



Documentação TP2 - BDI

Alice Teles Lucena

Igor de Souza Lima

Nathália Rodrigues Machado dos Santos

1. Casos Especiais do Arquivo de Entrada

O arquivo de entrada `artigo.csv` possui alguns casos especiais de registros. Vamos mencionar nesta seção esses casos especiais e o que fizemos para lidar com esses casos.

Primeiramente, podemos observar um padrão nos registros do arquivo de entrada:

```
1 "1";"Poster: 3D sketching and flexible input for surface design: A case study.";2013;"Anamary Leal|Doug A. Bowman";0";2016-07-28 09:36:29";"Poster: 3D sketching
2 "2";"Poster: Portable integral photography input/ output system using tablet PC and fly's eye lenses.";2013;"Yusuke Kawano|Kazuhiya Yanaka";0";2016-07-28 09:36:4
3 "3";"Poster: Real-time markerless Kinect based finger tracking and hand gesture recognition for HCI.";2013;"Arun Kulshreshtha|Christopher Zorn|Joseph J. LaViola Jr.
4 "4";"Poster: Real time hand pose recognition with depth sensors for mixed reality interfaces.";2013;"Byungkyu Kang|Mathieu Rodrique|Tobias H&ouml;llerer|Hwasup Lim
5 "5";"Design and implementation of an immersive virtual reality system based on a smartphone platform.";2013;"Anthony Steed|Simon Julier";12";2016-10-03 21:10:56"
6 "6";"Poster: Lifted road map view on windshield display.";2013;"Takaya Kawamata|Itaru Kitahara|Yoshinari Kameda|Yuichi Ohta";1";2016-10-03 21:34:50";"Poster: Lif
7 "7";"Poster: Comparing usability of a single versus dual interaction metaphor in a multitask healthcare simulation.";2013;"Lauren Cairco Dukes|Jeffrey W. Bertrand|
8 "8";"Poster: Creating a user-specific perspective view for mobile mixed reality systems on smartphones.";2013;"Yuki Matsuda|Fumihisa Shibata|Asako Kimura|Hideyuki
9 "9";"Poster: Markerless fingertip-based 3D interaction for handheld augmented reality in a small workspace.";2013;"Huidong Bai|Lei Gao|Mark Billingham";5";2016
10 "10";"Poster: A wearable augmented reality system with haptic feedback and its performance in virtual assembly tasks.";2013;"Kazuya Murakami|Ryo Kiyama|Takuji Naru
```

Cada linha do arquivo consiste em 1 registro, onde cada linha está dividida em campos, por meio do ponto e vírgula (;). Estes campos estão, em ordem, de acordo com a **Tabela 1**, onde o primeiro campo corresponde ao ID, o segundo ao Título, o terceiro ao Ano, e assim por diante.

Tabela 1. Estrutura 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

Sabendo disso, podemos começar a tratar dos casos especiais encontrados no arquivo:

1. Campos Faltantes

Há alguns registros que não possuem certos campos, como título, autores e snippet. Nesses casos, o registro ainda é inserido e os campos faltantes são dados como valores nulos.

2. Citações Negativas

Alguns registros possuem citações negativas. Nesses casos, apenas transformamos o valor das citações em zero.

3. Ponto e vírgula dentro de um campo

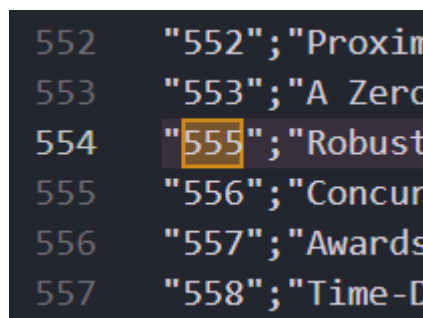
Alguns registros também possuem ponto e vírgula, que é o caractere responsável por separar os campos, como parte do campo. Por exemplo, um título pode possuir ponto e vírgula dentro dele. Também tratamos estes casos, de forma que consideramos o ponto e vírgula como parte do texto e não como um separador de campo.

4. Registros que ocupam 2 linhas

Temos 4 exceções de registros que ocupam 2 linhas, ao invés de 1. São eles os registros de ID: 368004, 424931, 500462 e 738289. Eles ocupam 2 linhas pois o campo Título deles tem uma quebra de linha e acaba sendo dividido em 2 linhas separadas. Assim, é necessário ler o campo Título da linha seguinte e concatenar com o campo Título da linha atual. Também tratamos esses casos ao ler o arquivo de entrada.

5. ID's faltantes

Esse caso é um dos mais importantes que precisamos ressaltar. Apesar dos registros estarem com os ID's em sequência no arquivo de entrada, tem ID's que são "pulados" e não constam no arquivo. Por exemplo, como mostra a imagem a seguir, o ID 554 não existe no arquivo de entrada, do ID 553 já pula para o ID 555.



```
552 "552"; "Proxim
553 "553"; "A Zero
554 "555"; "Robust
555 "556"; "Concur
556 "557"; "Awards
557 "558"; "Time-D
```

Isso irá refletir na quantidade de registros presentes no arquivo de entrada. Como podemos ver na imagem abaixo, que mostra o final do arquivo, **o ID do último registro é 1549146**. Entretanto, a quantidade de registros é afetada pelos ID's pulados, como apresenta a coluna à esquerda na imagem, que mostra que temos 1.021.443 registros. No entanto, contando que temos 4 casos especiais de

registros que ocupam duas linhas no arquivo de entrada, **temos na verdade 1.021.439 registros no arquivo.**

```
1021437 "1549140";"Developing crit
1021438 "1549141";"Toward a formal
1021439 "1549142";"Instantiating g
1021440 "1549143";"Flush: a system
1021441 "1549144";"Invariants come
1021442 "1549145";"The experience o
1021443 "1549146";"Ei-Łcient Symbo
1021444
```

2. Estrutura dos Arquivos de Dados e de Índices

Sobre as estruturas dos arquivos, tanto de dados quanto de índices, há padrões que foram definidos, e são mantidos em todos os arquivos. Por exemplo, para armazená-los em memória secundária, salvamos eles em **arquivos binários**. Além disso, foi definido o tamanho de bloco fixo de **4096 bytes**, em todos eles.

2.1. Estrutura do Arquivo de Dados

O arquivo de dados é salvo no arquivo binário `arquivo_dados.bin` estruturado com os seguintes padrões:

- blocos de tamanho fixo de **4096 bytes**.
- 1 bucket da tabela hash aponta para 2 blocos do arquivo de dados.
- 2 registros por bloco.
- registros de tamanho fixo.
- alocação não espalhada.

A **Figura 1** mostra como fica a estruturação do nosso arquivo de dados. Nela, podemos observar que o primeiro bloco sempre começa no byte 0 do arquivo, onde a primeira informação armazenada é o ID do primeiro registro. Podemos observar também que, cada bloco contém 2 registros, e as informações de cada registro estão organizadas sequencialmente conforme a **Tabela 1**.

Dessa forma, ler o arquivo de dados de 4096 bytes em 4096 bytes garante que estaremos lendo um bloco por vez, e que podemos ter até 2 registros dentro de cada bloco.

Outro ponto importante sobre a parte do arquivo de dados, é que ele é **organizado por hashing**, ou seja, ele foi sendo gerado conforme as chaves (*ID's*) eram mapeadas, de acordo com uma função hash, para os buckets da tabela.

Neste trabalho, foi definido que cada bucket iria apontar para 2 blocos do arquivo de dados, conforme mostra a **Figura 2**. Dessa forma, cada bucket pode conter até 4 registros, já que cada bloco guarda até 2 registros.

Figura 1. Arquivo de dados

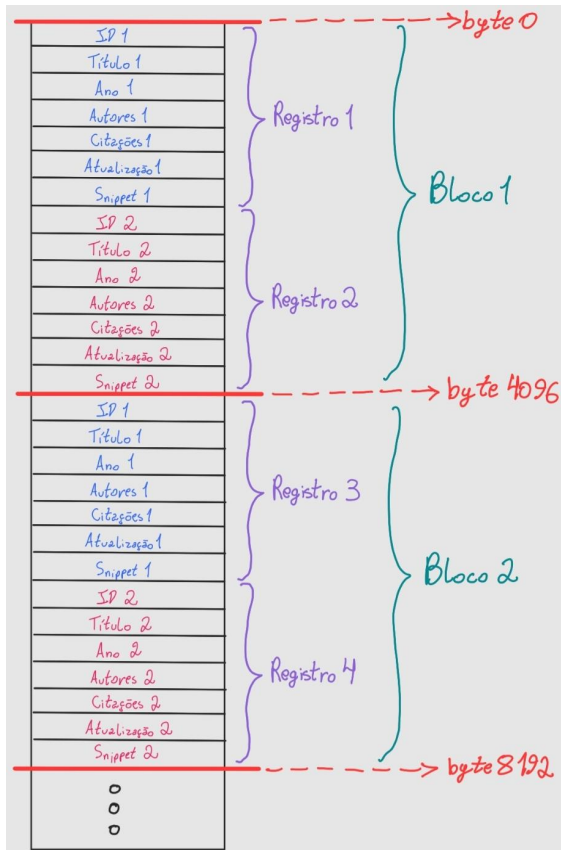
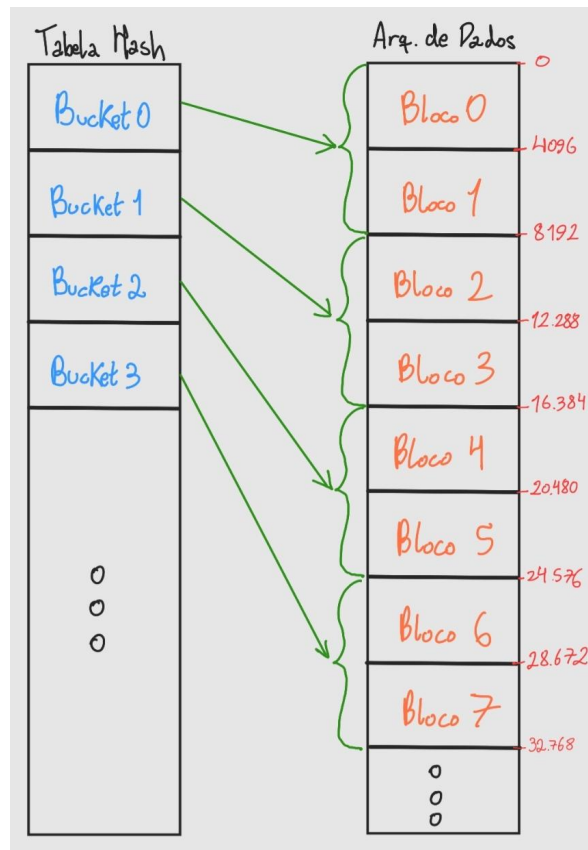


Figura 2. Buckets e Blocos



Por fim, uma última observação importante, neste trabalho nós já tratamos os possíveis casos de buckets e blocos de overflow. Nossa tabela hash e nosso arquivo de dados são construídos justamente para que **não ocorram esses casos de overflow**.

Ou seja, como no arquivo de entrada `artigo.csv` temos 1.021.439 registros, dividindo isso por 4, ficará aproximadamente 255.360 buckets. Este é o número de buckets que temos em nossa tabela hash, justamente para que não ocorra overflow e todos os registros caibam perfeitamente nos blocos.

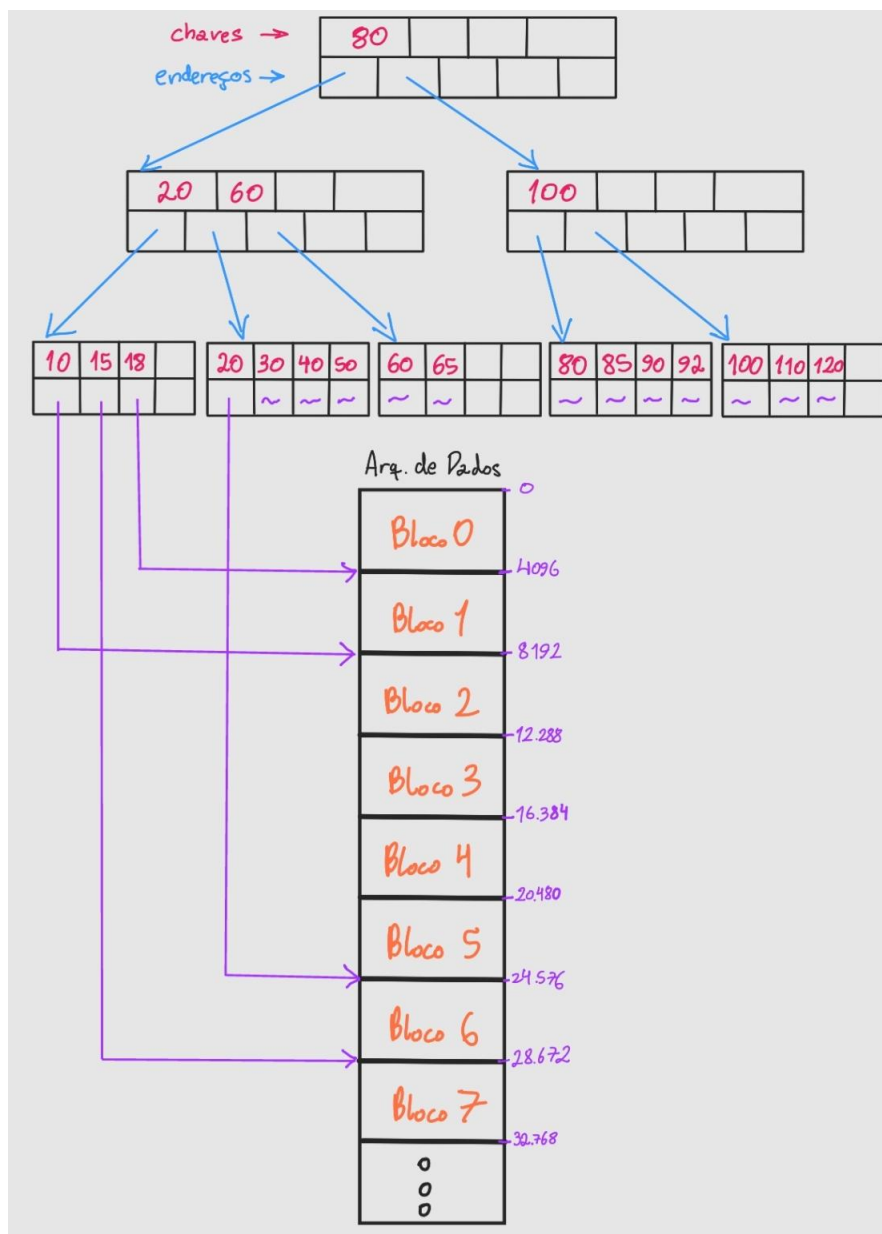
2.2. Estrutura do Arquivo de Índice Primário

O arquivo de índice primário é salvo no arquivo binário `arquivo_indice_primario.bin` estruturado com os seguintes padrões:

- blocos de tamanho fixo de **4096 bytes**.
- 1 nó da árvore B+ corresponde a 1 bloco no arquivo de índice.

Na **Figura 3**, vemos um exemplo de uma árvore B+ de ordem 4 ($m = 4$), que possui 3 nós internos, um deles sendo a raiz, e 5 nós folhas. Nos nós folhas, temos as chaves (*ID's*) e seus respectivos endereços do arquivo de dados, que correspondem aos endereços dos blocos no arquivo de dados onde está aquela respectiva chave.

Figura 3. Exemplo de Árvore B+



Tendo como exemplo a árvore da figura acima, vamos explicar melhor como nosso arquivo de índice primário está estruturado.

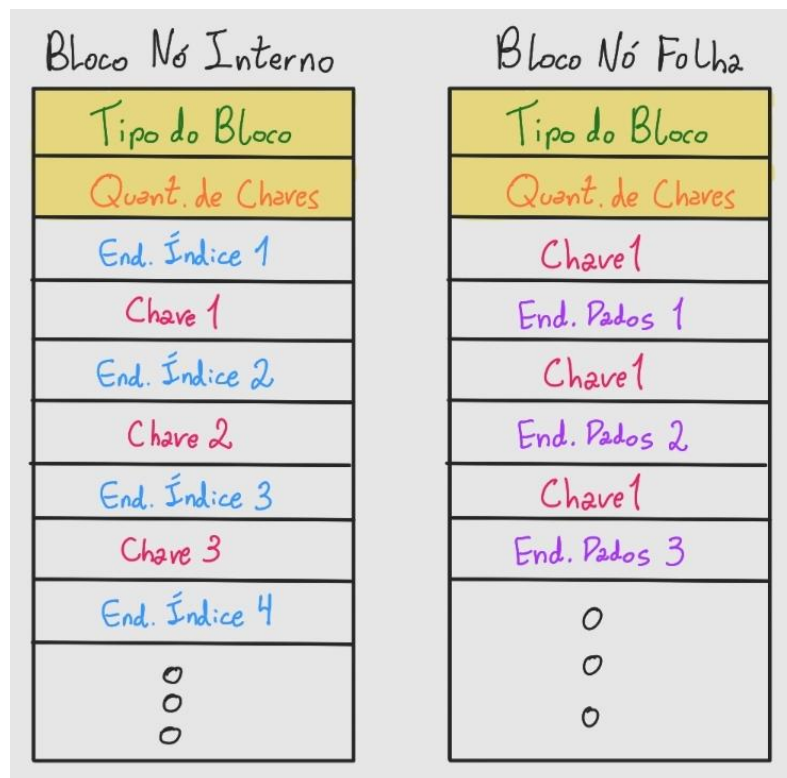
Primeiramente, temos 2 tipos de blocos no arquivo de índice:

1. **blocos internos:** correspondem a um nó interno da árvore.
2. **blocos folha:** correspondem a um nó folha da árvore.

Na **Figura 4** podemos ver como é a estrutura interna de cada um desses blocos. Em ambos, temos a presença de um **cabeçalho** (representado na figura pelos retângulos em amarelo), onde o primeiro campo é um inteiro que indica o tipo daquele bloco (0 = interno, 1 = folha) e o segundo campo é um inteiro que indica a quantidade de chaves presentes no bloco naquele instante.

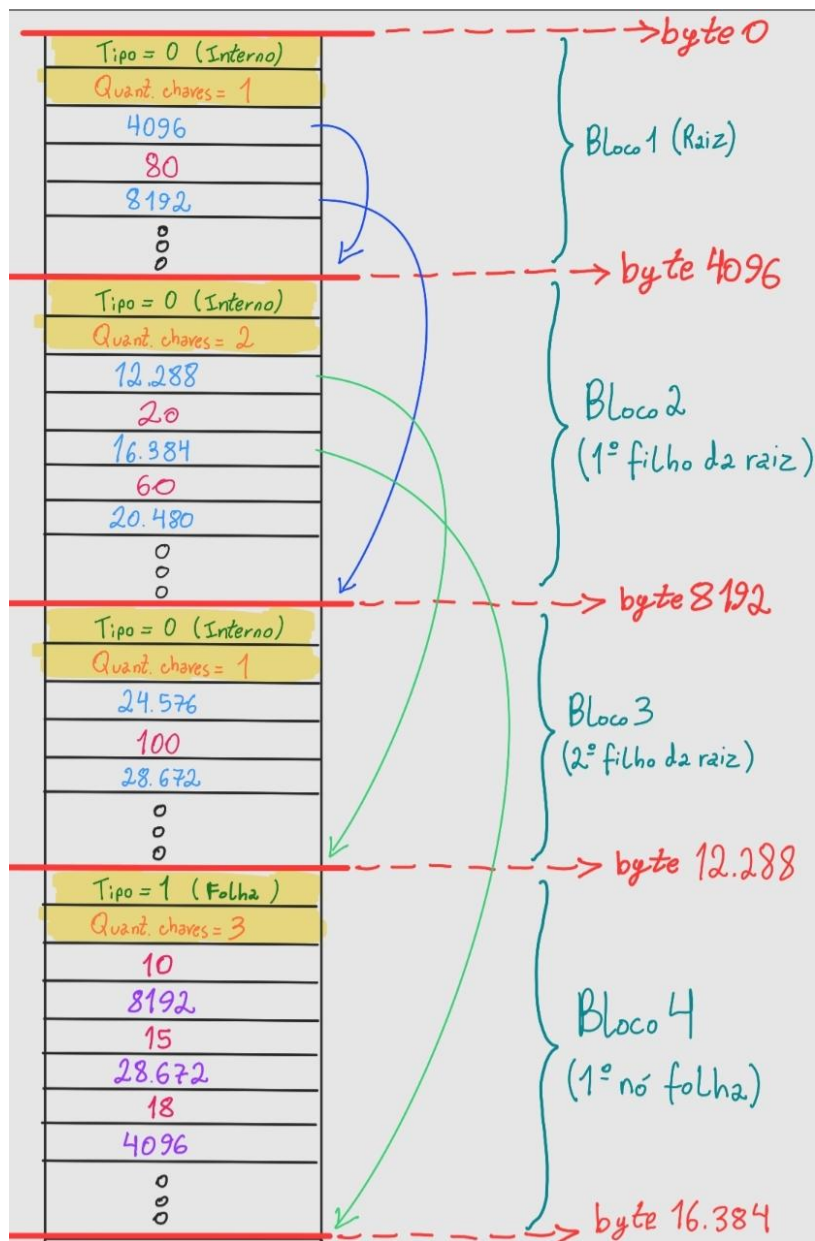
O restante do bloco já é correspondente às chaves e endereços. Em ambos, as chaves são os ID's, o que muda é o endereço. No caso do bloco interno, os endereços são dos blocos no arquivo de índice. Enquanto no caso do bloco folha, os endereços já são dos blocos no arquivo de dados.

Figura 4. Blocos Interno e Folha (Primário)



Dessa forma, a **Figura 5** mostra de maneira mais detalhada como vai ficar parte do nosso arquivo de índice primário para a árvore de exemplo da **Figura 3**.

Figura 5. Arquivo de Índice Primário



Como podemos ver, o primeiro bloco já no byte 0 do arquivo sempre será a raiz da árvore. Seguido desse bloco, temos o bloco 2 correspondente ao primeiro filho da raiz (byte 4096), e logo depois o bloco do segundo filho da raiz (byte 9182). Somente após esse, temos o primeiro bloco folha da árvore (byte 12288).

Assim, podemos observar um padrão:

- A raiz sempre estará no byte 0 do arquivo.
- Logo após, estarão em ordem, os blocos filhos da raiz.
- E assim sucessivamente, de maneira que os blocos de um mesmo nível da árvore estejam alocados em sequência.
- Ou seja, a distribuição dos blocos no arquivo pode ser comparada com a técnica de **busca em largura (BFS)**.

Podemos também observar que, os blocos internos possuem endereços do arquivo de índice (representados pela cor azul na **Figura 5**), que correspondem aos blocos que possuem chaves menores ou maiores do que a chave a que ele se refere. Por exemplo, no bloco 1, temos a chave 80 e os endereços 4096 e 8192. Esses endereços correspondem, respectivamente, aos blocos no arquivo de índice que guardam chaves menores e maiores do que a chave 80.

Já os blocos folha, possuem endereços do arquivo de dados (representados pela cor roxo na **Figura 5**), que correspondem ao bloco no arquivo de dados onde aquela chave está contida. Por exemplo, na **Figura 3** vemos que a chave 15 está apontando para o bloco 7 do arquivo de dados, que está no endereço 28.672. Assim, no arquivo de índice da **Figura 5** podemos ver que este endereço está sendo guardado logo após a chave 15, formando o par (chave, endereço).

Por último, uma observação muito importante é que, o arquivo de índice primário cresce conforme a árvore cresce; ou seja, a cada nova inserção na árvore, alteramos também o arquivo de índice. Por mais que isso não seja um requisito do trabalho, optamos por fazer desta forma, pois, em um banco de dados “de verdade”, a chance de não ter memória suficiente para guardar de uma vez todos os nós (blocos) do índice, é grande. Dessa forma, o nó (bloco) já pode ser armazenado em disco assim que ficar “pronto”.

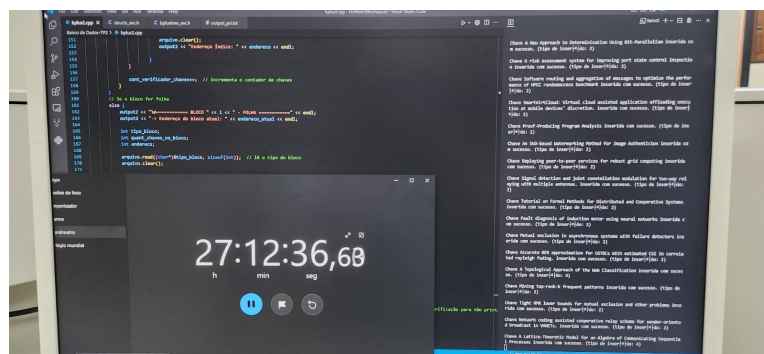
2.3. Estrutura do Arquivo de Índice Secundário

O arquivo de índice secundário é salvo no arquivo binário `arquivo_indice_secundario.bin` estruturado com os seguintes padrões:

- blocos de tamanho fixo de **4096 bytes**.
- 1 nó da árvore B+ corresponde a 1 bloco no arquivo de índice.

A estrutura do arquivo de índice secundário é parecida com a do arquivo de índice primário. Entretanto, pelo fato da chave ser uma string (*Título*) e não um número inteiro (*ID*), enfrentamos alguns problemas durante o processo de criação do arquivo de índice.

Inicialmente, estávamos tentando realizar da mesma forma que foi feita no primário, com o arquivo de índice crescendo junto com a árvore. Entretanto, o código estava com um custo muito alto, demorando muito para rodar. Como prova, tiramos a seguinte foto em um dos dias de execução do código:



Esse código não terminou de ser executado, então estimamos que a duração dele seria muito mais alta do que mostra na imagem. Chegamos à conclusão de que isso aconteceu provavelmente por conta do maior tamanho de uma string em comparação com um número inteiro. Por isso, as operações do código acabavam ficando muito custosas, já que a cada chave (*Título*) inserida tínhamos que ficar fazendo alterações no arquivo de índice secundário.

Portanto, visto que o fato de gerar o arquivo de índice ao mesmo tempo que a árvore não é um requisito explícito do trabalho, a solução que encontramos para o problema foi: primeiramente geramos a árvore B+ por completo, e somente após isso, escrevemos os nós (blocos) no arquivo de índice secundário. Dessa forma, o código termina sua execução em um tempo muito mais compreensível e aceitável, e o resultado do arquivo de índice gerado permanece o mesmo.

Partindo agora para a estrutura em si do arquivo de índice secundário, como falado anteriormente, não é uma estrutura muito diferente do arquivo de índice primário. A única mudança que teve na estrutura foi a adição de um novo campo dentro dos blocos, como mostra a **Figura 6**. Por se tratar de uma string, é importante que guardemos o tamanho dela em algum local, pois o fato de não guardarmos o tamanho específico e lermos apenas um tamanho fixo estava gerando problemas na escrita e leitura do arquivo binário. Dessa forma, decidimos adicionar um campo que guarda o tamanho da chave (*Título*). Esse campo, como podemos ver na **Figura 6**, fica armazenado sempre antes da sua respectiva chave.

Figura 6. Blocos Interno e Folha (Secundário)

Bloco Nó Interno	Bloco Nó Folha
Tipo do Bloco	Tipo do Bloco
Quant. de Chaves	Quant. de Chaves
End. Índice 1	Tamanho Chave 1
Tamanho Chave 1	Chave 1
Chave 1	End. Páds 1
End. Índice 2	Tamanho Chave 2
Tamanho Chave 2	Chave 2
Chave 2	End. Páds 2
End. Índice 3	Tamanho Chave 3
Tamanho Chave 3	Chave 3
Chave 3	End. Páds 3
End. Índice 4	o
o	o
oo	o

Tendo essas alterações em vista, o resto da estrutura e do funcionamento, tanto da árvore como do arquivo de índice, é semelhante ao descrito na parte do arquivo de índice primário.

3. Programas relacionados a cada arquivo

3.1. Arquivo de Dados

[upload_dados.cpp](#)

Função: Cria a tabela hash, mapeia e faz a carga dos dados de entrada no arquivo de dados. Também faz a leitura do arquivo binário em um arquivo texto para que fique legível a organização dos registros dentro do arquivo de dados.

Autor(es): Nathália

Importa os programas: `structs_dados.h`

Observações Importantes: —

Entrada: A entrada é o arquivo de entrada `artigo.csv` e ele deve ser passado como argumento ao executar o código.

Saída: Além das mensagens de execução que aparecem no terminal, tem como saída o arquivo binário `arquivo_dados.bin` que é o arquivo de dados gerado, e o arquivo de texto `output_hash.txt`, que lê o arquivo de dados gerado e o deixa legível, mostrando informações como tamanho do arquivo, número de blocos, e todos os registros mapeados nos seus respectivos blocos.

Compilação: `g++ upload_dados.cpp -o up_dados`

Execução: `./up_dados artigo.csv`

[findrec.cpp](#)

Função: Faz a busca diretamente no arquivo de dados por um registro com o ID informado. Caso exista, retorna os campos do registro, a quantidade de blocos lidos para encontrá-lo e a quantidade total de blocos do arquivo de dados.

Autor(es): Nathália

Importa os programas: `structs_dados.h`

Observações Importantes: —

Entrada: A entrada é o ID que deseja ser buscado e deve ser passado como argumento ao se executar o programa.

Saída: O resultado da busca é mostrado no próprio terminal, o programa não gera nenhum arquivo adicional.

Compilação: `g++ findrec.cpp -o findrec`

Execução: `./findrec <ID>`

[structs_dados.h](#)

Função: Não tem funções, apenas guarda as structs e as constantes mais importantes utilizadas no `upload_dados.cpp` e no `findrec.cpp`.

Autor(es): Nathália

Importa os programas: —

Observações Importantes: A constante `NUMBER_OF_BUCKETS` deve ser alterada de acordo com o arquivo de entrada. **Se somente for utilizado o**

arquivo de entrada padrão `artigo.csv`, então **nada precisa ser alterado**.

Caso contrário, para saber quantos buckets serão necessários, apenas divida a quantidade de registros do arquivo de entrada por 4 (número de registros por bucket) e coloque esse novo valor na constante `NUMBER_OF_BUCKETS`.

Entrada: —

Saída: —

Compilação: —

Execução: —

3.2. Arquivo de Índice Primário

`upload_pri.cpp`

Função: Faz a leitura dos *ID*'s dos registros presentes no arquivo de dados e realiza a inserção deles na árvore B+, por meio de funções declaradas no `bplustree_pri.h`. Além disso, ao final, também faz a leitura do arquivo de índice gerado e printa o resultado em um arquivo de texto.

Autor(es): Nathália, Alice e Igor

Importa os programas: `bplustree_pri.h` e `structs_pri.h`

Observações Importantes: —

Entrada: A entrada é o arquivo de dados `arquivo_dados.bin` originado pelo hashing e **não é necessário passá-lo como argumento**, ele já é chamado dentro do próprio código pois supõe-se que ele já será gerado primeiro ao compilar e executar o `upload_dados.cpp`

Saída: Além das mensagens de execução que aparecem no terminal, tem como saída o arquivo binário `arquivo_indice_primario.bin` que é o arquivo de índice primário gerado e o arquivo de texto `output_pri.txt`, que lê o arquivo de índice e o deixa legível, mostrando informações como tamanho do arquivo, número de blocos, e todas as chaves e endereços mapeados nos seus respectivos blocos.

Compilação: `g++ upload_pri.cpp -o up_pri`

Execução: `./up_pri`

`bplustree_pri.h`

Função: Possui todas as funções relacionadas à árvore B+ e à construção do arquivo de índice. Vai criando o arquivo de índice primário junto com a árvore; conforme ela vai crescendo, o arquivo de índice também cresce.

Autor(es): Nathália, Alice e Igor

Importa os programas: —

Observações Importantes: A constante `MAX_ID`, presente logo no início do programa, deve ser alterada de acordo com o arquivo de entrada utilizado na parte do hashing para gerar o arquivo de dados. **Se foi utilizado o arquivo de entrada padrão `artigo.csv`, então nada precisa ser alterado**. Caso contrário, substitua a constante `MAX_ID` pelo valor do maior (último) ID

presente no arquivo de entrada. No caso do `artigo.csv`, o último ID do arquivo é o 1549146.

Entrada: —

Saída: —

Compilação: —

Execução: —

[seek1.cpp](#)

Função: Faz a busca diretamente no arquivo de índice por um registro com o ID (chave) informado. Caso exista, busca pelo endereço da chave no arquivo de dados e retorna: os campos do registro, a quantidade de blocos lidos para encontrá-lo e a quantidade total de blocos do arquivo de índice.

Autor(es): Nathália

Importa os programas: `structs_pri.h`

Observações Importantes: —

Entrada: A entrada é o ID que deseja ser buscado e deve ser passado como argumento ao se executar o programa.

Saída: O resultado da busca é mostrado no próprio terminal, o programa não gera nenhum arquivo adicional.

Compilação: `g++ seek1.cpp -o seek1`

Execução: `./seek1 <ID>`

[structs_pri.h](#)

Função: Não tem funções, apenas guarda as structs e as constantes mais importantes utilizadas no `upload_pri.cpp` e no `seek1.cpp`.

Autor(es): Nathália

Importa os programas: —

Observações Importantes: Assim como no `bplustree_pri.h`, a constante `MAX_ID`, presente logo no início do programa, deve ser alterada de acordo com o arquivo de entrada utilizado na parte do hashing para gerar o arquivo de dados. **Se foi utilizado o arquivo de entrada padrão `artigo.csv`, então nada precisa ser alterado.** Caso contrário, substitua a constante `MAX_ID` pelo valor do maior (último) ID presente no arquivo de entrada. No caso do `artigo.csv`, o último ID do arquivo é o 1549146.

Entrada: —

Saída: —

Compilação: —

Execução: —

3.3. Arquivo de Índice Secundário

upload_sec.cpp

Função: Faz a leitura dos *Títulos* dos registros presentes no arquivo de dados e realiza a inserção deles na árvore B+, por meio de funções declaradas no `bplustree_sec.h`. Ele também chama a função que vai gerar o arquivo de índice e, ao final, também faz a leitura do arquivo de índice gerado, printando o resultado em um arquivo de texto.

Autor(es): Nathália, Alice e Igor

Importa os programas: `bplustree_sec.h` e `structs_sec.h`

Observações Importantes: —

Entrada: A entrada é o arquivo de dados `arquivo_dados.bin` originado pelo hashing e **não é necessário passá-lo como argumento**, ele já é chamado dentro do próprio código pois supõe-se que ele já será gerado primeiro ao compilar e executar o `upload_dados.cpp`

Saída: Além das mensagens de execução que aparecem no terminal, tem como saída o arquivo binário `arquivo_indice_secundario.bin` que é o arquivo de índice secundário gerado e o arquivo de texto `output_sec.txt`, que lê o arquivo de índice e o deixa legível, mostrando informações como tamanho do arquivo, número de blocos, e todas as chaves e endereços mapeados nos seus respectivos blocos.

Compilação: `g++ upload_sec.cpp -o up_sec`

Execução: `./up_sec`

bplustree_sec.h

Função: Possui todas as funções relacionadas à árvore B+ e à construção do arquivo de índice. Cria o arquivo de índice secundário após a árvore ter sido gerada por completo.

Autor(es): Nathália, Alice e Igor

Importa os programas: —

Observações Importantes: A constante `MAX_ID`, presente logo no início do programa, deve ser alterada de acordo com o arquivo de entrada utilizado na parte do hashing para gerar o arquivo de dados. **Se foi utilizado o arquivo de entrada padrão `artigo.csv`, então nada precisa ser alterado.** Caso contrário, substitua a constante `MAX_ID` pelo valor do maior (último) ID presente no arquivo de entrada. No caso do `artigo.csv`, o último ID do arquivo é o 1549146.

Entrada: —

Saída: —

Compilação: —

Execução: —

[seek2.cpp](#)

Função: Faz a busca diretamente no arquivo de índice por um registro com o Título (chave) informado. Caso exista, busca pelo endereço da chave no arquivo de dados e retorna: a quantidade de blocos lidos para encontrá-lo e a quantidade total de blocos do arquivo de índice.

Autor(es): Nathália

Importa os programas: `structs_sec.h`

Observações Importantes: —

Entrada: A entrada é o Título que deseja ser buscado e deve ser passado como argumento ao se executar o programa, **é importante que o Título seja passado entre aspas no terminal**, para não dar problema ao executar o programa.

Saída: O resultado da busca é mostrado no próprio terminal, o programa não gera nenhum arquivo adicional.

Compilação: `g++ seek2.cpp -o seek2`

Execução: `./seek2 <Título>`

[structs_sec.h](#)

Função: Não tem funções, apenas guarda as structs e as constantes mais importantes utilizadas no `upload_sec.cpp` e no `seek2.cpp`.

Autor(es): Nathália

Importa os programas: —

Observações Importantes: Assim como no `bplustree_sec.h`, a constante `MAX_ID`, presente logo no início do programa, deve ser alterada de acordo com o arquivo de entrada utilizado na parte do hashing para gerar o arquivo de dados. **Se foi utilizado o arquivo de entrada padrão `artigo.csv`, então nada precisa ser alterado.** Caso contrário, substitua a constante `MAX_ID` pelo valor do maior (último) ID presente no arquivo de entrada. No caso do `artigo.csv`, o último ID do arquivo é o 1549146.

Entrada: —

Saída: —

Compilação: —

Execução: —

4. Funções de cada programa

4.1. upload_dados.cpp

upload_dados.cpp		
Função	Papel	Autor(es)
<code>funcaoHash()</code>	<p>Função que calcula para que bucket da tabela hash o ID será mapeado.</p> <p>Função hash: $h(k) = k \bmod m$, onde k é o contador* de registros e m é o número de buckets.</p> <p>*Esse contador é a posição do ID de acordo com a quantidade de registros. Foi utilizado esse contador e não o próprio ID pois neste trabalho estamos tratando os casos de overflow, logo, precisávamos de uma função hash que distribuisse bem os registros pela tabela, de forma que não houvesse overflow. Por conta dos casos de ID's faltantes mencionado na Seção 1 deste relatório, optamos por fazer este contador que é incrementado +1 a cada registro.</p>	Nathália
<code>tabela_hash_criar()</code>	Função que cria a tabela hash já definindo os 2 blocos de cada bucket e seus respectivos endereços	Nathália
<code>insere_registro()</code>	Função que insere o registro em um bloco no arquivo de dados	Nathália
<code>libera_tabela()</code>	Função que libera a memória alocada para a tabela hash, ao final do programa	Nathália
<code>ler_arquivo_binario()</code>	Função que faz a leitura do arquivo de dados, salvo em memória secundária em um arquivo binário, e imprime os dados no arquivo de texto <code>output_hash.txt</code>	Nathália
<code>read_field()</code>	Função que faz a leitura de um campo do arquivo de entrada e ignora o ponto e vírgula dentro das aspas do campo, não considerando ele como um separador, e sim como parte do texto	Nathália
<code>read_field_exception()</code>	Função que faz a leitura do campo de Título nos 4 casos de exceção dos registros que ocupam duas linhas, mencionado na Seção 1 deste relatório.	Nathália
<code>main()</code>	Abre o arquivo de entrada, cria o arquivo binário que conterà o arquivo de dados, faz a leitura do arquivo de entrada e chama as respectivas funções citadas anteriormente.	Nathália

4.2. findrec.cpp

findrec.cpp		
Função	Papel	Autor(es)
<code>funcaoHash()</code>	<p>Função que calcula para que bucket da tabela hash o ID será mapeado.</p> <p>Função hash: $h(k) = k \bmod m$, onde k é o contador* de registros e m é o número de buckets.</p> <p>*Esse contador é a posição do ID de acordo com a quantidade de registros. Foi utilizado esse contador e não o próprio ID pois neste trabalho estamos tratando os casos de overflow, logo, precisávamos de uma função hash que distribuisse bem os registros pela tabela, de forma que não houvesse overflow. Por conta dos casos de ID's faltantes mencionado na Seção 1 deste relatório, optamos por fazer este contador que é incrementado +1 a cada registro.</p>	Nathália
<code>tabela_hash_criar()</code>	Função que cria a tabela hash já definindo os 2 blocos de cada bucket e seus respectivos endereços	Nathália
<code>libera_tabela()</code>	Função que libera a memória alocada para a tabela hash, ao final do programa	Nathália
<code>buscaID()</code>	Função que busca um registro no arquivo de dados de acordo com o ID passado como parâmetro	Nathália
<code>read_field()</code>	Função que faz a leitura de um campo do arquivo de entrada e ignora o ponto e vírgula dentro das aspas do campo, não considerando ele como um separador, e sim como parte do texto	Nathália
<code>main()</code>	Abre o arquivo de dados e o arquivo de entrada (utilizado apenas para incrementar o contador utilizado na função hash), lê o arquivo de entrada procurando pelo ID informado e vai incrementando o contador, calcula a posição do registro na tabela hash de acordo com o contador e chama as respectivas funções citadas anteriormente nesta tabela.	Nathália

4.3. upload_pri.cpp

upload_pri.cpp		
Função	Papel	Autor(es)
<code>ler_arquivo_dados()</code>	Função que faz a leitura do arquivo de dados e vai salvando os ID's dos registros e seus respectivos endereços em dois vetores separados	Nathália e Alice
<code>ler_arquivo_primario()</code>	Função que faz a leitura do arquivo de índice, salvo em memória secundária em um arquivo binário, e imprime os dados no arquivo de texto <code>output_pri.txt</code>	Nathália e Igor
<code>main()</code>	Chama as funções citadas anteriormente nesta tabela e a função <code>insert()</code> do programa <code>bplustree_pri.h</code>	Nathália e Alice

4.4. bplustree_pri.h

bplustree_pri.h		
Função	Papel	Autor(es)
<code>insert()</code>	<p>Função que recebe como parâmetro um ID (chave) e seu endereço no arquivo de dados e o insere na árvore B+.</p> <p>Existem 3 tipos de inserção na árvore e para cada um desses 3 tipos existe uma função <code>alocaArvore_tipoX()</code> que faz a alocação do nó no arquivo de índice.</p> <p>Portanto, a função <code>insert()</code> insere a chave na árvore e logo após chama a respectiva função de <code>alocaArvore</code> para alocar o nó no arquivo de índice, lembrando que 1 nó da árvore corresponde a 1 bloco no arquivo.</p>	Alice e Igor
<code>insertInternal()</code>	<p>Função que faz a inserção em um nó interno da árvore B+.</p> <p>Essa função é chamada pela <code>insert()</code> sempre que há um split na árvore ou é necessário fazer o mapeamento em nós internos.</p>	Alice e Igor
<code>findParent()</code>	Função que procura o nó pai de um respectivo nó.	Alice e Igor
<code>contaBlocosInternos()</code>	Função que retorna a quantidade de nós (blocos) internos na árvore B+.	Nathália

<code>getRoot()</code>	Função que retorna o cursor para a raiz da árvore.	Alice
<code>buscaBlocoBinario()</code>	Função que recebe como parâmetro o cursor de um nó da árvore e busca esse mesmo nó no arquivo de índice, retornando o endereço do seu bloco.	Nathália e Igor
<code>alocaArvore_tipo1()</code>	<p>Função que aloca um nó (bloco) no arquivo de índice.</p> <p>O tipo 1 de inserção/alocação só acontece 1 vez, quando a primeira chave é inserida na árvore. Logo, ela é inserida na raiz, que ainda é um nó (bloco) folha.</p> <p>Neste caso, escrevemos a chave e suas informações logo no começo do primeiro bloco (byte 0).</p>	Nathália
<code>alocaArvore_tipo2()</code>	<p>Função que aloca um nó (bloco) no arquivo de índice.</p> <p>O tipo 2 de inserção/alocação acontece sempre que uma chave vai ser inserida em um nó (bloco) que há espaço, ou seja, a árvore não sofre split.</p> <p>Neste caso, procuramos a que bloco o respectivo nó se refere, por meio da função <code>buscaBlocoBinario()</code> e reescrevemos apenas ele, já com a nova chave inserida.</p>	Nathália
<code>alocaArvore_tipo3()</code>	<p>Função que aloca um nó (bloco) no arquivo de índice.</p> <p>O tipo 3 de inserção/alocação acontece quando uma chave vai ser inserida em um nó que já não tem espaço sobrando, logo, é necessário fazer split.</p> <p>Neste caso, como pode haver mudanças em diversos nós da árvore, ficaria difícil acompanhar todas essas mudanças e garantir que todas elas fossem corretamente feitas no arquivo de índice. Portanto, neste caso reescrevemos todos os blocos, desde a raiz.</p>	Nathália

4.5. seek1.cpp

seek1.cpp		
Função	Papel	Autor(es)
<code>contaBlocos()</code>	Função que retorna a quantidade de blocos presente no arquivo de índice	Nathália
<code>buscaArquivoDados()</code>	<p>Função que recebe como parâmetro o ID que está sendo buscado e o endereço (dado pela função <code>buscaChaveIndice()</code>).</p> <p>Este endereço se refere ao endereço do bloco no arquivo de dados onde está o registro pelo ID que está sendo buscado.</p> <p>A partir dessas informações, a função vai até o respectivo endereço no arquivo de dados, lê o bloco e procura, dentre os 2 registros presentes no bloco, aquele que tem o ID que bate com a busca. Em seguida, printa no terminal os campos do registro, a quantidade de blocos lidos até encontrá-lo e a quantidade de blocos totais no arquivo de índice.</p>	Nathália
<code>buscaChaveIndice()</code>	<p>Função que busca, no arquivo de índice, pelo ID (chave) informado.</p> <p>A busca nessa função é igual a uma busca feita na árvore B+, o que muda é que estamos buscando em disco, nos blocos do próprio arquivo de índice.</p> <p>A busca é feita até chegarmos em um bloco folha, a partir daí varremos sequencialmente as chaves dentro do bloco folha até acharmos a que procuramos. Ao achar, chamamos a função <code>buscaArquivoDados()</code> para buscar aquela chave no endereço do arquivo de dados específico.</p>	Nathália
<code>main()</code>	Abre o arquivo de dados e o arquivo de índice e chama a função <code>buscaChaveIndice()</code> para buscar o ID (chave) no arquivo de índice.	Nathália

4.6. upload_sec.cpp

upload_sec.cpp		
Função	Papel	Autor(es)
<code>ler_arquivo_dados()</code>	Função que faz a leitura do arquivo de dados e vai salvando os Títulos dos registros e seus respectivos endereços em dois vetores separados	Nathália e Alice
<code>ler_arquivo_secundario()</code>	Função que faz a leitura do arquivo de índice, salvo em memória secundária em um arquivo binário, e imprime os dados no arquivo de texto <code>output_sec.txt</code>	Nathália e Alice
<code>main()</code>	Chama as funções citadas anteriormente nesta tabela e as funções <code>insert()</code> e <code>alocaArvore()</code> do programa <code>bplustree_pri.h</code>	Nathália e Igor

4.7. bplustree_sec.h

bplustree_sec.h		
Função	Papel	Autor(es)
<code>insert()</code>	Função que recebe como parâmetro um Título (chave) e seu endereço no arquivo de dados e o insere na árvore B+.	Nathália, Alice e Igor
<code>insertInternal()</code>	Função que faz a inserção em um nó interno da árvore B+. Essa função é chamada pela <code>insert()</code> sempre que há um split na árvore ou é necessário fazer o mapeamento em nós internos.	Nathália, Alice e Igor
<code>findParent()</code>	Função que procura o nó pai de um respectivo nó.	Alice e Igor
<code>contaBlocosInternos()</code>	Função que retorna a quantidade de nós (blocos) internos na árvore B+.	Nathália
<code>getRoot()</code>	Função que retorna o cursor para a raiz da árvore.	Alice
<code>buscaBlocoBinario()</code>	Função que recebe como parâmetro o cursor de um nó da árvore e busca esse mesmo nó no arquivo de índice, retornando o endereço do seu bloco.	Nathália e Igor
<code>alocaArvore()</code>	Função que escreve toda a árvore, bloco por bloco, no arquivo de índice secundário.	Nathália

4.8. seek2.cpp

seek2.cpp		
Função	Papel	Autor(es)
<code>contaBlocos()</code>	Função que retorna a quantidade de blocos presente no arquivo de índice	Nathália
<code>buscaArquivoDados()</code>	<p>Função que recebe como parâmetro o Título que está sendo buscado e o endereço (dado pela função <code>buscaChaveIndice()</code>).</p> <p>Este endereço se refere ao endereço do bloco no arquivo de dados onde está o registro pelo Título que está sendo buscado.</p> <p>A partir dessas informações, a função vai até o respectivo endereço no arquivo de dados, lê o bloco e procura, dentre os 2 registros presentes no bloco, aquele que tem o Título que bate com a busca. Em seguida, printa no terminal os campos do registro, a quantidade de blocos lidos até encontrá-lo e a quantidade de blocos totais no arquivo de índice.</p>	Nathália
<code>buscaChaveIndice()</code>	<p>Função que busca, no arquivo de índice, pelo Título (chave) informado.</p> <p>A busca nessa função é igual a uma busca feita na árvore B+, o que muda é que estamos buscando em disco, nos blocos do próprio arquivo de índice.</p> <p>A busca é feita até chegarmos em um bloco folha, a partir daí varremos sequencialmente as chaves dentro do bloco folha até acharmos a que procuramos. Ao achar, chamamos a função <code>buscaArquivoDados()</code> para buscar aquela chave no endereço do arquivo de dados específico.</p>	Nathália
<code>main()</code>	Abre o arquivo de dados e o arquivo de índice e chama a função <code>buscaChaveIndice()</code> para buscar o ID (chave) no arquivo de índice.	Nathália

5. Compilação e Execução

Por fim, nesta seção iremos detalhar o passo a passo para a compilação e execução dos programas e as alterações que devem ser feitas caso queira testar os programas com outro arquivo de entrada senão o `artigo.csv`.

5.1. Dockerfile

Começando pela criação do ambiente para execução dos códigos, vamos explicar como está estruturado nosso Dockerfile, que é mostrado na imagem logo abaixo. Nele, estamos definindo o sistema operacional (*ubuntu 22.04*) com o comando **FROM**, instalando o compilador g++ com o comando **RUN** e copiando para o ambiente os programas necessários para a compilação e execução, com o comando **COPY**. Os programas que estão sendo copiados são todos que possuem as extensões `.cpp` e `.h`, e também o arquivo de entrada `artigo.csv`. Ademais, não estamos instalando nenhuma biblioteca em específico pois todas que foram utilizadas nos códigos são bibliotecas padrão do C++, que não requerem instalações adicionais.

```
1 # Definição do sistema operacional
2 FROM ubuntu:22.04
3
4 # Instalação do compilador C++ (g++), as bibliotecas são padrão do C++
5 RUN apt-get update && apt-get install -y g++
6
7 # Programas necessários para o trabalho
8 COPY *.cpp *.h /src/
9 COPY artigo.csv /src/
```

5.2. Preparação e Criação do Ambiente

Primeiramente, comece baixando os programas presentes no nosso repositório: https://github.com/NathSantos/tp2_Nathalia_Alice_Igor.

Em seguida, baixe o arquivo de entrada por meio do seguinte link: <https://drive.google.com/file/d/1EVoP0d9Wwzj1O6eoFlkel9I3cpe43Gbv/view?usp=sharing>. Basta fazer o download, descompactar o arquivo `artigo.csv.gz` e colocar o `artigo.csv` na mesma pasta junto com os outros programas.

Após isso, já podemos criar nosso ambiente com o Dockerfile. Abra o terminal, navegue até a pasta onde estão os programas e construa a imagem Docker com base no Dockerfile:

```
sudo docker build -t nome_da_imagem .
```

Após a conclusão da construção, execute um contêiner com base na imagem criada:

```
sudo docker run -it nome_da_imagem
```


Isso iniciará o contêiner com a imagem que você construiu e você já estará dentro do contêiner. Agora, como em nosso Dockerfile copiamos os programas para a pasta '/src', navegue até essa pasta, pelo comando `cd src` e liste o conteúdo do diretório com o comando `ls` para se certificar de que está na pasta certa. Todos os programas, assim como o arquivo de entrada, devem aparecer. O resultado será algo parecido com a imagem abaixo.

```
BD TP2 FINAL docker run -it nome_da_imagem
root@0beaba45f52f:/# cd src
root@0beaba45f52f:/src# ls
artigo.csv      bplustree_sec.h  seek1.cpp      structs_dados.h  structs_sec.h    upload_pri.cpp
bplustree_pri.h findrec.cpp      seek2.cpp      structs_pri.h    upload_dados.cpp  upload_sec.cpp
root@0beaba45f52f:/src#
```

Após isso, o ambiente já estará criado e estará tudo pronto para começar a compilação e execução dos programas.

5.3. Geração do Arquivo de Dados

OBS: Importante ressaltar que, caso o arquivo de entrada **não** seja o arquivo padrão `artigo.csv`, deve-se alterar a constante `NUMBER_OF_BUCKETS` no programa `struts_dados.h` como citado anteriormente na seção 3.1.

A primeira etapa, é gerar o arquivo de dados. Para isso, o programa a ser compilado é o `upload_dados.cpp`:

```
g++ upoad_dados.cpp -o up_dados
```

Agora, basta executar o programa, passando o arquivo de entrada como argumento:

```
./up_dados artigo.csv
```

Durante a execução, serão exibidos logs no terminal que indicarão quando o código terminar de ser executado, como mostra a imagem abaixo. Após a execução, teremos nosso arquivo de dados pronto no arquivo binário `arquivo_dados.bin` e um arquivo de texto `output_hash.txt` que lê o arquivo binário e passa para `.txt` apenas para se ter uma visualização de como ficou estruturado o arquivo de dados.

```
Registro 1549145 mapeado para o bucket 255358
Registro 1549146 mapeado para o bucket 255359
***** INSERÇÃO CONCLUÍDA *****
==> Quantidade de Registros Inseridos: 1021439
Liberando tabela hash ...
Tabela hash liberada!
Lendo o arquivo de dados ...
Arquivo binário lido!
Abra o arquivo output hash.txt para verificar a leitura! :D
icomp@grad1-25:~/Downloads/tp2_Nathalia_Alice_Igor-main$
```

5.4. Geração do Arquivo de Índice Primário

OBS: Importante ressaltar que, caso o arquivo de entrada utilizado para gerar o arquivo de dados **não** tenha sido o arquivo padrão `artigo.csv`, deve-se alterar a constante `MAX_ID` nos programas `struts_pri.h` e `bplustree_pri.h` como citado anteriormente na seção 3.2.

A segunda etapa é gerar o arquivo de índice primário. Para isso, o programa a ser compilado é o `upload_pri.cpp`:

```
g++ upoad_pri.cpp -o up_pri
```

Em seguida, vamos executar o programa, sem passar nada como argumento, pois o arquivo de dados já está sendo chamado dentro do programa:

```
./up_pri
```

Assim como na geração do arquivo de dados, serão exibidos logs no terminal para indicar quando ainda está sendo feita a inserção dos ID's na árvore B+ e, conseqüentemente, a geração do arquivo de índice. Após a execução, teremos nosso arquivo de índice pronto no arquivo binário `arquivo_indice_primario.bin` e um arquivo de texto `output_pri.txt` que mostra o arquivo de índice primário.

```
Chave 1039791 inserida com sucesso.
Chave 1549146 inserida com sucesso.
Inserção Concluída!

=====

Lendo arquivo de índice primário ...
Arquivo de índice primário lido!
Abra o arquivo output_pri.txt para verificar a leitura! :D

=====

Número máximo de chaves por bloco: 510
icomp@grad1-25:~/Downloads/tp2_Nathalia_Alice_Igor-main$
```

5.5. Geração do Arquivo de Índice Secundário

OBS: Importante ressaltar que, caso o arquivo de entrada utilizado para gerar o arquivo de dados **não** tenha sido o arquivo padrão `artigo.csv`, deve-se alterar a constante `MAX_ID` nos programas `struts_sec.h` e `bplustree_sec.h` como citado anteriormente na seção 3.3.

A terceira etapa é gerar o arquivo de índice secundário. Para isso, o programa a ser compilado é o `upload_sec.cpp`:

```
g++ upoad_sec.cpp -o up_sec
```

Em seguida, vamos executar o programa, sem passar nada como argumento, pois o arquivo de dados já está sendo chamado dentro do programa:

```
./up_sec
```

Da mesma forma, haverá logs no terminal indicando a situação do programa. Após a execução, teremos nosso arquivo de índice secundário pronto no arquivo binário `arquivo_indice_secundario.bin` e um arquivo de texto `output_sec.txt` que mostra o arquivo de índice secundário.

```
Chave "Stability and instability of individual nodes in multi-hop wireless CSMA/CA network" inserida com sucesso.
Chave "Eficient Symbolic Model Checking for Process Algebras" inserida com sucesso.
Inserção Concluída!
=====
Construindo arquivo de índice secundário ...
Arquivo de índice pronto!
=====
Lendo arquivo de índice secundário ...
Arquivo de índice secundário lido!
Abra o arquivo output_sec.txt para verificar a leitura! :D
=====
Número máximo de chaves por bloco: 12
icomp@grad1-25:~/Downloads/tp2_Nathalia_Alice_Igor-main$
```

5.6. Busca no Arquivo de Dados

OBS: Importante ressaltar que, caso o arquivo de entrada **não** seja o arquivo padrão `artigo.csv`, deve-se alterar a constante `NUMBER_OF_BUCKETS` no programa `struts_dados.h` como citado anteriormente na seção 3.1.

Por fim, podemos partir para realizar as buscas nos arquivos gerados. Começando pelo arquivo de dados, compile o programa `findrec.cpp`:

```
g++ findrec.cpp -o findrec
```

E, em seguida, a busca já pode ser feita diretamente no arquivo de dados por um registro com o ID passado como argumento:

```
./findrec <ID>
```

Se existir, os campos do registro serão retornados, assim como a quantidade de blocos lidas para encontrá-lo e a quantidade total de blocos no arquivo de dados:

```

icomp@grad1-25:~/Downloads/tp2_Nathalia_Alice_Igor-main$ g++ findrec.cpp -o findrec
icomp@grad1-25:~/Downloads/tp2_Nathalia_Alice_Igor-main$ ./findrec 922663
>>> Programa que busca por um registro diretamente no arquivo de dados <<<

Buscando pelo registro de ID 922663 ...

Registro encontrado no bucket!
Procurando no bloco 1 do bucket ...
Procurando no bloco 2 do bucket ...

Registro encontrado no segundo bloco do bucket!
Endereço do bloco: 1481781248

-----
ID: 922663
TITULO: Mining, ranking and recommending entity aspects
ANO: 2015
AUTORES: R Reinanda, E Meij, M de Rijke
CITACOES: 2
ATUALIZACAO: 2016-06-23 13:39:42
SNIPPET: Mining, ranking and recommending entity aspects. R Reinanda, E Meij, M de R
ijke - Development in Information Retrieval, 2015 - dl.acm.org. Abstract Entity qu
eries constitute a large fraction of web search queries and most of these queries a
re in the form of an entity mention plus some context terms that represent an intent
in the context of that entity. We refer to these entity-oriented search intents as
entity .."
-----
Quantidade de blocos lidos: 2
Quantidade de blocos totais no arquivo: 510720
-----
icomp@grad1-25:~/Downloads/tp2_Nathalia_Alice_Igor-main$

```

5.7. Busca no Arquivo de Índice Primário

OBS: Importante ressaltar que, caso o arquivo de entrada utilizado para gerar o arquivo de dados **não** tenha sido o arquivo padrão `artigo.csv`, deve-se alterar a constante `MAX_ID` no programa `struts_pri.h` como citado anteriormente na seção 3.2.

Para fazer a busca no índice primário, é semelhante à busca no arquivo de dados, vamos compilar o `seek1.cpp`:

```
g++ seek1.cpp -o seek1
```

E, em seguida, a busca já pode ser feita através do arquivo de índice primário por um registro com o ID passado como argumento:

```
./seek1 <ID>
```

Se existir, os campos do registro serão retornados, assim como a quantidade de blocos lidos para encontrá-lo e a quantidade total de blocos no arquivo de índice:

```

icomp@grad1-25:~/Downloads/tp2_Nathalia_Alice_Igor-main$ g++ seek1.cpp -o seek1
icomp@grad1-25:~/Downloads/tp2_Nathalia_Alice_Igor-main$ ./seek1 922663
>>> Programa que busca por um registro pesquisando através do arquivo de índice prim
ário <<<

Buscando pelo registro de ID 922663 ...
Registro encontrado!

Endereço do bloco no arquivo de dados: 1481781248
----- Campos do Registro -----
ID: 922663
TITULO: Mining, ranking and recommending entity aspects
ANO: 2015
AUTORES: R Reinanda, E Meij, M de Rijke
CITACOES: 2
ATUALIZACAO: 2016-06-23 13:39:42
SNIPPET: Mining, ranking and recommending entity aspects. R Reinanda, E Meij, M de R
ijke - Development in Information Retrieval, 2015 - dl.acm.org. Abstract Entity qu
eries constitute a large fraction of web search queries and most of these queries a
re in the form of an entity mention plus some context terms that represent an intent
in the context of that entity. We refer to these entity-oriented search intents as
entity .."
----- Blocos -----
Quantidade de blocos lidos (somente no arquivo de índice): 3
Quantidade de blocos lidos (arquivo de índice +1 bloco do arquivo de dados): 4
Quantidade de blocos totais no arquivo: 2684
-----
icomp@grad1-25:~/Downloads/tp2_Nathalia_Alice_Igor-main$

```

5.8. Busca no Arquivo de Índice Secundário

OBS: Importante ressaltar que, caso o arquivo de entrada utilizado para gerar o arquivo de dados **não** tenha sido o arquivo padrão `artigo.csv`, deve-se alterar a constante `MAX_ID` no programa `struts_sec.h` como citado anteriormente na seção 3.3.

Para fazer a busca no índice secundário, vamos compilar o `seek2.cpp`:

```
g++ seek2.cpp -o seek2
```

E, em seguida, a busca já pode ser feita através do arquivo de índice secundário por um registro com o Título passado como argumento. Importante lembrar que o título deve ser passado entre aspas, caso contrário irá dar erro no terminal por conta dos possíveis espaços em branco.

```
./seek2 <Título>
```

Se existir, os campos do registro serão retornados, assim como a quantidade de blocos lidos para encontrá-lo e a quantidade total de blocos no arquivo de índice:

```
icomp@grad1-25:~/Downloads/tp2_Nathalia_Alice_Igor-main$ g++ seek2.cpp -o seek2
icomp@grad1-25:~/Downloads/tp2_Nathalia_Alice_Igor-main$ ./seek2 "Mining, ranking and recommending entity aspects"
>>> Programa que busca por um registro pesquisando através do arquivo de índice secundário <<<

Buscando pelo registro de Título: Mining, ranking and recommending entity aspects ..
Registro encontrado!

Endereço do bloco no arquivo de dados: 1481781248
----- Campos do Registro -----
ID: 922663
TITULO: Mining, ranking and recommending entity aspects
ANO: 2015
AUTORES: R Reinanda, E Meij, M de Rijke
CITACOES: 2
ATUALIZACAO: 2016-06-23 13:39:42
SNIPPET: Mining, ranking and recommending entity aspects. R Reinanda, E Meij, M de Rijke - Development in Information Retrieval, 2015 - dl.acm.org. Abstract Entity queries constitute a large fraction of web search queries and most of these queries are in the form of an entity mention plus some context terms that represent an intent in the context of that entity. We refer to these entity-oriented search intents as entity ..
----- Blocos -----
Quantidade de blocos lidos (somente no arquivo de índice): 7
Quantidade de blocos lidos (arquivo de índice +1 bloco do arquivo de dados): 8
Quantidade de blocos totais no arquivo: 144980
-----
icomp@grad1-25:~/Downloads/tp2_Nathalia_Alice_Igor-main$
```