

Trabalho 2 – Árvore-B

| | |
|--|------------------------|
| Nome: Alexandre Batistella Bellas | Nº USP: 9763168 |
| Caique Honorio Cardoso | 8910222 |
| Felipe Manfio Barbosa | 9771640 |
| João Vitor Granzotti Machado | 9393322 |

1. Descrição do Problema

O projeto consiste em implementar um TAD para Árvore-B de ordem 5, com a função de indexação de dados, e as operações de criação, inserção e busca. A árvore-B deve ser mantida em memória secundária (em disco). Considere que os registros de dados (arquivo de dados) são formados pelos seguintes campos:

1. ID numérico da música;
2. Título da música;
3. Gênero.

Para o arquivo de dados, implemente:

- registros de tamanho variável: no início de cada registro deve haver um byte indicando o tamanho total do registro; na sequência, as informações referentes aos campos Id, Título e Gênero devem ser separadas pelo caractere “|”. O registro é definido da seguinte forma:

```
typedef struct{
    int id;
    char titulo[30];
    char gênero[20];
}tRegistro;
```

Figura 1: estrutura de um registro definido na descrição do trabalho.

- inserção no arquivo de dados: somente no final do arquivo (para simplificar);
- remoção no arquivo de dados: somente com “marcador” de registro removido (para simplificar).

OBS: a busca para remoção no arquivo de dados deve ser feita a partir da estrutura de indexação.

O sistema deve armazenar os dados e a estrutura de indexação (árvore-B) de modo a realizar consultas eficientes. As funcionalidades requeridas são:

Funcionalidade 1: Criação do (arquivo de) índice a partir do arquivo de dados já pronto (considerando a ordem dos registros no arquivo de dados).

Funcionalidade 2: Inserção de novas músicas no arquivo de dados e no índice.

Funcionalidade 3: Pesquisa (busca) por Id da música.

Funcionalidade 4: Remoção de música a partir do Id (funcionalidade bônus – valendo 2 pontos extras).

Funcionalidade 5: Mostrar Árvore-B (funcionalidade bônus - valendo 1 ponto extra se as funcionalidades de 1 a 3 estiverem corretas).

2. Funcionamento

1) Organização do arquivo de dados:

O arquivo de dados segue o padrão proposto, onde existem basicamente três campos que identificam uma música: seu ID, título e gênero, os quais são de tamanho variável. Entretanto, existe um tamanho limite (o qual é dado pelo número de elementos dos vetores que o representam). O modelo da figura 1 ilustra a definição da estrutura de registro em si.

2) Organização do arquivo de índice:

Para o arquivo de índices foi proposto um padrão onde os registros têm tamanho fixo. Cada registro representa uma página na árvore-B, ou seja, um registro é composto pelo seu número de elementos e seu vetor de chaves, o qual contém os IDs e seus respectivos filhos, seguindo a estrutura de uma página (nó) para árvore-B de ordem N, como ilustrado na figura abaixo.

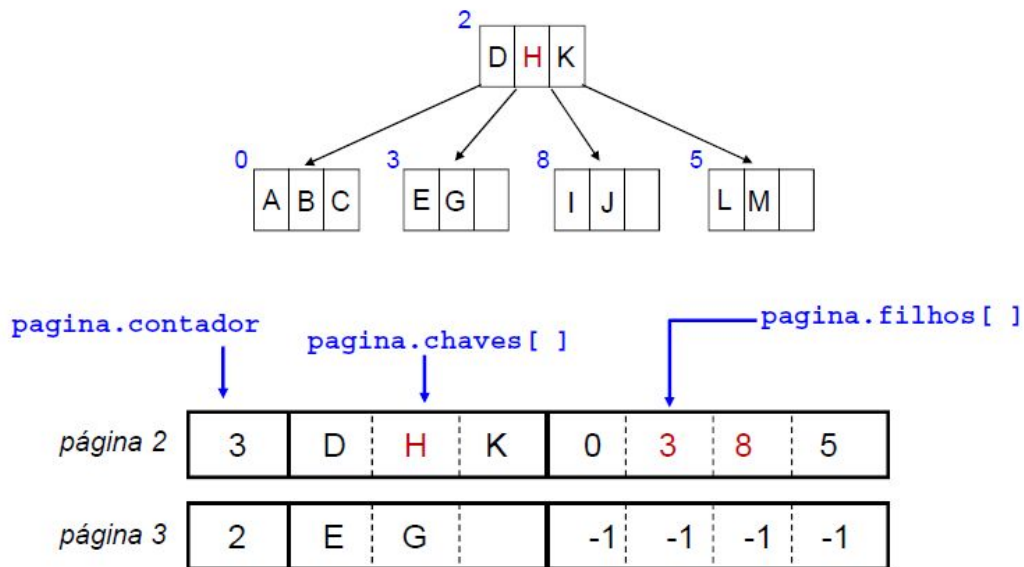


Figura 2: exemplo de página - nó de árvore-B com ordem 4.

Dentro do programa a seguinte estrutura é definida para criar um tipo abstrato de dados que satisfaça a página que queremos criar, como exemplificado acima:

```
typedef struct Pagina{
    //numero de chaves na pagina
    int numeroChaves;
    //vetor de chaves
    int chaves[ORDEM-1][2];
    //vetor de ponteiros das paginas filhas
    int filhos[ORDEM];
    //define se é folha, folha = 1, não folha = 0
    int folha;
} PAGINA;
```

Figura 3: Codificação da estrutura de dados esquematizada na figura 3.

3) Organização do arquivo de log:

Para o arquivo de log as instruções fornecidas exigiam que frases, referentes aos passos executados, fossem gravadas em um arquivo de nome “log_X.txt”, em que X é 1ª letra do nome concatenada ao último sobrenome do integrante do grupo responsável pela submissão do trabalho no TIDIA. De acordo com a situação as seguintes mensagens serão gravadas:

Caso operação 1 - criação de índice, gravar no arquivo de log:

“Execução da criação do arquivo de índice <NOME1> com base no arquivo de dados

<NOME2>.” <NOME1> e <NOME2> indicam, respectivamente, o nome dos arquivos de índice e de dados.

Caso operação 2 - operação de inserção, gravar no arquivo de log:

“Execucao de operacao de *INSERCAO* de <Id>, <titulo>, <genero>.” que indicam os dados da música a ser inserida.

Na sequência gravar no arquivo de log de acordo com o que ocorreu durante a operação de inserção:

- “Divisao de no - pagina X” deve ser impressa sempre que um nó for dividido - X é RRN da página dividida;

- “Chave <Id> promovida” deve ser impressa sempre que uma chave for promovida.

- “Chave <Id> inserida com sucesso” deve ser impressa ao final da inserção indicando sucesso da operação.

- “Chave <Id> duplicada” deve ser impressa ao final da inserção e indica que a operação de inserção não foi realizada.

Caso operação 3 - busca por ID, gravar no arquivo de log:

“Execucao de operacao de *PESQUISA* de <Id>”. <Id> indica o número a ser pesquisado. Na sequência gravar no arquivo de log:

- “Chave <Id> encontrada, offset <Y>,”

Titulo: <titulo>, Genero: <genero>”, indica que a chave <Id> foi encontrada e possui offset associado <Y> e os dados do registro são <titulo> e <gênero>.

- “Chave <Id> nao encontrada” indica que a chave <ID> não está presente na árvore-B e nem no arquivo de dados.

Um header separado foi criado para administrar todas as funções que dizem respeito ao log, seu nome é “TAD_log.h”, onde funções para criação e atualização do arquivo são encontradas. Seu funcionamento é simples. A mensagem que se deseja escrever vem como parâmetro da função atualizar para ser gravada, e segue o pretexto descrito anteriormente.

```

//Abre o arquivo de log, ou o cria se ele não existir
void criarArquivoDeLog() {
    FILE *arquivo;
    arquivo = fopen(ARQLOG, "w");
    if (arquivo == NULL) {
        printf ("Houve um erro ao criar o arquivo de log %s. Erro 0x2000.\n", ARQLOG);
        return;
    }
    fprintf(arquivo, "Historico das operacoes realizadas no programa:\n");
    fclose (arquivo);
    printf("Arquivo de log criado com sucesso.\n");
}

// coloca as ações no arquivo de log
void atualizaArquivoDeLog(char linhaAdicionada[], int att){
    FILE *arquivo;
    arquivo = fopen (ARQLOG, "a");
    if (arquivo == NULL) {
        printf ("Houve um erro ao abrir o arquivo de log %s. Erro 0x2001.\n", ARQLOG);
        return;
    }
    fprintf(arquivo, linhaAdicionada);
    if(att == LOG_MSG_ON)
        printf ("Arquivo de 'log' atualizado com sucesso.\n");

    fclose (arquivo);
}

```

Figura 4: lógica responsável pela manutenção do arquivo de Log.

4) Funcionamento da inserção de novas músicas:

A operação de inserção, que corresponde à funcionalidade 2 do trabalho, pode ser dividida em três etapas. Na primeira etapa a informação (ID, título e gênero) é obtida como entrada do usuário, e sua validade é verificada através de funções específicas. Na segunda etapa, a função “inserirMusica” é chamada. Nela, a música é inserida no arquivo de dados e no de índices, tomando os cuidados necessários:

- Se a árvore estiver vazia, um nó raiz é criado e a chave é inserida nele;
- Se a árvore não estiver vazia, recuperamos as informações da raiz e damos início ao processo de inserção, onde primeiro fazemos uma busca na árvore para verificar se o ID em questão já existe; se este não existir, a inserção é realizada seguindo a teoria de árvore-B; se existir o programa retorna uma FLAG de erro;
- Na terceira e última etapa todas as mensagens de interesse são enviadas para a função de atualização do arquivo de Log. A imagem abaixo ilustra a declaração da função com seus parâmetros e suas variáveis locais, usadas na execução.

```

//Insere as musicas no arquivo dados.dad
void inserirMusica(tRegistro novoRegistro){
    char size;
    int i, j, pos;
    char buffer[1000];
    char *mensagem = malloc(100*sizeof(char));
    int RRNraiz;
    int offsetNovaChave;
    int flagBusca;
    long tamArq;

```

Figura 5: detalhe da função de inserção. Podemos ver os parâmetros recebidos e suas variáveis locais.

Podemos verificar as etapas descritas acima na imagem abaixo, onde as funções foram compactadas, a fim de mostrar dar ênfase à estrutura e lógica gerais.

```

//Se estiver vazio, ou seja, só tiver o bit de atualizado,
if(tamArq == sizeof(int)){
//Se não estiver vazio:
else{
//Fechamos o arquivo pois não precisamos mais dele nessa função
fclose(arqind);

//Buscamos o id passado a partir da raiz
flagBusca = busca(novoRegistro.id, RRNraiz);

//Se deu erro, informamos que deu erro
if(flagBusca == ERRO){
//Se não encontramos, inserimos normalmente
else if(flagBusca == NAOENCONTRADO){
//Se encontrou
else{

fclose (arqdados);
free(mensagem);
}
}
}
}

```

Figura 6: exemplo em código dos passos descritos anteriormente para a inserção.

5) Funcionamento da busca:

A operação de pesquisa, que corresponde à funcionalidade 3 do trabalho, pode ser dividida em três etapas. Na primeira etapa é pedido ao usuário um valor de ID a ser procurado na árvore.

Na segunda etapa o ID é procurado na árvore e verifica-se se o ID se encontra na página atual; se não, vai para a página correta - acessada por meio dos filhos do nó - de acordo com a comparação do ID pesquisado com os IDs existentes naquele nó, e assim por diante, até encontrar o valor em questão. Se o valor não for encontrado, ou seja, a pesquisa chegou a uma página folha e não obteve o valor desejado, então uma FLAG de erro é ativa.

Na terceira e última etapa, o arquivo de log é atualizado de acordo com as funções que ocorreram durante a execução. A imagem abaixo mostra o código da função de maneira compactada, com o intuito de evidenciar a lógica utilizada.

```
//É feita a busca pelo offSet do id.
offSet = busca(idBusca, RRNraiz);

//Se deu erro,
if(offSet == ERRO){
//Se não encontrou,
else if(offSet == NAOENCONTRADO){

//Abre o arquivo de dados para obter o registro pesquisado.
arq = fopen(ARQDADOS, "r");
if(arq == NULL){

//Pega o registro pesquisado.
fseek(arq, offSet, SEEK_SET);

fread(&size, sizeof(size), 1, arq);
fread(buffer, size, 1, arq);
pos = 1;
sscanf(parser(buffer, &pos), "%d", &reg.id);
strcpy(reg.titulo, parser(buffer, &pos));
strcpy(reg.genero, parser(buffer, &pos));

printf("Registro encontrado:\nid = %d\nTitulo = %s\nGenero = %s\n\n", reg.id, reg.titulo, reg.genero);
```

Figura 7: passos da execução da função de busca.

A função busca() é ilustrada abaixo:

```
int busca(int id, int RRNAtual){
//Contador
int i;
//Abrir arquivo de índice para busca
FILE *arq = fopen(ARQIND, "rb");
//Se não abriu o arquivo,
if(arq == NULL){
return ERRO;
}

//Struct para pegar a página do arquivo de índices
PAGINA buscaPag;

//Percorrimeto no arquivo de índice para pegar a página em disco, e logo fecha o arquivo
fseek(arq, 3*sizeof(int) + RRNAtual*sizeof(buscaPag), SEEK_SET);
fread(&buscaPag, sizeof(buscaPag), 1, arq);
fclose(arq);
```



```

//Busca pela chave na página atual
for(i = 0; i < buscaPag.numeroChaves && id >= buscaPag.chaves[i][0]; i++){
    //Se o id estiver na página,
    if(buscaPag.chaves[i][0] == id){
        //Retorna o offset do id
        return buscaPag.chaves[i][1];
    }
    //senão, continua...
}
//Aqui, i é a posição que id "deveria estar". Ir para o filho i, se existir.

if(buscaPag.filhos[i] != -1){
    return busca(id, buscaPag.filhos[i]);
}
//Se não existir filho, então a chave não existe na árvore.
else{
    return NAOENCONTRADO;
}
}

```

Figura 8: código referente à função de busca.

6) Funcionamento da criação do arquivo de índices:

A criação de um arquivo de índices, a partir de um arquivo de dados existente, diz respeito à funcionalidade 1 do trabalho. Podemos dividir essa operação em 3 etapas.

Começamos em um loop que lê o arquivo de dados, registro a registro, para que os mesmos possam ser inseridos um por um na árvore. Além disso, o tamanho do arquivo de dados é obtido para saber a posição de inserção na árvore. Existe a possibilidade de o arquivo de índice estar vazio; se acontecer, é necessário criar uma raiz e inserir a chave em questão. Se já existir uma raiz, ela é obtida e a função inserir(), já descrita aqui, é chamada. Na última etapa, o arquivo de Log é atualizado de acordo com as funções realizadas e seguimos no loop até chegarmos ao fim no arquivo de dados. A figura abaixo mostra a função de criação de índice compactada.

```

void criarArquivoDeIndice(int modoLog){
    //Abrir os arquivos de dados e índice, o primeiro como leitura e o segundo como escrita
    FILE *arqdados, *arqind;
    arqdados = fopen (ARQDADOS, "rb");
    arqind = fopen (ARQIND, "wb");

    //Se não deu para abrir um deles
    if (arqdados == NULL) {
        if(arqind == NULL){

            //Variáveis a serem utilizadas
            int i, j; //Contadores
            int atualizado = 1; //Variável para dizer que o arquivo de índice está atualizado
            int RRNraiz; //Variável para guardar o RRN da página raiz
            int offset = 0; //Variável para guardar o offset do próximo registro a ser inserido
            int pos; //Variável auxiliar para ajudar na leitura do registro
            long tamanhoArquivo; //Variável para obter o tamanho de um arquivo
            char size; //Variável para pegar o tamanho de um registro
            char buffer[1000]; //Vetor para pegar o conteúdo de um registro
            tRegistro reg; //Registro auxiliar

```



```

//É escrito que o arquivo está atualizado
fwrite(&atualizado, sizeof(int), 1, arqind);
fclose(arqind);

//Lendo o arquivo de dados, registro a registro
while (fread(&size, sizeof(size), 1, arqdados)) {

//Operação de escrever no log
//Somente escreve no log se for feita a operação 1.
if(moduloLog == FUNC 1){

//Fecha arquivos de dados, pois o de índice já foi fechado antes
fclose (arqdados);
}

```

Figura 9: código da função de criação de índices, com alguns trechos omitidos(forma compactada).

Podemos verificar em detalhes a operação realizada dentro do “while” que pega registro a registro do arquivo de dados e insere na árvore.

```

//Lendo o arquivo de dados, registro a registro
while (fread(&size, sizeof(size), 1, arqdados)) {
    fread(buffer, size, 1, arqdados);
    pos = 1;
    sscanf(parser(buffer, &pos), "%d", &reg.id);
    strcpy(reg.titulo, parser(buffer, &pos));
    strcpy(reg.genero, parser(buffer, &pos));

    //Pega o tamanho do arquivo de índice
    arqind = fopen(ARQIND, "r+b");
    fseek(arqind, 0, SEEK_END);
    tamanhoArquivo = ftell(arqind);
    fclose(arqind);

    //Se só tiver o bit de estar atualizado ou não
    if(tamanhoArquivo == sizeof(int)){
        //Se não, já possui uma ou mais páginas no arquivo de índice
        else{

            //offset pula o tamanho do registro para ir para o próximo
            offset += sizeof(size)+size;

            //Como offset, agora, é o offset do próximo elemento a ser inserido, registramos ele
            //no cabeçalho do arquivo de índices
            atualizaOffsetCabeçalho(offset);
        }
    }
}

```

Figura 10: estrutura responsável pela recuperação do conteúdo do arquivo de dados.

7) Funcionamento da Função responsável pela exibição da árvore B:

Para realização desta funcionalidade foi necessário implementar um TAD de filas, já que a árvore será impressa utilizando uma busca em largura. Um header “TAD_fila.h” foi adicionado, contendo todas as operações em fila que serão executadas no processo. Podemos dividir esse processo em etapas. Na primeira etapa uma função responsável por

atualizar o arquivo de log e pegar a raiz da árvore é chamada, solicitando uma busca em largura na árvore e enviando a raiz como parâmetro.

```
void mostrarArvore(){
    FILE *arq = fopen(ARQIND, "rb");

    if(arq == NULL){
        printf("Nao foi possivel abrir o arquivo de indice %s. Erro 0x1009.\n", ARQIND);
        return;
    }

    char *mensagem;
    mensagem = malloc(50*sizeof(char));

    sprintf(mensagem, "Execucao de operacao para mostrar a arvore-B gerada:\n");
    atualizaArquivoDeLog(mensagem, LOG_MSG_ON);

    int RRNraiz;
    fseek(arq, sizeof(int), SEEK_SET);
    fread(&RRNraiz, sizeof(RRNraiz), 1, arq);

    bfs(RRNraiz);
}
```

Figura 11: código da função de exibição da árvore B.

A função “bfs()” realiza a busca em largura na árvore e printa seu resultado, ou seja, a partir da raiz vai colocando os nós encontrados na fila e atualizando o arquivo de log com os nós de um mesmo nível, por meio de um loop que utiliza a fila de âncora para administrar as passagens. A imagem abaixo ilustra a função de maneira compactada.

```
//Função para busca em largura na árvore B.
void bfs(int RRNbfs){
    int i;
    int nivel = 0;
    PAGINA busca;
    PAGINA pagBFS;
    PAGINA k;
    char *mensagem;
    Fila F;

    mensagem = malloc(30*sizeof(char));

    //Cria-se a fila para podermos utilizá-la
    criarFila(&F);

    //pegamos a página para começar a bfs
    pagBFS = getPagina(RRNbfs);
    //Se a função getPagina retornar erro, ou seja, uma página com número de chaves -1
    if(pagBFS.numeroChaves == -1){
        printf("Nao foi possivel encontrar a pagina raiz com RRN %d. Erro 0x3000.\n", RRNbfs);
    }

    //Colocamos a página inicial na fila
    push(&F, pagBFS, nivel);

    //Enquanto a fila não está vazia, ou seja, não percorremos todos os nós
    while(!empty(&F)){

        //Terminamos esvaziando a fila caso exista algum elemento ainda e liberando a memória utilizada
        esvaziaFila(&F);
        free(mensagem);
    }
}
```

Figura 12: código da função de busca em largura na árvore com alguns trechos omitidos (forma compactada).

Podemos verificar o loop que percorre todos os nós em detalhes.

```
//Enquanto a fila não está vazia, ou seja, não percorremos todos os nós
while(!empty(&F)){
    //Pegamos o nó da frente da fila e o nível respectivo
    k = pull(&F, &nivel);

    //Joga no arquivo de log o nível e o número de chaves
    sprintf(mensagem, "%d %d ", nivel, k.numeroChaves);
    atualizaArquivoDeLog(mensagem, LOG_MSG_OFF);

    //Esse for irá jogar todos os filhos na fila e colocará todas as chaves da página no arquivo de log
    for(i = 0; i <= k.numeroChaves; i++){
        //Se não extrapolamos o número máximo de chaves
        if(i != k.numeroChaves){
            //Joga no arquivo de log a chave e seu offset
            sprintf(mensagem, "< %d/%d> ", k.chaves[i][0], k.chaves[i][1]);
            atualizaArquivoDeLog(mensagem, LOG_MSG_OFF);
        }

        //Se existir filho, insere na fila
        if(k.filhos[i] != -1){
            //Pega o RRN referente ao filho
            int RRN = k.filhos[i];
            //E pega a página referente ao RRN
            busca = getPagina(RRN);
            //Se a busca deu erro, ou seja, retornou a página que era setada como "erro"
            if(busca.numeroChaves == -1){
                printf("Não foi possível encontrar a página com RRN %d. Erro 0x3001.\n", RRN);
                return;
            }
            //Insere a página filha na fila, com um nível somado
            push(&F, busca, nivel+1);
        }
    }
    //Somente para ter um \n no arquivo de log:
    atualizaArquivoDeLog("\n", LOG_MSG_OFF);
}
```

Figura 13: código da função de busca em largura na árvore com alguns trechos omitidos (forma compactada).

3. Exemplo de entrada

Para a seguinte entrada genérica de dados vamos verificar o uso de cada uma das funcionalidades. Abaixo segue a informação de todos os registos já existentes no arquivo:

tamanho|ID|titulo|genero|

| | |
|----------|----------|
| 7 1 a a | 8 14 a n |
| 7 2 a b | 8 15 o a |
| 7 9 a i | 8 16 a p |
| 8 10 a j | 8 17 a q |
| 8 11 a k | 8 30 a r |
| 7 3 a c | 8 31 a s |
| 7 4 a d | 8 32 a t |
| 7 5 a e | 8 33 a u |
| 7 6 a f | 8 34 a v |
| 7 7 a g | 8 29 a w |
| 7 8 a h | 8 28 a x |
| 8 12 a l | 8 27 a y |
| 8 13 a m | 8 35 a z |

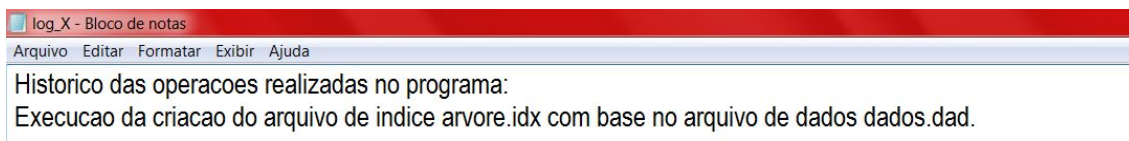
Inicialmente nos deparamos com a tela de menu, onde a funcionalidade desejada poderá ser escolhida através do input do dígito correspondente:

```
Arquivo de 'log' criado com sucesso.  
Trabalho Alg. II - Arvore B  
Digite o numero da opcao desejada:  
1. Criar indice.  
2. Inserir Musica.  
3. Pesquisar Musica por ID.  
4. Remover Musica por ID.  
5. Mostrar Arvore-B.  
6. Fechar o programa.
```

Figura 14: tela de menu, onde o usuário escolhe qual opção deseja executar.

1. Esta funcionalidade cria o arquivo de índices a partir do dados e como resultado são geradas as seguintes informações:

Se o arquivo de Log for aberto a seguinte mensagem foi adicionada



log_X - Bloco de notas
Arquivo Editar Formatar Exibir Ajuda
Historico das operacoes realizadas no programa:
Execucao da criacao do arquivo de indice arvore.idx com base no arquivo de dados dados.dad.

Figura 15: resultado do arquivo de Log quando da criação do arquivo de índice.

Se o arquivo de índice for aberto é possível verificar que os registros em forma binária podem ser encontrados seguindo a padrão de correteude.

2. Esta funcionalidade insere uma nova música.

Primeiramente, os dados referentes a nova música são solicitados

```
Inserir nova musica:  
Digite um numero inteiro com ID da musica:  
50  
Digite o titulo da musica:  
Exemplo  
Digite o genero da musica:  
MPB  
Arquivo de 'log' atualizado com sucesso.  
Arquivo de 'log' atualizado com sucesso.  
Arquivo dados.dad e arvore.idx atualizados com sucesso.
```

Figura 16: tela referente à solicitação da entrada pelo usuário, quando a opção 2 é selecionada.

Após a obtenção da música os arquivos de Log, Dados e Índice são atualizados e as seguintes saídas podem ser verificadas.

Se abrimos o arquivo de Log, a seguinte mensagem foi adicionada:

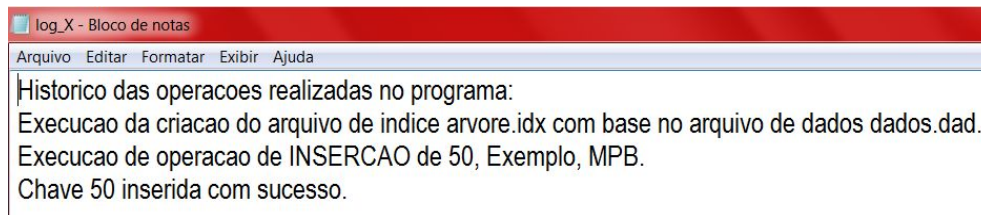


Figura 17: conteúdo do arquivo de Log resultante da execução das operações solicitadas.

Se abrirmos o arquivo de Dados podemos verificar que em seu final foi adicionado o novo registro, o mesmo vale para o de índices, só que não necessariamente no final.



Figura 18: conteúdo do arquivo de dados após a inserção de um novo registro.

3. Ao selecionar esta opção um ID a ser pesquisado é solicitado; o programa retorna a informação do que foi encontrado:

```
Digite um numero inteiro com ID da musica a ser pesquisada:
50
Registro encontrado:
id = 50
Titulo = Exemplo
Genero = MPB
Arquivo de 'log' atualizado com sucesso.
```

Figura 19: Resultado da pesquisa de um registro(música) por ID.

Se abrirmos o arquivo de Log podemos verificar a seguinte atualização:

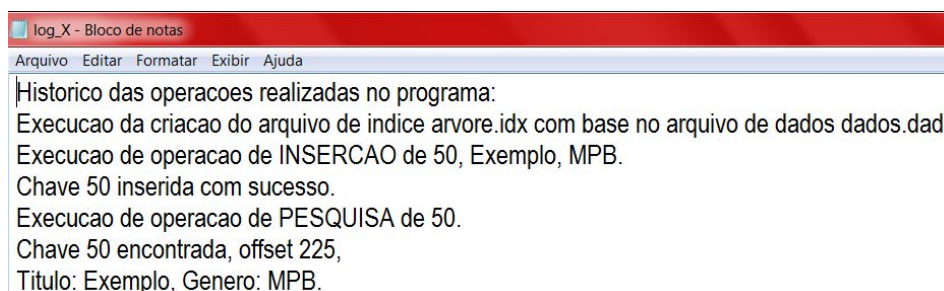
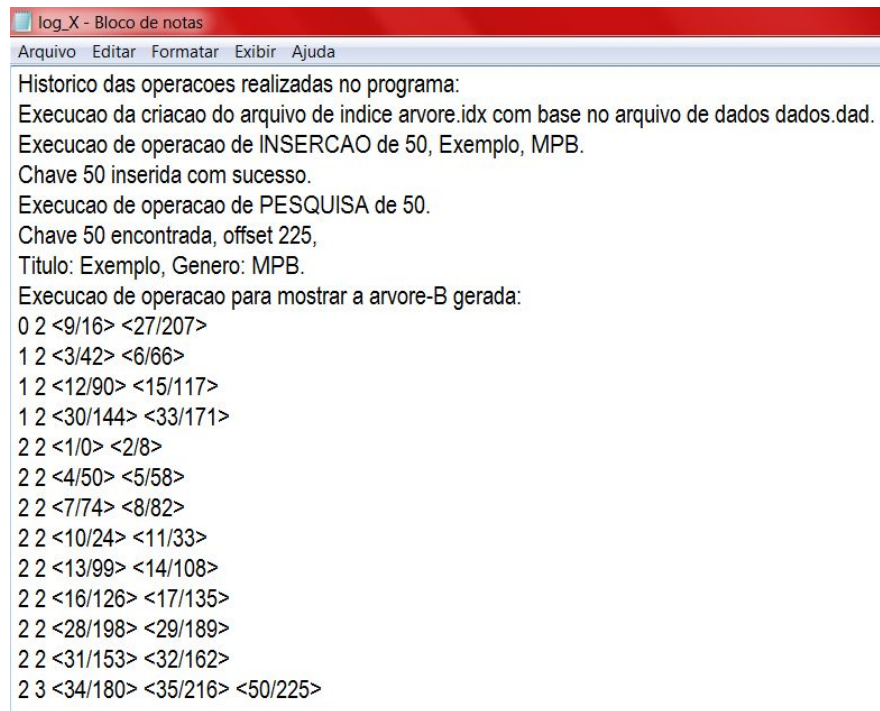


Figura 20: Arquivo de Log com atualização referente à operação de busca.

4. Não implementada.

5. Ao selecionar esta opção, o programa exibe a árvore B no formato exigido pelas especificações. Se abrirmos o arquivo de Log podemos verificar tal fato, pois na tela do prompt só será informado que a operação foi realizada.



```
log_X - Bloco de notas
Arquivo  Editar  Formatar  Exibir  Ajuda

Historico das operacoes realizadas no programa:
Execucao da criacao do arquivo de indice arvore.idx com base no arquivo de dados dados.dad.
Execucao de operacao de INSERCAO de 50, Exemplo, MPB.
Chave 50 inserida com sucesso.
Execucao de operacao de PESQUISA de 50.
Chave 50 encontrada, offset 225,
Titulo: Exemplo, Genero: MPB.
Execucao de operacao para mostrar a arvore-B gerada:
0 2 <9/16> <27/207>
1 2 <3/42> <6/66>
1 2 <12/90> <15/117>
1 2 <30/144> <33/171>
2 2 <1/0> <2/8>
2 2 <4/50> <5/58>
2 2 <7/74> <8/82>
2 2 <10/24> <11/33>
2 2 <13/99> <14/108>
2 2 <16/126> <17/135>
2 2 <28/198> <29/189>
2 2 <31/153> <32/162>
2 3 <34/180> <35/216> <50/225>
```

Figura 21: Resultado da operação de exibição da árvore-B no arquivo de Log.

6. Ao selecionar esta opção no menu, a execução do programa é finalizada e o prompt é fechado.

4. Instruções de execução e compilação

Durante a implementação, os sistemas operacionais e as IDEs utilizados variam um pouco, porém o programa funciona tanto em Linux quanto em Windows, com a exceção de que no Linux o terminal irá constantemente mostrar a mensagem de que não reconhece o comando para limpar os textos - isso se deve ao fato de que o Linux reconhece o comando “system(“clear”)” e não o comando “system(“cls”)” usado pelo Windows.

- **IDEs utilizadas:** Code:Blocks 16.01 e Sublime Text 3.0.
- **Sistemas operacionais:** Manjaro Linux, Windows 7 Ultimate e Windows 10 Home Basic.
- **Bibliotecas necessárias:** <stdio.h>, <stdlib.h> e <string.h>
- **Arquivos do programa:** “main.c”, “TAD.h”, “TAD_arvore_B.h”, “TAD_fila.h”, “TAD_log.h”, “utilidades.h”, “dados.dad”, “arvore.idx” e “log_abellas.txt”.

5. Estimativas de tempo e espaço

Com relação ao espaço gasto pela implementação, diz respeito aos bytes ocupados pelos arquivos de dados, índice e Log, que representam as informações em disco do suposto banco de dados (dados.dad), do arquivo da árvore (arvore.idx) e de um arquivo de histórico de operações (log_abellas.txt). Como esperado, devido ao uso de um Índice, organizado em forma de árvore-B, o número de acessos ao disco para localizar uma informação será reduzido, já que a árvore nos possibilita uma abordagem de busca mais eficiente. A literatura permite estimar alguns valores de tempo e ordem de complexidade.

No pior caso a inserção temos:

$$O(\log_{\lceil m/2 \rceil} N)$$

No pior caso da busca temos:

$$O(\log_{\lceil m/2 \rceil} N)$$

Onde

$m \rightarrow$ ordem da árvore-B

$N \rightarrow$ número de chaves até o nível das folhas (N é o total de chaves armazenadas)

Em relação ao uso de memória, sabendo que o tamanho de uma página de uma árvore B de ordem 5 é igual a 60 bytes no nosso programa (equivalente a 15 *ints* em memória), na inserção será utilizada o tamanho de uma página da árvore (que depende da ordem da árvore) e do tamanho do registro (que tem tamanho variável), e na busca, somente dependerá do tamanho de uma página da árvore.