UNIVERSIDADE FEDERAL DE OURO PRETO

DECOM

Arthur Henrique Santos Celestino 21.1.4019

Caio silas de Araujo Amaro 21.1.4111

Mateus Viana Maschietto 21.1.4020

Henrique Dantas Pighini 21.1.4025

Trabalho Prático 1 - Pesquisa Externa

Ouro Preto - MG

2022

Sumário

1) Introdução:

Quando a memória principal não suporta os dados, surge a necessidade do uso da memória secundária. Portanto, foi necessário a criação de métodos utilizados para pesquisa em memória externa, tais como, acesso sequencial indexado, árvore binária, árvore B e árvore B*. Essa necessidade surge, pois, normalmente se trabalha com um volume de dados muito grande, e apenas ler os dados dos arquivos de maneira simples deixa o processo muito lento, pois o acesso a disco é mais demorado que um acesso a memória interna. Cada um dos métodos citados acima, tem uma estratégia diferente para ler e pesquisar dados em arquivos, visando tornar o processo mais rápido.

A partir disso, com o objetivo de estudar a complexidade dos algoritmos acima, implementamos um código que contém esses.

A criação desses arquivos contou com a utilização das seguintes TADs:

```c
typedef struct {
    int chave;
    long int dado1;
    char dado2[1000];
    char dado3[5000];
} Reg;
```

```c
typedef struct {
    int num_trans;
    int num_comp;
    double temp_exec;
    int valido;
} Comp;
```

```c
typedef struct{
    int chave;
    int posicaoArq;
} Indice;
```

Além disso, como proposto pelo professor e doutor Guilherme Tavares de Assis, o programa foi implementado de maneira que sua execução ocorre através da seguinte linha de comando.

```
./nomeExecutavel <método> <quantidade> <situação> <chave> [-P]
```

- <método> representa o método de pesquisa externa a ser executado, podendo ser um número inteiro de 1 a 4, de acordo com a ordem dos métodos mencionados;
- <quantidade> representa a quantidade de registros do arquivo considerado;
- <situação> representa a situação de ordem do arquivo, podendo ser 1 (arquivo ordenado ascendentemente), 2 (arquivo ordenado descendentemente) ou 3 (arquivo desordenado aleatoriamente);
- <chave> representa a chave a ser pesquisada no arquivo considerado;
- [-P] representa um argumento opcional que deve ser colocado quando se deseja que as chaves de pesquisa dos registros do arquivo considerado sejam apresentadas na tela.

Com o objetivo de analisar os algoritmos, consideramos os seguintes quesitos:

- Número de transferências (leitura) entre as memórias interna e externa;
- Número de comparações entre as chaves de pesquisa;
- Tempo de execução do código.

Para compilar nosso código, usamos as seguintes diretivas de compilação (usando o terminal do Linux, pois alguns comandos e execução do código apresentam leves erros no Windows):

$ gcc main.c acessoSequencialIndexado.c arq_binary.c arvoreB.c arvoreBE.c -o programa -Wall
$ ./programa <metodo> <quantidade> <situação> <chave>

Para geração dos arquivos, usamos um programa gerador criador por nós mesmos, que se encontra presente na pasta GeraArquivo, simplesmente executando o arquivo .c presente na mesma. Esse arquivo são 3 mains, cada um gerando um dos tipos de arquivo, onde apenas comentamos os 2 que não estavam em uso no momento e executávamos apenas um. Após gerado, jogamos todos os arquivos na pagina do main do programa principal.

2) Análise dos Quesitos:

2.1 Acesso sequencial indexado

Este método está ligado a pesquisa sequencial, em que a menor chave de cada página é armazenada no vetor.

```
Indice *tabela = malloc(sizeof(Indice) * tamanhoTabela);

while (posicao < tamanhoTabela)
{
  fread(&x, sizeof(x), 1, arquivo);
  fseek(arquivo, (sizeof(x) * (ITENSPAGINA - 1)), SEEK_CUR);
  tabela[posicao].chave = x.chave;
  resultado[0].num_trans++;
  posicao++;
}
```

Agora, utilizando o vetor acima, nós somos capazes de encontrar em qual página o item está contido. Essa pesquisa é feita verificando se a chave do item é maior que a chave de menor valor da página. Assim, no momento em que o valor for menor ou igual a pagina que ele está contido terá sido encontrado:

```
while(i < tamanhoTabela && tabela[i].chave <= item->chave) i++;
```

Ademais, com a página achada, nós navegamos até a parte do arquivo que faz referência a essa página e lemos os itens dela (página):

```
desloc = (i - 1) * ITENSPAGINA * sizeof(Reg);
fseek(arquivo, desloc, SEEK_SET);
fread(&pagina, sizeof(Reg), quantidadeItens, arquivo);
```

Por fim, passamos por essa página e procuramos o item dentro dela, caso o registro seja achado, o mesmo será retornado por referência pela função:

```
for(i = 0; i < quantidadeItens; i++) {
  resultado[1].num_comp++;
  if(pagina[i].chave == item->chave) {
    *item = pagina[i];
    resultado[1].valido = 0;
    break;
  }
}
```

Se não, o valor de valido não será alterado e retornará para o main dizendo que não foi possível encontrar o item.

2.2 Árvore Binária

A Árvore de pesquisa binária é um método que irá pegar a ideia de uma árvore binária normal e modificará sua base para ser manipulável em um arquivo.

Para a criação desse método de pesquisa vamos precisar de uma maneira de representar uma árvore em um arquivo binário, e para isto, foi criado uma TAD com elementos necessários para realizar tal necessidade:

```
typedef struct{
    //Índice que salva a chave e a posição de memória
    Indice indice;
    int F_esq, F_dir;

} ArvoreExt;
```

Ao montar a árvore os valores de F_esq e F_dir são inicializados com -1 e em seguida é conferido se a chave é maior ou menor para fazer uma chamada recursiva para inserir o índice, se a chave for igual ao índice isso quer dizer que o elemento já foi inserido na árvore.

Para fazer a chamada recursiva utilizamos a função fseek para acessarmos a posição do elemento atual no arquivo binário criado e pegar a posição do filho:

```
if(aux.indice.chave > index.chave && aux.F_esq != -1){
    fseek(arq, aux.F_esq*sizeof(ArvoreExt), SEEK_SET);
    conta.num_comp++;

    Comp temp = InsertArvoreExterna(arq, index);
    conta.num_trans += temp.num_trans;
    conta.num_comp += temp.num_comp;
    conta.valido = temp.valido;

    return conta;
}
if(aux.indice.chave < index.chave && aux.F_dir != -1){
    fseek(arq, aux.F_dir*sizeof(ArvoreExt), SEEK_SET);
    conta.num_comp++;

    Comp temp = InsertArvoreExterna(arq, index);
    conta.num_trans += temp.num_trans;
    conta.num_comp += temp.num_comp;
    conta.valido = temp.valido;

    return conta;
}
```

Após a chamada recursiva chegar em um nó em que o filho está inválido (-1) a função não entrará na recursão e vai para a parte que realiza a inserção do elemento no arquivo que representa nossa árvore binária.

Para realizar o que foi dito acima, vamos precisar atualizar o valor do filho do pai como o fim do arquivo (já que a inserção no arquivo é sempre e estritamente no final).

```
if(aux.indice.chave > index.chave){
    conta.num_comp++;
    aux.F_esq = ftell(arq)/sizeof(ArvoreExt);
}

else{
    conta.num_comp++;
    aux.F_dir = ftell(arq)/sizeof(ArvoreExt);
}
```

Após alterar os valores internamente, é necessário registrar o novo filho e a posição dele no pai dentro do arquivo.

Usando uma posição previamente registrada do pai, é inserido no fim do arquivo o filho e após darmos um fseek para a posição do pai sobrescrevemos o valor de seus filhos mantendo assim a árvore binária externa correta:

```
    fwrite(&inserido, sizeof(ArvoreExt), 1, arq);

    fseek(arq, tmp*sizeof(ArvoreExt), SEEK_SET);
    fwrite(&aux, sizeof(ArvoreExt), 1, arq);
```

Na função de Search Árvore Externa é feita uma procura pela chave, se a chave procurada for igual ao índice lido da árvore externa, o índice é atualizado e é retornado, se não, é checado se a chave e maior ou menor e o ponteiro do arquivo é atualizado de acordo com isso através do fseek utilizando o valor dos filhos como parâmetro, caso essa chave não seja encontrada significa que ela não está presente na árvore.

```
if(aux.indice.chave == chave){
    *index = aux.indice;
    return conta;
}

conta.num_comp++;
if(aux.indice.chave > chave && aux.F_esq != -1){
    fseek(arq, aux.F_esq*sizeof(ArvoreExt), SEEK_SET);

    Comp temp = SearchArvoreExterna(arq, chave, index);
    conta.num_trans += temp.num_trans;
    conta.num_comp += temp.num_comp;
    conta.valido = temp.valido;

    return conta;
}
else if(aux.indice.chave < chave && aux.F_dir != -1){
    fseek(arq, aux.F_dir*sizeof(ArvoreExt), SEEK_SET);

    Comp temp = SearchArvoreExterna(arq, chave, index);
    conta.num_trans += temp.num_trans;
    conta.num_comp += temp.num_comp;
    conta.valido = temp.valido;

    return conta;
}
```

2.3 Árvore B

A árvore B é uma árvore de ordem M, que possui como propriedades:

- A página raiz contém entre 1 a 2M itens;
- As demais páginas possuem, no mínimo, M itens e M+1 descendentes e, no máximo, 2M itens e 2M+1 descendentes;
- As páginas folhas estão todas no mesmo nível;
- Os registros estão ordenados de forma crescente dentro das páginas.

O código da árvore B já nos foi disponibilizado nos slides da matéria de Estruturas de Dados II, porém, foram necessárias algumas mudanças para que fosse possível realizar as pesquisas condizentes com o enunciado do trabalho.

A TAD criada para representar a árvore possui um vetor de registros do tipo Índice e um vetor de apontadores do tipo Apontador, além de um contador para guardar a quantidade de registros presentes na página da árvore.

```c
#define M 5

typedef long TipoChave;

typedef struct TipoRegistro{
    TipoChave chave;
}TipoRegistro;

typedef struct Indice {
    TipoChave chave;
    int posicaoArquivo;
} IndiceB;

typedef struct TipoPagina* TipoApontador;

typedef struct TipoPagina{
    int n;
    IndiceB itens[2*M];
    TipoApontador descendentes[2*M + 1];
}TipoPagina;
```

E em vez de utilizarmos o TipoRegistro tivemos que utilizar o tipo Reg que possui os campos requisitados no TP 1 de pesquisa externa:

```c
//Struct dos registros com os campos requisitados no enunciado do TP
typedef struct {
    int chave;
    long int dado1;
    char dado2[1000];
    char dado3[5000];
} Reg;
```

O método de inserção foi dividido em três funções.

```c
void InsereNaPagina(TipoApontador ApAtual, IndiceB Reg, TipoApontador ApDir, Comp *contagem);
void Ins(IndiceB Registro, TipoApontador Ap, int *Cresceu, IndiceB *RegRecursao, TipoApontador *ApRecursao, Comp *contagem);
void Insere(IndiceB Registro, TipoApontador *Ap, Comp *contagem);
```

A inserção tem início na função insere, que recebe o registro a ser inserido na árvore e o apontador para a página raiz da mesma. Em seguida, há a chamada função Ins que recursivamente acessa a árvore B. Após essa chamada é verificado se a raiz cresceu e se é necessário criar uma nova página:

```c
void Insere(IndiceB Registro, TipoApontador *Ap, Comp *contagem){ //Funçao que começa o processo de inserção
    int Cresceu;
    IndiceB RegRetorno;
    TipoPagina *ApRetorno, *ApAux;

    Ins(Registro, *Ap, &Cresceu, &RegRetorno, &ApRetorno, contagem);
    if(Cresceu) //Arvore crescendo um nivel
    {
        ApAux = (TipoPagina*) malloc(sizeof(TipoPagina));
        ApAux->n = 1;
        ApAux->itens[0] = RegRetorno;
        ApAux->descendentes[1] = ApRetorno;
        ApAux->descendentes[0] = *Ap;
        *Ap = ApAux;
    }
}
```

A função Ins procura a posição onde deverá ser inserido o novo registro. Se a árvore for nova e não ter elementos na primeira vez que rodar irá parar neste if e será criado a primeira página da árvore após retornar para a função insere com o cresceu = 1

```
if(Ap == NULL){ //Atualizando o cresceu, que indica se é preciso fazer uma divisão naquele nivel, aumentando a altura
    *Cresceu = 1;
    (*RegRecursao) = Registro;
    (*ApRecursao) = NULL;
    return;
}
```

Se tiver mais elementos na árvore será necessário encontrar a posição que vamos inserir o registro.

No código abaixo, a variável i armazena a posição a se inserir o novo registro, ou a página a ser apontada para que se encontre a posição para inserção.

E após encontrarmos tal i vamos verificar se o elemento já existe e se não existir vamos fazer alterações no valor encontrado para direcionarmos o nosso ponteiro.

```
while(i < Ap->n && Registro.chave > Ap->itens[i-1].chave){
    i++;
    /*contagem->num_compP++;*/
    contagem[1].num_comp++;
} //Verificando os itens que estao numa pagina

// contagem->num_compC++;
contagem[0].num_comp++;
if(Registro.chave == Ap->itens[i-1].chave){ //Tentando inserir um item repetido
    printf("Erro: Registro ja presente\n");
    *Cresceu = 0;
    return;
}

// contagem->num_compC++;
contagem[0].num_comp++;
if(Registro.chave < Ap->itens[i-1].chave){
    i--;
    /*contagem->num_compC++;*/
    contagem[0].num_comp++;
}
```

Em seguida, a função realiza chamadas recursivas para encontrar a página folha correta para a inserção do novo registro. Depois, a função verifica se há espaço na página para armazenar o registro. Caso haja espaço na página, o novo registro é inserido nela, chamando a função InsereNaPagina. Caso a página onde deverá ser inscrito o registro estiver cheia, cria-se uma nova página e metade dos registros presentes na página cheia são transferidos para esta nova e o registro do meio é transferido para a página pai das duas páginas:

```
if(Ap->n < 2*M){ //Pagina tem espaço
    InsereNaPagina(Ap, *RegRecursao, *ApRecursao, contagem);
    *Cresceu = 0;
    return;
}

//Overflow: Pagina tem que ser dividida
ApontadorAux = (TipoApontador) malloc(sizeof(TipoPagina));
ApontadorAux->n = 0;
ApontadorAux->descendentes[0] = NULL;

if(i < M+1){
    InsereNaPagina(ApontadorAux, Ap->itens[2*M - 1], Ap->descendentes[2*M], contagem);
    Ap->n--;
    InsereNaPagina(Ap, *RegRecursao, *ApRecursao, contagem);
}
else
    InsereNaPagina(ApontadorAux, *RegRecursao, *ApRecursao, contagem);

for(j = M+2; j <= 2*M; j++)
    InsereNaPagina(ApontadorAux, Ap->itens[j-1], Ap->descendentes[j], contagem);

Ap->n = M; ApontadorAux->descendentes[0] = Ap->descendentes[M+1];
*RegRecursao = Ap->itens[M]; *ApRecursao = ApontadorAux;
```

Após ser realizado a inserção os valores de RegRecursao e ApRecusrsao são alterados para que o elemento que vai subir para ser inserido seja facilmente adicionado nos níveis acima.

A função InsereNaPagina insere o novo registro na página enviada como referência.

Essa inserção é realizada de maneira iterativa de maneira que até que seja encontrada a posição que será inserido o novo elemento, vamos empurrando os registros e os ponteiros assim mantendo a propriedade da árvore B e não perdendo os ponteiros

```c
void InsereNaPagina(TipoApontador ApAtual, IndiceB Reg, TipoApontador ApDir, Comp *contagem){
    int NaoAchouPosicao;
    int k;
    k = ApAtual->n; NaoAchouPosicao = (k > 0);

    while(NaoAchouPosicao){ //Verificando se tem espaço na pagina para inserir
        // contagem->num_compC++;
        contagem[0].num_comp++;
        if(Reg.chave >=  ApAtual->itens[k-1].chave){
            NaoAchouPosicao = 0;
            break;
        }
        ApAtual->itens[k] = ApAtual->itens[k-1]; //Arredando os itens e ponteiros
        ApAtual->descendentes[k+1] = ApAtual->descendentes[k];
        k--;
        if(k < 1) NaoAchouPosicao = 0;
    }
    ApAtual->itens[k] = Reg;
    ApAtual->descendentes[k+1] = ApDir;
    ApAtual->n++;
}
```

A função pesquisa vai procurar recursivamente o elemento de maneira similar à inserção e vai retornar válido e o elemento se for encontrado ou invalido se não for possível encontrar:

```c
int Pesquisa(Reg* x, TipoApontador Apontador, Comp *contagem, FILE *arquivo){ //Função de pesquisa da arvore
    int i = 1;

    if(Apontador == NULL){
        return 1;
    } //Chegou em uma folha e não achou o elemento

    while(i < Apontador->n && x->chave > Apontador->itens[i-1].chave) {
        i++;
        /*contagem->num_compP++;*/
        contagem[1].num_comp++;
    }

    // contagem->num_compP++;
    contagem[1].num_comp++;
    if(x->chave == Apontador->itens[i-1].chave){ //Achou o item
        // contagem->num_transP++;
        contagem[1].num_trans++;
        fseek(arquivo, Apontador->itens[i-1].posicaoArquivo * sizeof(Reg), SEEK_SET);
        fread(x, sizeof(Reg), 1, arquivo);
        //Pego a posição no arquivo do item achado, e ai sim dou fread no arquivo desse item para a varaivel x
        return 0;
    }
    // contagem->num_compP++;
    contagem[1].num_comp++;
    if(x->chave < Apontador->itens[i-1].chave) //Chamando para os lados, se a chave for maior ou menor
        Pesquisa(x,Apontador->descendentes[i-1], contagem, arquivo);
    else
        Pesquisa(x, Apontador->descendentes[i], contagem, arquivo);
    return 0;
}
```

2.4 ÁRVORE B*

O método árvore B* é uma melhoria da árvore B, com algumas características diferentes. Será mantido a ideia do limite de valores em uma página e descendentes, porém, teremos páginas internas e externas na nossa árvore.

As páginas internas de uma árvore B* funcionam exatamente como as páginas de uma árvore B, entretanto, é nas páginas externas que o diferencial desse método de pesquisa externa será relevante. Os itens úteis de uma B* serão todos encontrados em suas folhas que serão externas, isso significa que todos os itens que você deseja procurar estarão em uma página externa e que sua busca só irá parar quando for encontrado a mesma. Desta forma, a árvore B* possuirá todos os registros do arquivo em memória interna, porém, será fácil acessar a pesquisa já que a mesma possuirá a estrutura de uma árvore B.

Devido a esses diferenças, tivemos que implementar novas funções e refatorar outra TAD, elas estão abaixo, respectivamente:

```
typedef enum{
    Interna,
    Externa
} TipoNodo;
```

```
typedef struct TipoPagina{
    TipoNodo seletor;
    union {
        struct {
            int nInterno;
            int chavesInternas[2*M];
            TipoApontador descendentesInternos[2*M + 1];
        } Inter;
        struct {
            int nExterno;
            Reg registrosExternos[2*M];
        } Exter;
    } Paginas;
```

A função de inserção (InsereBE), caso a árvore esteja vazia ele fará primeiro a criação de uma página externa.

```
if(*Ap == NULL){
    ApAux = (TipoPagina*) malloc(sizeof(TipoPagina));
    ApAux->seletor = Externa;
    ApAux->Paginas.Exter.nExterno = 0;
    *Ap = ApAux;
}
```

Após isso, é importante lembrar da propriedade das páginas da árvore B*. Caso ela seja interna seu funcionamento será o mesmo que vimos anteriormente, porém, caso ela seja externa, seu funcionamento será diferente. Nesse sentido, começamos procurando pela posição que o elemento será inserido ou o ponteiro que será acessado na árvore:

```
while(
    i < Ap->Paginas.Exter.nExterno &&
    Registro.chave > Ap->Paginas.Exter.registrosExternos[i-1].chave
) {
    i++;
    contagem[0].num_comp++;
} //Verificando os itens que estao numa pagina
```

Após chegarmos em uma página externa, verificamos se existe espaço livre para a inserção e se for possível inserir este será inserido sem necessidade de manutenção da árvore:

```
if(Ap->Paginas.Exter.nExterno < 2*M){ //Pagina tem
    InsereNaPaginaExt(Ap, *RegRecursao, contagem);
    *Cresceu = 0;
    return;
}
```

Caso contrário, criaremos uma nova página externa.

```
ApontadorAux = (TipoApontadorE) malloc(sizeof(TipoPaginaE));
ApontadorAux->seletor = Externa;
ApontadorAux->Paginas.Exter.nExterno = 0;
Reg aux = Ap->Paginas.Exter.registrosExternos[M];
```

E a partir do valor da posição que foi calculado anteriormente será possível identificar se o novo valor será inserido na nova página ou se continuará na mesma. Ademais, os valores que necessitam de ser transferidos para a nova página vão ser inseridos no “for” e os ponteiros que retornaram na recursão para a manutenção da árvore.

```
if(i < M+1){
    InsereNaPaginaExt(ApontadorAux, Ap->Paginas.Exter.registrosExternos[2*M - 1], contagem);
    Ap->Paginas.Exter.nExterno--;
    InsereNaPaginaExt(Ap, *RegRecursao, contagem);
}
else{
    InsereNaPaginaExt(ApontadorAux, *RegRecursao, contagem);
}
for(j = M+2; j <= 2*M; j++)
    InsereNaPaginaExt(ApontadorAux, Ap->Paginas.Exter.registrosExternos[j-1], contagem);

Ap->Paginas.Exter.nExterno = M + 1;
*RegRecursao = Ap->Paginas.Exter.registrosExternos[M]; *ApRecursao = ApontadorAux;
```

Voltando a função principal da árvore B*, nós chamamos a função que pesquisa dentro da árvore B*(pesquisaBE). No início ela funciona igual a pesquisaB, até chegar à página folha, onde ela funciona de maneira sequencial.

```
while(
    i <= PaginaAtual->Paginas.Exter.nExterno &&
    x->chave > PaginaAtual->Paginas.Exter.registrosExternos[i-1].chave
){
    i++;
    contagem[1].num_comp++;
}
```

3) Análise dos Resultados:

Foram realizados alguns testes para determinar o desempenho de cada método, de forma a conseguir demonstrar maior eficiência de um método em relação a outro:

Pesquisa Sequencial Indexada

A pesquisa sequencial indexada só é possível quando o arquivo está ordenado, assim, só foi possível realizar testes concretos do arquivo em ordem crescente:

Qtd. Registros		100	1000	10000	100000	1000000
Criação	Transferencia	25	250	2500	25000	250000
	Comparação	0	0	0	0	0
	Tempo (s)	0	0	0,046875	0,859375	7,6875
Pesquisa	Transferencia	1	1	1	1	1
	Comparação	16	127	1252	12502	125002
	Tempo (s)	0	0	0	0	0

A partir dos dados analisados, percebemos que o tempo de criação cresce rapidamente, considerando que nos primeiros casos o processo foi tão rápido que foi imperceptível.

Além disso, aumenta exponencialmente pois com o aumento do número de páginas, surge o aumento do número de comparações das chaves. Isso durante a pesquisa.

Como o acesso sequencial indexado é limitado à organização ascendente do arquivo, ele se torna menos eficiente, já que ele não funciona com qualquer tipo de arquivo, necessitando de uma organização posterior que vai aumentar o custo da pesquisa.

Árvore Binária Com Arquivo Ascendente

Qtd. Registros		100	1000	10000	100000	1000000
Criação	Transferencia	5051	500501	50005001	x	x
	Comparação	4950	499500	49995000	x	x
	Tempo (s)	0,015625	0,067188	61,1875	x	x
Pesquisa	Transferencia	100	1000	10000	x	x
	Comparação	150	1500	15000	x	x
	Tempo (s)	0	0	0,015625	x	x

Como a árvore binária não consegue se manter balanceada, ela se torna menos benéfica em arquivos que foram ordenados de modo crescente ou decrescente, já que ao pesquisar por um valor ela faz uma pesquisa quadrática, o que não é a intuição ao usar uma árvore binária.

Para concluir, é possível notar que nos casos de entrada extensos não foi possível concluir o código, já que ele demandou mais que a máquina que o estava compilando era capaz de processar.

Árvore Binária Com Arquivo Descendente

Qtd. Registros		100	1000	10000	100000	1000000
Criação	Transferencia	5051	500501	50005001	x	x
	Comparação	4950	499500	49995000	x	x
	Tempo (s)	0,015625	0,067188	61,1875	x	x
Pesquisa	Transferencia	100	1000	10000	x	x
	Comparação	150	1500	15000	x	x
	Tempo (s)	0	0	0,015625	x	x

De maneira análoga a árvore binária com arquivo ascendente, ele mantém os mesmos defeitos tanto que as tabelas de ambos são iguais

Árvore Binária Com Arquivo Aleatório

Qtd. Registros		100	1000	10000	100000	1000000
Criação	Transferencia	870	13438	181626	2275639	27355244
	Comparação	769	12437	171625	2175638	26355243
	Tempo (s)	0,015625	0,078125	1,0625	14,015625	206,50
Pesquisa	Transferencia	3	9	13	24	26
	Comparação	6	18	26	48	51
	Tempo (s)	0	0	0	0	0

No caso em que a entrada contém valores não ordenados, o método é muito mais eficiente, já que, mesmo que sem o balanceamento, ela é uma árvore de pesquisa. Nesse sentido, esse método propõe um menor número de comparações e tempo de pesquisa.

Árvore B Com Arquivo Ascendente

Qtd. Registros		100	1000	10000	100000	1000000
Criação	Transferencia	100	1000	10000	100000	1000000
	Comparação	1706	27750	385894	4988106	60629110
	Tempo (s)	0	0,0156	0,046875	0,453125	1,65625
Pesquisa	Transferencia	1	1	1	1	1
	Comparação	9	16	22	26	39
	Tempo (s)	0	0	0	0	0

A quantidade de transferências durante a criação dos índices é igual ao número de chaves e, a porcentagem entre o número de chaves e a quantidade de comparações do índice vai aumentando à medida que a quantidade de chave aumenta. Deste modo, para casos maiores fica inviável a reprodução. Com relação às comparações na pesquisa, em todos os casos com árvore B e B* que virão abaixo, temos uma quantidade baixa pois como a árvore sempre é bem balanceada, vão sendo a cada comparação, cortados uma grande parte dos elementos da mesma.

Árvore B Com Arquivo Descendente

Qtd. Registros		100	1000	10000	100000	1000000
Criação	Transferencia	100	1000	10000	100000	1000000
	Comparação	1706	27750	385894	4988106	60629110
	Tempo (s)	0	0,0156	0,046875	0,453125	1,65625
Pesquisa	Transferencia	1	1	1	1	1
	Comparação	9	16	22	26	39
	Tempo (s)	0	0	0	0	0

Pôr a árvore B ser balanceada o método ascendente e descendente do código apresenta o mesmo resultado.

Árvore B Com Arquivo Aleatório

Qtd. Registros		100	1000	10000	100000	1000000
Criação	Transferencia	100	1000	10000	100000	1000000
	Comparação	1661	23793	309916	3817897	44960728
	Tempo (s)	0	0,015625	0,125	0,296875	1,5625
Pesquisa	Transferencia	1	1	1	1	1
	Comparação	8	16	21	27	38
	Tempo (s)	0	0	0	0	0

A quantidade de transferências e comparações para a criação da árvore B se mantém alta, mas a pesquisa continua sendo feita de forma eficiente e rápida. Com isso podemos concluir que a natureza do arquivo em questão não é muito relevante para a execução da árvore B, já que o fator de maior influência é a quantidade de registros considerados.

Árvore B* Com Arquivo Ascendente

Qtd. Registros		100	1000	10000	100000	1000000
Criação	Transferencia	100	1000	10000	100000	1000000
	Comparação	1103	17289	235567	2991698	36241650
	Tempo (s)	0,022995	0,19845	1,56034	7,18232	24,48437
Pesquisa	Transferencia	0	0	0	0	0
	Comparação	11	12	14	15	24
	Tempo (s)	0,000001	0,000002	0,000005	0,000005	0,0022140

Para a árvore B*, os resultados foram relativamente parecidos com o da árvore B normal, pelo menos em número de comparações, porém como estamos fazendo a árvore em memória interna, e nesse tipo de árvore os registros tem que ficar nas páginas mais externas, a criação da árvore ficou lenta, já que são registros grandes.

Árvore B* Com Arquivo Descendente

Qtd. Registros		100	1000	10000	100000	1000000
Criação	Transferencia	100	1000	10000	100000	1000000
	Comparação	1215	16757	211151	2538670	29624077
	Tempo (s)	0,027	0,232051	1,8496	11,5633	24,015600
Pesquisa	Transferencia	0	0	0	0	0
	Comparação	3	6	8	10	21
	Tempo (s)	0,000002	0,000002	0,000002	0,000003	0,002

Para os registros de forma descendente, conseguimos resultados muito semelhantes com relação ao tempo se comparado a aplicação no arquivo ascendente. Na pesquisa tivemos menos comparações porque como nesse tipo de algoritmo, a chave do elemento de uma página externa pode ser a mesma de uma chave da árvore interna, temos algumas comparações a mais de acordo com a implementação escolhida (se o elemento repetido vai para o filho a esquerda ou direita).

Árvore B* Com Arquivo Aleatório

Qtd. Registros		100	1000	10000	100000	1000000
Criação	Transferencia	100	1000	10000	100000	1000000
	Comparação	1169	17281	222259	2742806	32308799
	Tempo (s)	0,037185	0,185680	2,08046	13,365400	43,82823
Pesquisa	Transferencia	0	0	0	0	0
	Comparação	3	7	8	12	23
	Tempo (s)	0,000001	0,000002	0,000002	0,000004	0,00004

Por fim, no arquivo aleatório, temos no geral uma média de tempos e comparações razoável, mas que não foge muito dos casos anteriores. Ou seja, na árvore B e B*, como são balanceadas, a condição do arquivo não influencia tanto quanto nos outros métodos apresentados.

4) Conclusão:

Neste trabalho, foi criada uma aplicação capaz de executar 4 métodos diferentes de pesquisa externa, juntamente com 5 quantidades de chaves diferentes (100, 1.000, 10.000, 100.000 e 1.000.000) e 3 métodos de ordenação (crescente, decrescente e aleatório).

No entanto, devido ao grande uso da memória em alguns casos, não foi possível com que o grupo rodasse todos os testes em tempo hábil, tendo alguns membros não conseguindo rodar nenhum dos maiores testes por erro de memória.

Deste modo, podemos tirar conclusões da importância dos métodos de organização e os seus desempenhos. Generalizando, nossa árvore B foi a melhor para as três situações de arquivos com muitos registros, pois ela é sempre balanceada e otimizada, desse modo, sempre vão sendo eliminados metade dos registros (daí sua complexidade logarítmica), gerando assim um tempo total muito pequeno. Tivemos uma discrepância grande entre ela e a B* pois fizemos a B utilizando apenas os índices, acessando essa posição do item pesquisado no arquivo depois, não precisando inserir o registro inteiro, economizando muita memória e processamento. Para entradas de poucos dados, a árvore binária também se torna bem eficiente, tanto em número de comparações quanto em tempo, porém se torna muito custosa em entradas grandes, já que a cada elemento, precisa ler a árvore toda novamente

Tivemos como principais dificuldades, a implementação da B*, principalmente quando estávamos fazendo as divisões das páginas externas e passando isso para um outro tipo de página (no caso a interna). Diversas vezes a árvore estava sendo dividida errada, gerando erros de busca e falhas. Também tivemos problemas com a árvore binária, pois como é um método totalmente baseado em memória externa, foi necessário mexer com uma grande quantidade de ponteiros dentro do arquivo, que combinado a uma recursão, nos deixou perdido na leitura e análise do algoritmo.