Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
Departamento de Ciências de Computação

Trabalho 2

Disciplina: SSC0215 - TURMA B - ORGANIZAÇÃO DE ARQUIVOS
Profª. Dra. CRISTINA D. A. CIFERRI

Alunos:

Gabriel H. C. Scalici	9292970
Mateus Virgínio Silva	10284156
Eduardo A. Baratela	10295270
Rodrigo Noventa Júnior	9791243

Índice:

Seções:

Decisões de Projeto

O grupo optou por utilizar a mesma estrutura do trabalho 1 na construção dessa segunda parte do projeto, de forma que todas as funções do anterior estejam presentes em conjunto com as novas funcionalidades exigidas para o novo projeto.
Uma das decisões que tivemos, foi utilizar a numerologia da descrição do trabalho 2, isto é, a primeira função não sendo a 1 mas sim a 10.

Foi utilizado como base para a implementação dos algoritmos referentes à árvore B, o livro Algoritmos (Thomas H. Cormen), de forma que as funções foram adaptadas para uso de *buffer pool* exigido na descrição do trabalho.

Para todas as funcionalidades criadas no projeto, foram utilizadas mensagens padrão para quando a operação é realizada com sucesso ou quando houve qualquer tipo de falha.

Vale lembrar que, para que seja facilitada a correção, toda a parte pertinente à segunda parte foi desenvolvida no final dos documentos *registro.c, registro.h.*
Tanto as variáveis criadas, quanto novas funções implementadas.

Estrutura

Cabecalho:

1 byte	4 bytes	4 bytes
status	noRaiz	altura

```c
//Definindo o cabecalho do arquivo de arvore B
typedef struct{
     int noRaiz;
     int altura;
     int ultimoRRN;
     char status;
} Cabecalho_B;
```

Árvore:

4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	...	4 bytes	4 bytes	4 bytes
n	P_1	C_1	P_{R1}	P_2	C_2	P_{R2}	...	C_9	P_{R9}	P_{10}
0	1	...							114	115

```c
//Definindo os nos da arvore B
typedef struct{
    int n;
    int p[10];
    int c[9];
    int pr[9];
} arvoreB;
```

Estrutura da BufferPool:

```c
//Definindo a estrutura da bufferpool
typedef struct{
    arvoreB node[TAM_BUFFER];
    int RRN[TAM_BUFFER];          //Aux
    int freq[TAM_BUFFER];
} bPool;
```

Instruções para escolha da função correta:

```
enum options{
    LEITURA = 1,
    RECUPERA = 2,
    BUSCAPARAM = 3,
    BUSCARRN = 4,
    REMOCAO = 5,
    INSERE = 6,
    ATUALIZA = 7,
    COMPACTA = 8,
    RECUPERAREMOVIDOS = 9,
    INSERIR_INDICE = 10,
    INSERIR_REGISTRO_INDICE = 11,
    RECUPERAR_INDICE = 12,
};
```

Seção 1: Buffer Pool

As funções que fazer parte da lógica para a construção estão descritas abaixo de maneira simplificada. Aqui temos as funções com seus respectivos parâmetros:

```
//Buffer pool
int get(FILE *b, int RRN, bPool *bufPool);                  //Recupera o conteúdo de um nó
int put(FILE *b, int RRN, arvoreB* page, bPool *bufPool);   //Armazena página no buffer
void flush_full(bPool *bufPool);                            //Manda todas as páginas pro arquivo
void flush(FILE *b, arvoreB page, int RRN, bPool *bufPool); //Manda uma página específica pro arquivo
void swapRaiz(int RRN_novaRaiz, bPool* bp);                 //Coloca a nova raiz na posição certa do buffer
void printa_bPool(bPool* bp);                               //Printa o bufferpool inteiro
```

Foi implementado, assim como exigido na descrição do projeto, a capacidade máxima de 5 nós armazenados na buffer pool, porém sempre mantendo o nó raiz.

Algumas funções possuem nomes parecidos porém exercem tarefas um pouco diferentes, como por exemplo o flush_full (que após salvar todas as páginas do buffer_pool, esvazia ele por completo) e o flush que salva apenas uma página, retirando-a do buffer_pool.

Duas funções principais *put()* e *get()*, será esquematizado abaixo o funcionamento básico de ambas (Com o mesmo conteúdo que foi passado pela professora em sala de aula).

- put() :

  IF não está no buffer POOL

  1: copie o conteúdo da página para uma variável auxiliar do tipo IndexPage, chamada P (Não vi necessidade)

  2: se o buffer conter espaço disponível : insira P no espaço disponível

  3: caso contrário : remova uma página N do buffer e insira P (LFU)

  ELSE se está no buffer

  1: recupere a página armazenada no buffer e atualize seu conteúdo

  2: marque que P é uma página modificada

3: reorganize a estrutura interna do buffer (LFU)

- get():
  IF está no bufferpool
  1: copie o conteúdo da página requerida para uma variável auxiliar do tipo árvore B, chamada P
  2: realize possíveis reestruturações na organização do buffer (LFU)
  3: retorne P
  ELSE não está no buffer
  1: recupere o conteúdo da página do disco e armazena em uma variável auxiliar do tipo IndexPage, chamada P
  2: insira P no buffer, chamando a função put
  3: retorne P.

Seção 2: Função 10 - Inserção árvore B

Função deve ser acoplada à funcionalidade 1 (Criação do arquivo de índice com base no arquivo binário de dados), de forma que cada vez que o usuário inserir um novo registro no arquivo de dados, este também será inserido no arquivo de índice.

```
//ArvoreB
FILE* criar_indice(Registro *reg, int qtdRegs);
void criar_arvore_B(Registro *reg, int qtdRegs);
void inserir_B(FILE *b, Registro reg, int RRN_reg, bPool *bp);
void insere_naoCheio_B(FILE *b, arvoreB* x, int RRN_indiceX, Registro reg, int RRN_reg, bPool *bp);
void split_B(FILE *b, bPool *bp, arvoreB* pai, int RRN_pai, int pont, arvoreB* filhoCheio, int RRN_filhoCheio);
void busca_B(Registro *reg, int chave, int rrn);
```

Temos aqui também algumas funções para a própria criação da árvore binária, como o cria índice e cria árvore, que são chamados em conjunto com a criação do arquivo de registro (com cabeçalho).

A função que divide um determinado nó em dois outros nós atualizando os valores mais ou menos pela metade é chamado de split, assim como aprendido na teoria, e sua lógica para implementação foi feita com base na tentativa de inserir um novo valor e não ter espaço suficiente para ele, com uma checagem extra, se caso o nó que estamos dividindo não seja um nó folha, que está representado pela imagem a seguir.

```
//checa se filhoCheio nao é no folha
if(filhoCheio->p[0] != -1){
  for(i = 0; i < 5; i++)
    new->p[i] = filhoCheio->p[i+5];
}
filhoCheio->n = 4;
new->n = 4;
```

Onde a divisão, atualização de ponteiros e valores de uma página, são feitos seguindo a lógica abaixo, na qual além de pensarmos na nova organização, estamos tratando a nova divisão dos valores, sempre mantendo ordenado para que não perca a característica principal da árvore B que são seus elementos ordenados para facilitar a busca por uma determinada chave.

```
//ordena os ponteiros do no pai até o espaco do novo ponteiro
for(i = pai->n; i >= pont+1; i--)
  pai->p[i+1] = pai->p[i];

//ponteiro de pai recebe o rrn do novo no (do lado do no do irmao)
pai->p[pont+1] = ultimoRRN;

//ordena as chaves de pai até a posicao desejada
for(i = pai->n-1; i >= pont; i--){
  pai->c[i+1] = pai->c[i];
  pai->pr[i+1] = pai->pr[i];
}
```

Entretanto, como discutido anteriormente, não basta somente atualizar os registros, temos que manter a buffer_pool atualizada e consistente de forma que as funções não deixem de funcionar quando ocorre um split, que ocorre nas inserções na buffer pool explicado abaixo:

```
//insere os nos alterados no buffer
put(b, RRN_filhoCheio, filhoCheio, bp);
put(b, RRN_pai, pai, bp);
put(b, ultimoRRN, new, bp);
```

A função de inserir caso o nó não esteja vazio, é simples e não necessita de exemplos da implementação. Resumidamente, caso o nó esteja vazio e deseja-se inserir nele, é feito somente mais uma checagem para ver se é um nó folha ou não (pois um novo nó só pode ser inserido na árvore B caso seja um nó folha), caso seja então acha-se a posição correta e o insere, caso não seja folha, com base nos ponteiros, desce até o filho correto tentando novamente fazer a inserção, até que seja um nó folha e então possa ser inserido corretamente.

Seção 3: Função 11 - Inserção árvore B

Função deve ser acoplada à funcionalidade 6 (Inserção de um novo registro com os campos editados pelo usuário).

Tal função possui basicamente o mesmo objetivo que a anterior, porém antes era realizado a leitura dos registros do arquivo para a criação da árvore B, agora contamos com os dados desejados pelo usuário à ser inserido nos registro e consequentemente na árvore B.

Seção 4: Função 12 - Recuperação de Registro

A busca do registro deve ser feita com o arquivo de índice da árvore B, de forma a seguir a técnica ensinada em sala de aula.

O passo a passo do algoritmo trata-se basicamente de ler o valor a ser buscado, pegar o valor da raiz que já está armazenado no cabeçalho, e verificando se é maior ou menor que os valores do nó, ir descendo até no máximo a altura da árvore B.

Tendo como base o algoritmo abaixo, discutido no livro do Cormen:

```
B-TREE-SEARCH(x, k)
1 i ← 1
2 while i ≤ n[x] e k > chave_i[x]
3    do i ← i + 1
4 if i ≤ n[x] e k = chave_i[x]
5    then return (x, i)
6 if folha[x]
7    then return NIL
8    else DISK-READ(c_i[x])
9         return B-TREE-SEARCH(c_i[x], k)
```

Há uma grande diferença para nosso trabalho pois estamos trabalhando com a implementação de uma buffer-pool, o arquivo de índice não está sendo acessado diretamente, de forma que quem faz essa "interface" entre o usuário e o arquivo de índice, é o próprio buffer_pool.

Dessa forma usamos a função *get()* para retornar a posição determinada no buffer pool e comparando os valores dessa página com o valor codINEP desejado. Caso seja igual, é retornado o valor do rrn para a main, para que seja chamada a função de busca pelo rrn e exibido para o usuário o registro desejado.

Caso não seja encontrado, optamos por utilizar a busca de maneira recursiva, porém agora passando o rrn do filho (Após análise se o valor é menor ou maior que os valores da página do buffer pool).

Dessa forma estamos usando uma função já implementada no trabalho anterior (busca_rrn) para auxiliar na nova busca, pois estamos apenas pegando o rrn desejado a partir da busca pela árvore B.

```
case(RECUPERAR_INDICE):{
    int rrn_desejado = 0;
    //PAssando o rrn 0 da raiz para que seja encontrado o rrn do codInep desejado
    rrn_desejado = busca_B(Registro *reg, argv[2], 0);
    //Chamando a funcao de busca pelo registro de rrn desejado
    //Já exibe corretamente o valor da busca
    busca_rrn(rrn);
    break;
}
```

Portanto foi decidido que seria utilizado a função recursiva de busca na árvore binária pela simplicidade e facilidade de entendimento e implementação.

Seção 5: Função 13 - Remoção de Registro

Seção 6: Função 14 - Atualização de Registro

Conclusão

O grupo não achou o trabalho em si muito difícil, até pela facilidade do entendimento de como funciona uma árvore B na prática e de existir muito material de apoio na internet e livros de consulta (Como o utilizado para esse trabalho).

Porém encontramos muita dificuldade em conseguir juntar todo o conhecimento de árvore B com o buffer pool, de forma que até mesmo os pseudo-códigos do livro do Cormen precisassem ser adaptados. Dessa forma coloca o buffer_pool e implementar suas funções foi a parte mais complicado do trabalho, que demandou mais tempo e pesquisa por parte dos integrantes.