Trabalho Prático 2 de Banco de Dados

José Mateus Córdova Rodrigues[1], **Giovanna Andrade Santos**[2]

[1]Instituto de Computação – Universidade Federal do Amazonas (UFAM)
Manaus – AM – Brazil

jose.cordova@icomp.ufam.edu.br, giovanna.andrade@icomp.ufam.edu.br

Resumo. *Este trabalho consiste na implementação de programas para armazenamento e consulta em dados armazenados em memória secundária utilizando as estruturas de arquivo de dados e índice estudadas nas aulas. Os programas devem fornecer suporte para a inserção, assim como diferentes formas de busca, seguindo as técnicas apresentadas nas aulas de organização e indexação de arquivos.*

1. A Estrutura de Cada Arquivo de Dados e Índices

Nesta seção da Documentação será comentada as estruturas, decisões de projeto e formato utilizado em cada uma das estruturas

1.1. Registro

O registro utilizado no trabalho segue o formato mostrado na Tabela 1. No qual os campos de tipo alfa e data e hora serão considerados string para facilitar a implementação e podermos economizar espaço no armazenamento.

Table 1. Campos de um registro.

Campo	Tipo	Descrição
ID	inteiro	Código identificador do artigo
Título	alfa 300	Título de artigo
Ano	inteiro	Ano de publicação do artigo
Autores	alfa 150	Lista dos autores do artigo
Citações	inteiro	Número de vezes que o artigo foi citado
Atualização	data e hora	Data e hora da última atualização dos dados
Snippet	alfa entre 100 e 1024	Resumo textual do artigo

Já que temos alguns dados que podem variar bastante seu tamanho, inclusive serem NULL, decidimos fazer o registro de **formato fixo e tamanho variado** e alocação **não espalhada**. O formato é fixo pois todos os registros devem conter campos, na leitura caso não seja dado o valor de um campo é inserido NULL. O registro tem tamanho variável para que assim possamos colocar mais registros em um bloco e economizar espaço, uma vez que existem registros muito grandes e outros bem pequenos. A alocação será não espalhada para não gerar fragmentação do registro, assim evitando que tenhamos que carregar vários blocos para carregar o registro, assim diminuindo as idas em disco, e facilitar a implementação.

Ao Figura 1 representa como um registro é gravado em disco. Escrevemos primeiro o ID para que possamos saber saber se aquele é o registro que queremos sem precisarmos ler ele todo, o segundo campo é o tamanho do registro que nos ajuda a saber

onde o registro acaba, os outros campos foram os inteiros e de tamanho fixo, depois os de tamanho variado, assim nos garantindo que todos os registros sempre vão ter os primeiros bytes de um tamanho já conhecido (a parte dos campos fixos) e outra que cresce dependendo do conteúdo do registro. Os campos Atualização, Título, Autores e Snippet são tamanho variado, e os outros 4 primeiros são inteiro, portanto 4 bytes.

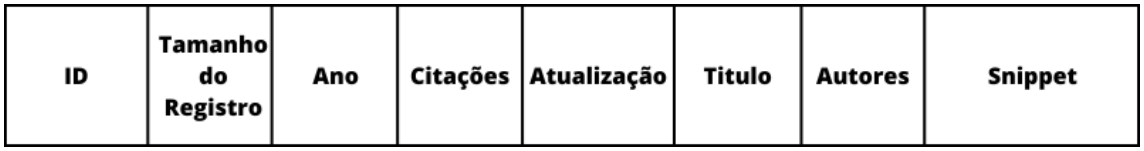

Figure 1. Exemplo de Formato do Registro

1.2. Bloco

O bloco no nosso trabalho tem **tamanho fixo de 4096 bytes**. Na nossa implementação ”dividimos” o bloco em 2 partes, o Header e o Corpo onde inserimos os registros, ambos estão no bloco. O header guarda quanto de espaço livre o bloco tem, o número de registros que tem e os próximos bytes são o offset para o primeiro byte do registro.

O Header cresce de acordo com a inserção de registros, pois temos que guardar seu offset. Inicialmente tem 8 bytes (espaço livre e 0 registros), podendo crescer até não caber mais registros. Supondo que tenhamos 3 registros no bloco, nosso header terá o seguinte formato: 4 bytes para espaço livre, 4 bytes para o número de registros (que tem valor 3), e depois 12 bytes (3 inteiros) que guardam os valores dos offsets do registro.

Já que nosso Header vai aumentar de tamanho de acordo com a inserção de registros, inserimos os registros na última posição livre do bloco, assim garantindo que não vamos sobrescrever um registro ou dado do header. A figura 2 mostra um exemplo onde foi inserido 2 registros, primeiro o Reg1 e depois o Reg2.

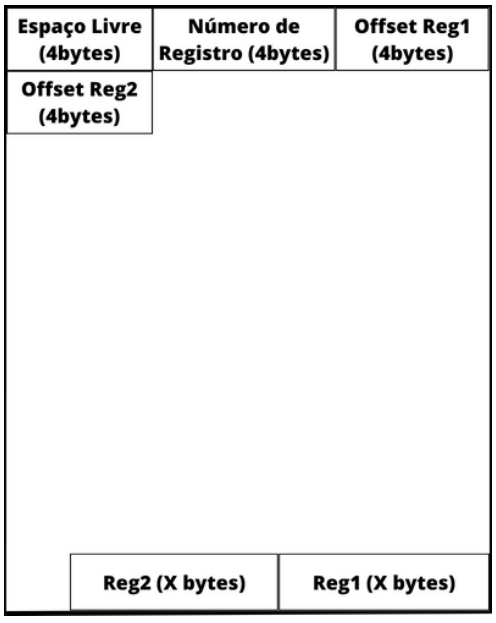

Figure 2. Exemplo de Bloco com 2 Registros

1.3. Bucket

O Bucket é uma abstração usada na Hash para que agrupemos x blocos, assim facilite o acesso e inserção/busca por registros. O nosso bucket é composto por 20 blocos, na implementação ele numera os blocos de 0 a 19, para que possamos usar essa numeração para formar o endereço do bloco no arquivo de dados, assim não precisamos que ele guarde um valor exato do bloco, mas sim sua numeração dentro dele, que nos ajuda a calcular sua posição.

1.4. Tabela Hash

A tabela Hash é similar a qualquer outra já implementada, ela recebe um chave que é formada pela hashFunction, que por sua vez recebe um parâmetro do registro (no nosso caso o ID) e forma a chave hash. Após acessarmos a tabela Hash com a nossa chave recebemos o bucket correspondente àquela posição da Hash, sendo assim fazemos busca e inserção do registro no Bucket. Nossa Hash tem tamanho 15000 e é um vetor do tipo Bucket. Na nossa implementação usamos a chave da Hash para calcularmos a posição exata do bloco que queremos encontrar, junto com o que foi explicado no bucket, dessa forma, sabendo qual a chave da Hash, qual bloco do bucket queremos, conseguimos calcular sua posição exata no arquivo de dados.

1.5. Arquivo de Dados

O arquivo de dados é um arquivo binário que guarda todos os nossos dados. Nós lemos ele de bloco em bloco, ou seja, a cada 4096 bytes, uma vez que esse intervalo de memória armazena um bloco de dados com seu header e os registros. Esse arquivo é inicializado com seu header contendo o espaço livre (4096 - 8 = 4088 byte) e o número de registros, que é 0, os resto dos bytes contem o valor 0, pois não alocamos nada neles. O espaço livre inicia com 4088 bytes, pois mesmo um bloco vazio contém os dois inteiros citados anteriormente.

1.6. Árvore B+

A árvore B+ guarda as chaves (ID ou Título) e os offsets para os blocos do arquivo de dados e está gravada em um arquivo binário. No caso da árvore B+, há dois tipos de nós: internos, que guardam somente as chaves e os ponteiros para os nós filhos; e as folhas, que guardam as chaves, os offsets para os blocos no arquivo de dados e um outro ponteiro para a próxima folha.

Além disso, outra informação importante sobre a árvore é a sua ordem m, ou seja, quantas 2m chaves a árvore guarda por nó. No caso da árvore construída nesse trabalho, a ordem é de m = 255, indicando que a árvore pode conter até 510 chaves e no mínimo 255 chaves a fim de manter o balanceamento. Isso se deve por conta do tamanho de bloco adotado (4096 bytes) e de alguns descontos de tamanho por conta da estrutura de nó.

Portanto, a busca por determinado registro pode se dar através através da recuperação do offset do bloco contendo esse registro por meio de uma busca nesse arquivo de índice montado pela árvore B+.

1.7. Arquivo de Índices

O arquivo de índices é um arquivo binário que guarda a árvore contendo as chaves e os offsets para o arquivo de dados. Ele foi montado a partir da árvore construída em

memória principal, através de um caminhamento em largura pela mesma, gravando nó por nó a partir da raiz.

2. Quais fontes formam cada programa

2.1. cte.hpp

Não utiliza nenhuma fonte.

2.2. registro.hpp

- string

2.3. bloco.hpp

- iostream
- vector
- cstring
- cte.hpp
- registro.hpp

2.4. bucket.hpp

- bits/stdc++.h
- bloco.hpp

2.5. hash.hpp

- bucket.hpp

2.6. bpt.hpp

- iostream
- cte.hpp
- bits/stdc++.h

2.7. upload.cpp

- hash.hpp

2.8. findrec.cpp

- hash.hpp

2.9. seek1.cpp

- hash.hpp

2.10. seek2.cpp

- hash.hpp

3. As funções que cada fonte contém, quem desenvolveu e o que ela faz

3.1. cte.hpp

Não possui funções.

3.2. registro.hpp

- `sizeRegistro(Registro *registro)`
  Desenvolvido por: José Mateus
  Papel: Essa função aceita um ponteiro para uma estrutura **Registro** e atualiza o campo **tamanhoRegistro** da estrutura para refletir o tamanho total do registro em bytes. Ela faz isso somando os tamanhos em bytes de todos os campos da estrutura Registro. Note que, para os campos de string, ela soma o número de caracteres na string mais 1 (para o caractere nulo '\0' que marca o fim da string).
- `printRegistro(Registro registro)`
  Desenvolvido por: José Mateus
  Papel: Essa função aceita uma estrutura **Registro** e imprime todos os campos do registro na saída padrão (normalmente, o console). Note que ela não imprime o campo **tamanhoRegistro**.
- `createRegistro(int id = 0, int ano = 0, int citacoes = 0, string atualizacao = "NULL", string titulo = "NULL", string autores = "NULL", string snippet = "NULL")`
  Desenvolvido por: José Mateus
  Papel: Essa função cria um novo registro. Ela aceita sete argumentos, todos com valores padrão, que são usados para inicializar os campos correspondentes na estrutura **Registro**. Depois de inicializar todos os campos, ela chama a função `sizeRegistro` para atualizar o campo **tamanhoRegistro** e, em seguida, retorna a estrutura Registro.

3.3. bloco.hpp

- `createBloco()`
  Desenvolvido por: José Mateus
  Papel: Esta função cria um novo bloco. Ela inicializa os campos do cabeçalho do bloco e define todos os bytes do bloco como 0.
- `extrairHeader(char *blocoBytes)`
  Desenvolvido por: José Mateus
  Papel: Esta função extrai as informações do cabeçalho de um bloco a partir de um vetor de bytes. Ela copia as informações do vetor de bytes para a estrutura do bloco e retorna o bloco.
- `printBloco(Bloco *bloco)`
  Desenvolvido por: José Mateus
  Papel: Esta função imprime as informações do cabeçalho de um bloco, incluindo o espaço livre, o número de registros e os offset dos registros.
- `insertRegistroBloco(Bloco *bloco, Registro &registro)`
  Desenvolvido por: José Mateus
  Papel: Esta função insere um registro em um bloco. Ela verifica se há espaço suficiente no bloco, copia as informações do registro para o bloco e atualiza o cabeçalho do bloco. Retorna true se a inserção foi bem-sucedida e false caso contrário.
- `searchRegistroBloco(Bloco * bloco, int registroId)`
  Desenvolvido por: José Mateus

Papel: Esta função procura um registro em um bloco com base em um ID de registro. Ela percorre os offsets dos registros no cabeçalho do bloco e compara o ID do registro com o ID fornecido. Se encontrar um registro correspondente, extrai as informações do registro do bloco e retorna o registro. Se não encontrar um registro correspondente, retorna NULL.

3.4. bucket.hpp

- `createBucket(ofstream &binDataFile)`
  Desenvolvido por: José Mateus
  Papel: Essa função cria um novo bucket, que é uma estrutura usada para armazenar blocos de dados. Cada bucket tem um tamanho definido por **BUCKET_SIZE** e é inicializado com blocos vazios, que são escritos no arquivo de dados binários fornecido. A função retorna um ponteiro para o bucket criado.
- `loadBloco(int blockAddress, ifstream &binDataFile)`
  Desenvolvido por: José Mateus
  Papel: Esta função carrega um bloco de dados a partir de um endereço específico em um arquivo de dados binário. O bloco de dados é lido no buffer e o cabeçalho é extraído para criar a estrutura de bloco. A função retorna um ponteiro para a estrutura do bloco criado.
- `writeRegistroBucket(int blockAddress, Bloco *bloco, Registro registro, ofstream &dataFileWrite)`
  Desenvolvido por: José Mateus
  Papel: Esta função escreve um registro em um bloco específico em um bucket. Primeiro, o registro é inserido no bloco usando a função `insertRegistroBloco`. Em seguida, o bloco é escrito no arquivo de dados no endereço do bloco fornecido. A função retorna verdadeiro se a operação de gravação for bem-sucedida.

3.5. hash.hpp

- `createHash(ofstream &binDataFile)`
  Desenvolvido por: José Mateus
  Papel: Essa função cria uma nova tabela hash. Ela percorre cada índice do array da tabela hash e atribui um novo bucket a cada um (usando a função `createBucket`). Retorna um ponteiro para a nova tabela hash.
- `hashFunction(int key)`
  Desenvolvido por: José Mateus
  Papel: Essa função é a função de hash, baseada na MurmurHash, utilizada para determinar a posição de um item na tabela hash, dado uma chave. Ela usa uma combinação de operações bit a bit para gerar um hash único para a chave dada e depois aplica o operador módulo para garantir que a chave caiba dentro do tamanho da tabela hash.
- `insertRegistroHashTable(Registro registro, ofstream &dataFileWrite, ifstream &dataFileRead)`
  Desenvolvido por: José Mateus
  Papel: Essa função tenta inserir um novo registro na tabela hash. Primeiro, ela usa a função de hash na ID do registro para encontrar o bucket apropriado. Então,

ela percorre cada bloco no bucket para encontrar um espaço que possa acomodar o registro. Se encontrar espaço suficiente, ela insere o registro no bloco e retorna true. Caso contrário, ela retorna false.

- `searchRegistroById(int registroId, ifstream &dataFileRead)`
  Desenvolvido por: José Mateus
  Papel: Essa função busca um registro na tabela hash usando a ID do registro como chave. Ela usa a função de hash para encontrar o bucket apropriado e então percorre cada bloco no bucket para encontrar o registro desejado. Se o registro for encontrado, ele é retornado; caso contrário, a função retorna NULL.

3.6. bpt.hpp

- `create_node()`
  Desenvolvido por: Giovanna
  Papel: Cria um novo nó, alocando espaço para o mesmo e inicializando seus valores.
- `create_leaf()`
  Desenvolvido por: Giovanna
  Papel: Cria um novo nó, alocando espaço para o mesmo e inicializando seus valores. No entanto, o marca como sendo do tipo folha.
- `create_tree(int key, block *b)`
  Desenvolvido por: Giovanna
  Papel: Cria uma árvore B+.
  - key: primeira chave a ser alocada na árvore.
  - *b: ponteiro para a estrutura block que contém o offset.

  O método cria um nó folha (raiz inicial), através da função `create_leaf` que possui somente a chave e o offset passados por parâmetro.
- `create_block(int offset)`
  Desenvolvido por: Giovanna
  Papel: Cria um objeto do tipo block, tomando como parâmetro o offset passado.
  - offset: offset de um bloco do arquivo de dados.
- `insert(node *root, int key, int offset)`
  Desenvolvido por: Giovanna
  Papel: Insere uma nova chave juntamente com um offset na árvore.
  - *root: raiz da árvore.
  - key: chave a ser inserida.
  - offset: offset a ser inserido.

  Cria um objeto do tipo block para alocar o offset e acha a folha onde a chave deve ser inserida com o offset, por meio da função `find_leaf`. Logo depois, verifica se ainda tem espaço para inserir na folha retornada. Caso tenha, insere a chave na folha `(insert_leaf)`, caso não, insere a chave após um split.
- `find_leaf(node *root, int key)`
  Desenvolvido por: Giovanna
  Papel: Acha a folha onde a nova chave deve ser inserida.
  - *root: ponteiro para a raiz da árvore.
  - key: chave a ser inserida.

- `insert_leaf(node *leaf, int key, block* b)`
  Desenvolvido por: Giovanna
  Papel: Insere uma nova chave em uma folha que ainda possui espaço.
  - *leaf: ponteiro para a folha onde a chave será inserida.
  - key: chave a ser inserida.
  - *b: ponteiro para o block que vai ser inserido.
- `insert_leaf_after_split(node *root, node *leaf, int key, block* b)`
  Desenvolvido por: Giovanna
  Papel: Insere uma nova chave em uma folha que já está cheia.
  - *root: ponteiro para a raiz da árvore.
  - *leaf: ponteiro para a folha onde a chave será inserida.
  - key: chave a ser inserida.
  - *b: ponteiro para o block que vai ser inserido.
    Realiza o split `split` da folha, origina duas folhas e realiza a inserção da chave com o offset. A chave do meio é copiada para o nó pai das duas folhas originadas do split `(insert_parent)`
    .
- `insert_parent(node *root, node *left, int key, node* right)`
  Desenvolvido por: Giovanna
  Papel: Insere a cópia de uma chave no nó pai.
  - *root: ponteiro para a raiz da árvore.
  - *left: ponteiro para o nó filho da esquerda.
  - key: chave a ser copiada.
  - *right: ponteiro para o nó filho da direita.
    Verifica se já existe algum nó pai. Caso não exista, cria uma nova raiz e insere a cópia da chave nela `(insert_new_root)`. Caso exista, e tenha espaço no pai, insere direto `(insert_node)`. No caso de não existir espaço no pai, executa um split `(insert_node_after_split)`.
- `insert_node(node *root, node *n, int left_index, int key, node* right)`
  Desenvolvido por: Giovanna
  Papel: Insere uma chave em um nó que tem espaço.
  - *root: ponteiro para a raiz da árvore.
  - *n: ponteiro para o nó em que a chave será inserida.
  - left_index: índice do ponteiro do nó mais à esquerda.
  - key: chave que vai ser inserida no nó.
  - *right: ponteiro para o nó filho da direita.
- `insert_node_after_split(node *root, node *old_node, int left_index, int key, node* right)`
  Desenvolvido por: Giovanna
  Papel: Insere uma nova chave em um nó que já está cheio.
  - *root: ponteiro para a raiz da árvore.
  - *old_node: ponteiro para o antigo nó que estava cheio.
  - left_index: índice do ponteiro do nó mais à esquerda.
  - key: chave a ser inserida.

  - *right: ponteiro para o nó filho da direita.
    Realiza o split `split` do nó, origina dois nós e realiza a inserção da chave. O nó do meio é copiado para o nó pai `(insert_parent)`
    .
- `insert_new_root(node *left, int key, node* right)`
  Desenvolvido por: Giovanna
  Papel: Cria uma nova raiz, no caso de após um split, não houver nó pai.
  - *left: ponteiro para o filho da esquerda.
  - key: chave a ser inserida.
  - *right: ponteiro para o nó filho da direita.
- `get_left_index(node *parent, node* left)`
  Desenvolvido por: Giovanna
  Papel: Retorna a posição do nó mais à esquerda.
  - *parent: ponteiro para o pai.
  - *right: ponteiro para o nó filho da esquerda.
- `cut(int size)`
  Desenvolvido por: Giovanna
  Papel: Retorna a posição onde deverá ser feito o split.
  - size: quantidade de ponteiros.
- `pathToLeaves(node *const root, node *child)`
  Desenvolvido por: Giovanna
  Papel: Retorna o nível de determinado filho em relação à raiz.
  - *root: ponteiro para a raiz.
  - *right: ponteiro para o nó filho.
- `imprime_node(NodeDisk no)`
  Desenvolvido por: Marcos
  Papel: Imprime os detalhes de um nó.
  - no: nó a ser impresso.
- `search_key(unsigned int key, unsigned long pos, unsigned int *n_faccess, FILE *arq)`
  Desenvolvido por: Marcos
  Papel: Efetua a busca de uma chave numa árvore B+ salva num arquivo em disco.
  - key: A chave a ser buscada.
  - pos: Variável de controle da posição atual no arquivo (nodo da árvore).
  - *n_facess: Variável contadora de acessos ao disco.
  - *arq: Ponteiro para o arquivo.
- `gravaTree(node *root, unsigned long parent, FILE *arq)`
  Desenvolvido por: Marcos
  Papel: Recebe a raiz de uma árvore B+ na memória e grava seus nodos num arquivo.
  - root: Nó raiz da árvore.
  - parent: Offset do nodo pai no arquivo, parâmetro de controle recursivo da função. na chamada inicial deve receber zero.
  - arq: Ponteiro para o arquivo onde a árvore deve ser salva.
- `gravaTree(node *root, unsigned long parent, FILE *arq)`
  Desenvolvido por: Marcos
  Papel: Recebe a raiz de uma árvore B+ na memória e grava seus nodos num arquivo.

  - root: Nó raiz da árvore.
  - parent: Offset do nodo pai no arquivo, parâmetro de controle recursivo da função. na chamada inicial deve receber zero.
  - arq: Ponteiro para o arquivo onde a árvore deve ser salva.
- `print_tree(node const* root)`
  Desenvolvido por: Giovanna
  Papel: Imprime o formato de uma árvore com suas chaves.
  - *root: ponteiro da raiz da árvore.

3.7. upload.cpp

- `get_next_field(FILE *arquivo, char field[], string pattern)`
  Desenvolvido por: Giovanna e Marcos
  Papel: Monta e retorna um campo do registro. Faz isso identificando casos especiais ou padrões do documento de entrada.
  Este método é responsável por ler o próximo campo de um arquivo CSV. Ele aceita três argumentos:
  - FILE *arquivo: Este é o arquivo que será lido.
  - char field[]: Este é um array de caracteres onde o campo lido será armazenado.
  - string pattern: Esta é a sequência de caracteres que delimita o final de um campo no arquivo CSV.

  O método lê o arquivo caractere por caractere até encontrar o padrão especificado no argumento pattern. Quando o padrão é encontrado, a leitura do campo é interrompida. Se o campo lido estiver vazio, o método preenche o array field com a string ”NULL”. A função retorna true se atingir o final do arquivo (EOF) durante a leitura e false caso contrário.
- `main(int argc, char *argv[])`
  Desenvolvido por: Giovanna e Marcos
  Papel: A função principal do programa. Ela abre um arquivo especificado pelo usuário, lê os registros desse arquivo e os insere em um arquivo binário organizado por hash. A tabela de hash é inicialmente criada e, em seguida, cada registro é extraído do arquivo de entrada usando a função `get_next_field()`, é criada uma estrutura **Registro** a partir desses dados e inserida na tabela de hash. O processo continua até que todos os registros tenham sido lidos e inseridos. No final, todos os arquivos abertos são fechados.

3.8. findrec.cpp

- `main(int, char const **argv)`
  Desenvolvido por: José Mateus
  Papel: Esta é a função principal que é executada quando o programa é iniciado. Ela verifica se o número correto de argumentos foi fornecido na linha de comando. Se o número correto de argumentos foi fornecido, ela abre um arquivo de dados binário, pesquisa um registro com o ID fornecido, com a função `searchRegistroById`, imprime os detalhes do registro se encontrado, com a função `printRegistro`, e, em seguida, fecha o arquivo.

3.9. seek1.cpp

- `main(int, char const **argv)`
  Desenvolvido por: José Mateus
  Papel: Esta função principal é executada ao iniciar o programa. Ela utiliza a função `search_key` (implementada por Marcos) para pesquisar uma chave no arquivo de índice, contando também o número de acessos. Caso a chave seja encontrada, o bloco de dados correspondente é carregado do arquivo de dados com a função `loadBloco` e um registro específico é buscado dentro deste bloco usando a função `searchRegistroBloco`. Por fim, os detalhes do registro são impressos com a função `printRegistro`, e ambos os arquivos são fechados.

3.10. seek2.cpp

- `main(int, char const **argv)`
  Desenvolvido por: José Mateus
  Papel: Utiliza a função `search_key` (implementada por Marcos) para encontrar a posição de um registro baseado no título, fornecido como argumento, utilizando o arquivo de índice da B+ tree. Caso o registro seja encontrado, o bloco de dados correspondente é carregado, com a função `loadBloco`, e todos os registros do bloco são lidos. A função procura o registro com o título correspondente ao argumento, e se encontrado, imprime os detalhes com a função `printRegistro`. Por fim, os arquivos são fechados.

4. Ambiente de execução

4.1. Sistema operacional

O trabalho prático foi desenvolvido no ambiente Ubuntu 22.04 LTS.

4.2. Linguagem de programação

O trabalho prático foi implementado utilizando C \C++ e compilado usando G++ 11.