave, No* direita)`:
   - **Papel**: Cria uma nova raiz após a divisão e insere a chave intermediária.
9. `int obterIndiceEsquerdo(No* pai, No* esquerda)`:
   - **Papel**: Retorna o índice do ponteiro esquerdo no nó pai.
10. `No* inserirEmNo(No* raiz, No* no, int indiceEsquerdo, int chave, No* direita)`:
    - **Papel**: Insere uma chave e ponteiro em um nó interno com espaço disponível.
11. `No* inserirNoPai(No* raiz, No* esquerda, int chave, No* direita)`:
    - **Papel**: Insere um nó no pai após a divisão de um nó filho, criando uma nova raiz se necessário.
12. `No* inserirEmNoAposDivisao(...)`:
    - **Papel**: Insere uma chave em um nó interno que precisa ser dividido, promovendo a chave intermediária para o pai.
13. `No* inserirEmFolhaAposDivisao(...)`:
    - **Papel**: Insere uma chave em uma folha que precisa ser dividida.

14. `No* inserir(No* raiz, int chave, int offset)`:
    - **Papel**: Função principal para inserir uma chave e seu offset na árvore B+, chamando as funções apropriadas dependendo da necessidade de divisão.
15. `int calcularNivel(No* const raiz, No* filho)`:
    - **Papel**: Calcula o nível de um nó em relação à raiz.
16. `void imprimirNoDisco(NoDisco no)`:
    - **Papel**: Exibe informações de um nó `NoDisco` no console, útil para depuração.
17. `unsigned long buscarChave(unsigned int chave, unsigned long posicao, unsigned int* acessosDisco, FILE* arquivo)`:
    - **Papel**: Realiza a busca de uma chave em uma árvore B+ armazenada em disco, retornando o offset do registro se encontrado.
18. `void imprimirArvoreDisco(unsigned long posicao, FILE* arquivo)`:
    - **Papel**: Imprime todos os nós de uma árvore B+ armazenada em disco.
19. `void imprimirArvore(No* const raiz)`:
    - **Papel**: Imprime a árvore B+ armazenada em memória, usando uma fila para percorrer a árvore em nível.
20. `unsigned long gravarArvore(No* raiz, unsigned long paiPosicao, FILE* arquivo)`:
    - **Papel**: Grava a árvore B+ em um arquivo no disco, usando a estrutura `NoDisco`.

1.4. hash.cpp - Responsável: Isabella e Lucas

1.4.1. Estrutura do Arquivo de Dados e Índices

Define uma estrutura de `tabela hash` para armazenar registros de forma rápida e eficiente, organizados por meio de uma função de hashing. Cada entrada da tabela hash é um `bucket`, que armazena blocos de registros. A organização é facilitada pelas estruturas e funções para gerar índices a partir de chaves inteiras ou strings.

- **TabelaHash**:
  - Contém um array de ponteiros para `Bucket`, onde cada bucket armazena uma lista de blocos de registros.
  - Tamanho: `HASH_SIZE`, definido para determinar o número de entradas na tabela hash.

1.4.2. Fontes Incluídas

- `"bucket.hpp"`: Inclui a definição de `Bucket` e funções associadas para criar e manipular buckets.
- `"btree.hpp"`: Inclui estruturas e funções para manipulação de árvores B+, utilizadas para indexação adicional.
- `<climits>`: Fornece constantes, como `INT_MAX`, para limitar o valor de hash gerado.
- `<string>`: Suporte para manipulação de strings, especialmente útil na função de hashing para strings.

1.4.3. Funções Contidas e Seus Papéis

1. `TabelaHash* inicializarTabela(std::ofstream& arquivoSaida)`:
    - **Papel**: Inicializa a tabela hash, preenchendo-a com buckets vazios e armazenando-os em um arquivo binário.
    - **Detalhes**:
        - Cria uma nova `TabelaHash`.
        - Para cada posição em `HASH_SIZE`, chama `criarBucket` para inicializar um bucket vazio no arquivo binário e adiciona-o à tabela.
        - Retorna o ponteiro para a tabela hash inicializada.
2. `int gerarIndice(int chave)`:
    - **Papel**: Gera um índice hash a partir de uma chave inteira, retornando um valor no intervalo da tabela hash.
    - **Detalhes**:
        - Usa uma série de operações bitwise e multiplicação para dispersão dos bits da chave.
        - Retorna o índice resultante ao fazer um módulo com `HASH_SIZE`.
3. `int gerarIndiceString(const std::string& texto)`:
    - **Papel**: Gera um índice hash a partir de uma string.
    - **Detalhes**:
        - Utiliza o algoritmo de hashing `djb2`, multiplicando cada caractere pelo fator 33 para gerar um hash.
        - Retorna o índice obtido pelo módulo com `INT_MAX + 1`.
4. `std::pair<No*, No*> inserirRegistro(...)`:
    - **Papel**: Insere um `Registro` na tabela hash, criando ou atualizando as raízes da árvore B+ para `ID` e `titulo`.
    - **Detalhes**:
        - Calcula o índice do registro pelo `ID` e tenta inserir o registro no bucket correspondente.
        - Para cada tentativa:
            - Carrega o bloco no endereço especificado.
            - Se houver espaço, insere o registro no bloco, inicializando ou atualizando as árvores B+ (`raizID` e `raizTitulo`) para `ID` e `titulo`.
            - Se o registro for inserido com sucesso, retorna as novas raízes da árvore B+.
            - Se o bucket está cheio após todas as tentativas, retorna as raízes sem inserção.
5. `Registro* buscarRegistroPorID(int idRegistro, std::ifstream& arquivoLeitura)`:
    - **Papel**: Localiza um registro na tabela hash usando o `ID`.
    - **Detalhes**:
        - Calcula o índice hash do `ID` e tenta encontrar o registro no bucket associado.
        - Para cada tentativa:

        * Carrega o bloco no endereço.
        * Se o bloco estiver vazio, encerra a busca.
        * Usa `buscarRegistroNoBloco` para verificar se o registro está no bloco carregado.
        * Se o registro for encontrado, exibe o número de blocos verificados e o total de blocos da tabela.
    - Retorna o registro encontrado ou `nullptr` se não encontrado.

1.5. uploads.cpp - Responsável: Lucas

Aqui está a análise detalhada do arquivo, abordando a estrutura dos arquivos de dados e índices, bibliotecas incluídas, funções contidas e o papel de cada uma.

1.5.1. Estrutura do Arquivo de Dados e Índices

Este programa utiliza uma `tabela hash` para armazenar registros de forma organizada e eficiente e cria índices B+ (para `ID` e `Título`) para consultas rápidas. A estrutura do programa segue estas etapas:

- **Tabela Hash**: Armazena buckets de registros, facilitando a recuperação com base em hashing.
- **Árvores B+** (Índice Primário e Secundário): São criadas para os campos `ID` (índice primário) e `Título` (índice secundário), permitindo buscas eficientes por esses campos.

1.5.2. Fontes Incluídas

- `"./Structure/hash.hpp"`: Inclui a definição e manipulação da tabela hash e funções associadas.
- `<iostream>`: Suporta operações de entrada e saída para interação com o usuário e exibição de progresso.
- `<fstream>`: Permite operações de leitura e escrita em arquivos binários.
- `<string>`: Suporte para manipulação de strings, necessário para os campos de texto dos registros.
- `<cstdio>` e `<cstring>`: Manipulação de arquivos em C e operações de cópia de strings, utilizadas na leitura dos campos.

1.5.3. Funções Contidas e Seus Papéis

1. `bool lerProximoCampo(FILE* arquivoEntrada, char campo[], const std::string& padrao)`:
    - **Papel**: Lê o próximo campo do arquivo de entrada até encontrar um delimitador (`padrao`).
    - **Detalhes**:
        - Armazena o conteúdo do campo até encontrar um caractere de parada (como `;` ou nova linha).

- Se o campo estiver vazio ou com um caractere especial inicial, armazena "NULL".
- Retorna true quando chega ao fim do arquivo, ou false caso contrário.

2. int main():
   - **Papel**: Função principal que realiza as seguintes etapas:
     - **Leitura do Caminho do Arquivo**:
       * Solicita o caminho do arquivo de entrada ao usuário e o abre.
       * Calcula o tamanho total do arquivo para controle de progresso.
     - **Inicialização e Criação da Tabela Hash**:
       * Cria um arquivo binário (dados.bin) para armazenar os registros.
       * Inicializa a tabela hash com buckets vazios, gravando-os no arquivo binário.
     - **Inicialização das Árvores B+ para Índices**:
       * Declara ponteiros para as raízes das árvores B+ de ID e Título.
       * Abre o arquivo dados.bin para leitura e escrita.
     - **Inserção de Registros e Atualização dos Índices**:
       * Para cada registro no arquivo de entrada:
         · Lê os campos (id, titulo, ano, autores, citacoes, atualizacao, snippet) usando lerProximoCampo.
         · Cria um Registro com os dados lidos.
         · Insere o registro na tabela hash e nas árvores B+, usando inserirRegistro.
         · Atualiza e exibe o progresso em porcentagem.
     - **Gravação dos Índices B+**:
       * Cria arquivos indiceID.bin e indiceTitulo.bin para armazenar os índices B+.
       * Grava as árvores B+ (para ID e Título) nos arquivos correspondentes, utilizando gravarArvore.
     - **Fechamento de Arquivos e Limpeza de Memória**:
       * Fecha todos os arquivos e libera a memória alocada para a tabela hash e raízes das árvores.

1.6. seek1.cpp - Responsável: Isabella

1.6.1. Estrutura do Arquivo de Dados e Índices

Este programa busca registros pelo ID em um arquivo de índice primário (indiceID.bin). O índice é organizado em uma estrutura de árvore B+, armazenada em disco, que permite buscas rápidas e estruturadas. Ao encontrar o registro no índice, o programa exibe a quantidade de blocos lidos e o total de blocos no arquivo de índice.

1.6.2. Fontes Incluídas

- `"./Structure/hash.hpp"`: Inclui definições e funções de hashing e busca, incluindo a função `buscarChave` para busca pelo `ID`.
- `<iostream>`: Suporta operações de entrada e saída para interação com o usuário e exibição dos resultados.
- `<fstream>`: Facilita operações de leitura e escrita em arquivos binários.
- `<string>`: Suporte para manipulação de strings, necessário para processar a entrada do `ID`.

1.6.3. Funções Contidas e Seus Papéis

1. `int main()`:
    - **Papel**: Função principal que realiza a busca de registros pelo `ID` no arquivo de índice e exibe informações sobre a busca.
    - **Etapas**:
        - **Abertura do Arquivo de Índice**:
            - Abre o arquivo `indiceID.bin` em modo binário para leitura e escrita.
            - Verifica se o arquivo foi aberto com sucesso. Caso contrário, exibe uma mensagem de erro e encerra o programa.
        - **Entrada do Usuário e Conversão para Inteiro**:
            - Solicita o `ID` do registro ao usuário em um loop contínuo até que o usuário insira `"exit"`.
            - Converte a entrada para um número inteiro (`idBusca`) usando `std::stoi`. Em caso de erro de conversão (entrada inválida), exibe uma mensagem de erro e continua o loop.
        - **Inicialização e Busca pelo Registro**:
            - Inicializa `contadorAcessos` para contar o número de acessos ao índice durante a busca.
            - Usa a função `buscarChave` para procurar o `enderecoRegistro` no índice:
                - `buscarChave` tenta localizar o `ID` especificado e atualiza `contadorAcessos` com o número de blocos acessados durante a busca.
        - **Verificação e Exibição do Resultado**:
            - Verifica se `enderecoRegistro` é válido (diferente de `-1`). Se não for, o registro não foi encontrado, e uma mensagem é exibida.
            - Se o registro foi encontrado:
                - Exibe o `ID` e o número de acessos ao índice (`contadorAcessos`).
                - Calcula o total de blocos no arquivo de índice:
                - Usa `fseek` para mover o ponteiro para o final do arquivo (`SEEK_END`) e `ftell` para obter o tamanho total do arquivo em bytes.

        - Divide o tamanho total pelo tamanho do bloco (`BLOCO_SIZE`) para determinar o número total de blocos.
        - Exibe o número total de blocos do arquivo de índice.
    - **Encerramento**:
        - Fecha o arquivo de índice antes de encerrar o programa.

1.7. seek2.cpp - Responsável: Isabella

1.7.1. Estrutura do Arquivo de Dados e Índices

Este programa permite a busca de registros pelo título em um arquivo de índice secundário (`indiceTitulo.bin`). O índice secundário é organizado em uma estrutura de `árvore B+`, que permite buscas eficientes por meio de um hash gerado a partir do título. Ao encontrar o registro no índice, o programa exibe a quantidade de blocos lidos e o total de blocos do arquivo de índice secundário.

1.7.2. Fontes Incluídas

- `"./Structure/hash.hpp"`: Inclui definições e funções para manipulação de tabelas hash, incluindo `gerarIndiceString` e `buscarChave`, necessárias para calcular o hash do título e buscar o registro.
- `<iostream>`: Suporta operações de entrada e saída para interação com o usuário e exibição dos resultados da busca.
- `<fstream>`: Facilita operações de leitura e escrita em arquivos binários.
- `<string>`: Suporte para manipulação de strings, essencial para processar a entrada do título.
- `<cstring>`: Oferece funcionalidades para manipulação de C-strings, se necessário.

1.7.3. Funções Contidas e Seus Papéis

1. `int main()`:
    - **Papel**: Função principal que realiza a busca de registros pelo título no arquivo de índice secundário e exibe informações sobre a busca.
    - **Etapas**:
        - **Abertura do Arquivo de Índice**:
            - Abre o arquivo `indiceTitulo.bin` em modo binário para leitura e escrita.
            - Verifica se o arquivo foi aberto com sucesso. Caso contrário, exibe uma mensagem de erro e encerra o programa.
        - **Entrada do Usuário e Leitura do Título**:
            - Solicita o título do registro ao usuário em um loop contínuo até que o usuário insira `"exit"`.
            - Lê o título fornecido pelo usuário e armazena em `titulo`.
        - **Inicialização e Cálculo do Hash do Título**:

        * Inicializa `contadorAcessos` para contar o número de acessos ao índice durante a busca.
        * Usa `gerarIndiceString` para calcular `chaveTitulo`, um hash baseado no título, que serve como chave para a busca no índice.
    - **Busca pelo Registro no Índice**:
        * Chama `buscarChave` com `chaveTitulo` para localizar `enderecoRegistro` no índice. A função também atualiza `contadorAcessos` com o número de blocos acessados.
        * Verifica se `enderecoRegistro` é válido (diferente de `-1`). Se não for, o registro não foi encontrado, e uma mensagem é exibida.
    - **Exibição do Resultado**:
        * Se o registro for encontrado:
            · Exibe o título, o número de acessos ao índice (`contadorAcessos`) e o total de blocos no arquivo de índice secundário.
            · Calcula o total de blocos usando `fseek` para mover o ponteiro para o final do arquivo (`SEEK_END`) e `ftell` para obter o tamanho total em bytes.
            · Divide o tamanho total pelo tamanho de bloco (`BLOCO_SIZE`) para obter o número de blocos.
        * Exibe o número total de blocos do arquivo de índice.
    - **Encerramento**:
        * Fecha o arquivo de índice antes de encerrar o programa.

1.8. findrec.cpp - Responsável: Lucas

1.8.1. Estrutura do Arquivo de Dados e Índices

Este programa permite a busca e exibição de registros pelo `ID` diretamente no arquivo de dados binário (`dados.bin`). A função `buscarRegistroPorID` é usada para localizar o registro específico no arquivo. Provavelmente, essa função usa um índice hash para localizar rapidamente o registro no arquivo de dados com base no `ID`.

1.8.2. Fontes Incluídas

- `"./Structure/hash.hpp"`: Inclui definições e funções de manipulação de tabelas hash, especificamente `buscarRegistroPorID`, que localiza registros no arquivo de dados pelo `ID`.
- `<iostream>`: Suporta operações de entrada e saída para interação com o usuário e exibição dos resultados da busca.
- `<fstream>`: Facilita operações de leitura no arquivo de dados binário.
- `<string>`: Suporte para manipulação de strings, essencial para processar a entrada do `ID`.

1.8.3. Funções Contidas e Seus Papéis

1. `int main()`:
    - **Papel**: Função principal que realiza a busca de registros pelo `ID` no arquivo de dados binário e exibe as informações do registro encontrado.
    - **Etapas**:
        - **Abertura do Arquivo de Dados**:
            - Abre o arquivo `dados.bin` em modo binário para leitura.
            - Verifica se o arquivo foi aberto com sucesso. Caso contrário, exibe uma mensagem de erro e encerra o programa.
        - **Entrada do Usuário e Conversão para Inteiro**:
            - Solicita o `ID` do registro ao usuário em um loop contínuo até que o usuário insira `"exit"`.
            - Converte a entrada para um número inteiro (`idRegistro`) usando `std::stoi`. Em caso de erro de conversão (entrada inválida), exibe uma mensagem de erro e continua o loop.
        - **Busca pelo Registro no Arquivo de Dados**:
            - Chama `buscarRegistroPorID` para procurar o `ID` especificado no arquivo de dados.
            - `buscarRegistroPorID` provavelmente usa um índice hash para localizar o registro com o `ID` fornecido, retornando um ponteiro para o registro encontrado, ou `nullptr` se o registro não estiver presente.
        - **Exibição do Resultado**:
            - Verifica se o ponteiro `registro` não é nulo, indicando que o registro foi encontrado.
            - Se o registro for encontrado:
                - Usa `printRegistro` para exibir todos os campos do registro no console.
                - Libera a memória alocada para o registro.
            - Se o registro não foi encontrado, exibe uma mensagem informando que o registro não está presente no arquivo de dados.
        - **Encerramento**:
            - Fecha o arquivo de dados antes de encerrar o programa.