Universidade Federal de Ouro Preto - UFOP

Instituto de Ciências Exatas e Biológicas - ICEB

Departamento de Computação - DECOM

Ciência da Computação

Métodos de pesquisa externa

BCC203 - Estrutura de Dados II

Felipe Braz, Lucas Chagas, Matheus Peixoto, Nicolas Mendes, Pedro Henrique, Pedro Morais

Professor: Guilherme Tavares de Assis

Ouro Preto
25 de junho de 2023

Sumário

Lista de Códigos Fonte

1 Introdução

Este trabalho prático tem como objetivo realizar um estudo da complexidade de desempenho de diferentes métodos de pesquisa externa. Os métodos que foram implementados, são os que foram ensinados em sala de aula. Sendo eles: Acesso sequencial indexado, árvore binária de pesquisa adequada à memória externa, Árvore B e Árvore B*, sendo os dois últimos com os dados carregados em memória interna.

1.1 Considerações iniciais

Algumas ferramentas foram utilizadas durante a criação deste projeto:

- Ambiente de desenvolvimento do código fonte: Visual Studio Code. [1]
- Linguagem utilizada: C.
- Ambiente de desenvolvimento da documentação: Overleaf LaTeX. [2]

1.2 Ferramentas utilizadas

Algumas ferramentas foram utilizadas para testar a implementação, como:

- *Valgrind*: ferramentas de análise dinâmica do código.

1.3 Especificações da máquina

As máquinas utilizadas para a realização dos testes possuem as seguintes especificações:

- Intel Core i5 10ª geração com 16GB de RAM.
- Intel Core i7 11ª geração com 16GB de RAM.
- Ryzen 5 5500U com 8Gb de RAM

Todos foram executados no WSL2 com Ubuntu 22.1 no Windows 11, sendo que todos os computadores possuem SSD's.

1.4 Instruções de compilação e execução

Para a compilação do projeto, basta digitar:

Compilando o projeto

```
gcc -Wall *.c -o exe
```

Usou-se para a compilação as seguintes opções:

- *-std=99*: para usar-se o padrão ANSI C 99, conforme exigido.
- *-Wall*: para mostrar todos os possível *warnings* do código.

Para a execução do programa basta digitar:

```
./(nome executável) (método a ser usado) (quantidade de itens no arquivo) (situação da ordem do arquivo) (chave a ser pesquisada) [-P]
```

[1] Visual Studio Code está disponível em https://code.visualstudio.com
[2] Disponível em https://www.overleaf.com/

2 Desenvolvimento

2.1 Acesso Sequencial Indexado

Esse método é o mais simples dentre todos presentes no trabalho prático, é uma técnica que combina o acesso sequencial de dados em blocos contíguos com uso de índices para localizar registros específicos de forma mais rápida. Os dados são organizados em blocos sequenciais, e um índice é criado para mapear a localização dos registros dentro desses própios blocos. Isso permite um acesso direto aos registros desejados, mas requer a atualização dos índices e pode não ser eficiente para operações fora da ordem sequencial.

Algumas melhorias foram feitas, em relação ao código apresentado em sala de aula. Sendo elas:

- Remoção dos comandos de saída de texto de dentro das funções fora da main, visando tornar as funções mais objetivas possível e deixando a interação com o usuário ser realizada integralmente na função main.

- Alteração do tipo de retorno das funções de pesquisa para o tipo bool, para indicar que a pesquisa da chave passada foi exitosa (true) ou não (false).

- Alteração da constante ITENSPAGINA de 4 para 10, para aumentar o número de itens por página, visando diminuir o número de acessos ao arquivo, uma vez que mais itens estão sendo transferidos para a memória principal.

- Implementação da pesquisa binária dentro da possível página em que o item, da chave correspondente, pode estar, visando diminuir o numero de comparações da pesquisa sequencial, que é linear, no caso médio, para logarítmico. Dessa forma, aproveita-se o fato das chaves já estarem ordenadas e diminui o tempo de execução no processo de pesquisa.

- Alteração na quantidade de itens que são lidos a cada acesso, adotando o sistema de paginação. Combinando o aumento na quantidade de itens por página, há uma diminuição no número de acessos ao arquivo, diminuindo o tempo de criação da tabela.

- Adicionamos um if para verificar a quantidade de itens na última página, pois, caso a operação (ftell(arq) / sizeof(TipoRegistro)) % ITENSPAGINA retornasse 0, seria que a divisão foi exata, ou seja, a última página é completa e possui o valor de ITENSPAGINA.

2.2 Árvore Binária de Pesquisa

As árvores binárias de pesquisa é um método de pesquisa muito eficiente, principalmente quando é possível que todos os itens estejam armazenadas em memória principal. Já em memória secundária, as árvores binárias são organizadas da seguinte forma: os nodos da árvore ficam armazenados em disco, e os apontadores da esquerda e da direita armazenam os endereços do disco referente aos filhos desse nodo. Visando diminuir o número de acessos ao disco para que a eficiência desse método seja melhorada, os nodos da árvore podem ser agrupados em páginas, obtendo o melhor caso quando os nodos pais e filhos estão sempre na mesma página.

Neste método, a implementação feita no trabalho foi desenvolvida do início, tendo em vista que não foi apresentado o código do mesmo em sala de aula. Entretanto, o desenvolvimento do código foi simples e rápido, já que a árvore binária de pesquisa não apresenta muitos empecilhos ao programador.

Explicando detalhadamente a implementação, o processo inicia-se pela leitura individual de cada item do arquivo e, por padrão, o item é inserido no final do arquivo. Após essa inserção, há uma função que percorrerá o arquivo atualizando os ponteiros, representados por int, para se adaptarem a inserção do novo item.

Algumas melhorias feitas, em relação ao código apresentado em sala de aula:

- Remoção dos comandos de saída de texto de dentro das funções fora da main, visando tornar as funções mais objetivas possível e deixando a interação com o usuário ser realizada integralmente na função main.

- Alteração do tipo de retorno das funções de pesquisa para o tipo bool, para indicar que a pesquisa da chave passada foi exitosa (true) ou não (false).

2.3 Árvore B

A árvore B consiste em um método de pesquisa e uma estrutura de dados extremamente eficiente e balanceada. Tal método consiste em um conjunto de nos interligados, também chamados de páginas, que suportam uma determinada quantidade de registros, sendo que esta quantidade é definida mediante um número m, o qual é o número mínimo suportado pelas páginas, e possui um limite de 2m itens que aparecem em ordem crescente, conforme os itens, nos quais a pagina possui. Além disso, tais páginas possui descendentes, que são limitados entre m+1 descendentes até 2m+1. Sobre a pesquisa da Árvore B, este procedimento é semelhante ao modo de procura da Árvore Binaria de Pesquisa, no qual se compara com os itens da página raiz para encontrar a chave desejada ou o intervalo em que o item procurado se encaixa.

Algumas melhorias feitas, em relação ao código apresentado em sala de aula:

- Remoção dos comandos de saída de texto de dentro das funções fora da main, visando tornar as funções mais objetivas possível e deixando a interação com o usuário ser realizada integralmente na função main.

- Alteração do tipo de retorno das funções de pesquisa para o tipo bool, para indicar que a pesquisa da chave passada foi exitosa (true) ou não (false).

- Para construir a árvore, estamos lendo o arquivo em um vetor do tipo TipoRegistros

2.4 Árvore B*

Árvore B* foi o último método apresentado em sala de aula, e o mais complexo em questão de implementação devido a sua estrutura. A Árvore B* é uma estrutura de dados em forma de árvore balanceada, ela é semelhante a Árvore B, com nós internos contendo chaves de referência e filhos, e folhas contendo os registros de dados propriamente ditos. No entanto ela apresenta alguns diferenças importantes. Primeiro, os nós internos da árvore B* contêm um número mínimo de chaves, tornando a estrutura mais compacta em comparação com a árvore B. Segundo, a árvore B* permite que mais chaves sejam armazenadas nas folhas, aumentando a capacidade de armazenamento e melhorando o desempenho em consultas que envolvem a recuperação de dados.

Algumas melhorias feitas, em relação ao código apresentado em sala de aula:

- Remoção dos comandos de saída de texto de dentro das funções fora da main, visando tornar as funções mais objetivas possível e deixando a interação com o usuário ser realizada integralmente na função main.

- Alteração do tipo de retorno das funções de pesquisa para o tipo bool, para indicar que a pesquisa da chave passada foi exitosa (true) ou não (false).

2.5 Incluindo fragmento de códigos

O Código 4 apresenta um exemplos do codigo do trabalho.

```
1
2  typedef struct{
3      long chave;
4  } Indice;
5
6  bool pesquisa(Indice *tab, int tam, TipoChave Chave, FILE *arq, TipoRegistro *
       resultado){
7      TipoRegistro pagina[ITENSPAGINA];
8      int i, quantItens;
9      long desloc;
10
```

```
    i = 0;
    // procura pela pagina onde o item pode ser encontrado
    while (i < tam && tab[i].chave <= Chave){
        i++;
        comparacoes();
    }

    // caso a chave desejada seja menor que a primeira chave, o item nao
       existe no arquivo
    if (i == 0) return false;

    else{
        // a ultima pagina pode nao estar completa
           if (i < tam) quantItens = ITENSPAGINA;

        else {
            fseek(arq, 0, SEEK_END);
            quantItens = (ftell(arq) / sizeof(TipoRegistro)) % ITENSPAGINA;

            if(quantItens == 0) quantItens = ITENSPAGINA;
        }
    }

    // le a pagina desejada do arquivo
    desloc = (i - 1) * ITENSPAGINA * sizeof(TipoRegistro);
    fseek (arq, desloc, SEEK_SET);
    fread (&pagina, sizeof(TipoRegistro), quantItens, arq);
    transferencias();

    // pesquisa binaria na pagina lida
    TipoRegistro item;
    if(pesquisaBinaria(pagina, Chave, &item)){
        *resultado = item;
        return true;
    }

    return false;
}
```

Código 1: Trecho do código onde houve melhorias no acesso indexado.

```
void constroiArvore(FILE * arq, FILE *arqAbp){

    TipoRegistro itemLeitura;

    //Le os dados do arquivo original e passa para o arquivo da arvore binaria
        de pesquisa
    transferenciasPreProcessamento();
    while ((fread(&itemLeitura, sizeof(TipoRegistro), 1, arq)) != 0){
        transferenciasPreProcessamento();

        TipoItem itemInserir;
        itemInserir.item = itemLeitura;
        itemInserir.dir = -1;
        itemInserir.esq = -1;

        fseek(arqAbp, 0, SEEK_END);
        fwrite(&itemInserir, sizeof(TipoItem), 1, arqAbp);
        atualizaPonteiros(arqAbp, &itemInserir);
    }
}

```

```c
void atualizaPonteiros(FILE *arq, TipoItem *itemInserir)
{
    // Verificando a quantidade de itens no arquivo
    fseek(arq, 0, SEEK_END);
    long qtdItensArquivo = ftell(arq) / sizeof(TipoItem);
    fseek(arq, 0, SEEK_SET);

    //Nao e necessario atualizar ponteiros quando tem-se apenas um item
    if(qtdItensArquivo == 1) return;

    TipoItem aux;

    //Caminhando o ponteiro do arquivo ate o local onde devera alterar o ponteiro da arvore
    long ponteiro = 1;
    long desloc;

    //procura o ponteiro que precisa ser atualizado com o numero da linha do item que foi inserido
    do{
        //Calcula o deslocamento necessario, a partir do horario_inicio do arquivo, para chegar ao no filho do pai
        desloc = (ponteiro - 1) * sizeof(TipoItem);
        fseek(arq, desloc, SEEK_SET);
        fread(&aux, sizeof(TipoItem), 1, arq);
        transferenciasPreProcessamento();
        //Caminhando o ponteiro pelo arquivo ate encontrar uma "folha" = (-1)
        ponteiro = (itemInserir->item.Chave > aux.item.Chave) ? aux.dir : aux.esq;
        comparacoesPreProcessamento();

    }while(ponteiro != -1);

    //Voltando uma posicao para tratar o "no" correto
    fseek(arq, -(sizeof(TipoItem)), SEEK_CUR);

    //Compara novamente as chaves para a insercao da linha
    comparacoesPreProcessamento();
    if(itemInserir->item.Chave > aux.item.Chave) aux.dir = qtdItensArquivo;
    else aux.esq = qtdItensArquivo;
    fwrite(&aux, sizeof(TipoItem), 1, arq); //insere a linha
    return;
}
```

Código 2: Partes relevantes do codigo da arvore binaria de pesquisa.

```c
    bool pesquisa_arvore_b (TipoRegistro *x, TipoApontador Ap, Resultados *resultados){
    if(Ap == NULL) return false;

    long i = 1;
    resultados->pesquisa.comparacoes += 1;
    while(i < Ap->n && x->Chave > Ap->r[i-1].Chave){
        i++;
        resultados->pesquisa.comparacoes += 1;
    }

    resultados->pesquisa.comparacoes += 1;
    if(x->Chave == Ap->r[i-1].Chave){
        *x = Ap->r[i-1];
        resultados->pesquisa.comparacoes += 1;
        return true;
```

```
    }

    resultados->pesquisa.comparacoes += 1;
    if(x->Chave < Ap->r[i-1].Chave) return pesquisa_arvore_b(x, Ap->p[i-1],
        resultados);

    else return pesquisa_arvore_b(x, Ap->p[i], resultados);
}

void imprime (TipoApontador arvore){
    if(arvore == NULL) return;

    int i = 0;
    while(i <= arvore->n){
        imprime(arvore->p[i]);

        if(i != arvore->n)
            printf("%ld ", arvore->r[i].Chave);

        i++;
    }
}
```

Código 3: Partes relevantes do codigo da arvore b de pesquisa.

```
bool Pesquisa(TipoRegistroEstrela *x, TipoApontadorEstrela *Ap, Resultados *
    resultados)
{
    int i;
    TipoApontadorEstrela Pag = *Ap;

    if(*Ap == NULL) return false;

    if (Pag->Pt == Interna)
    {
        i = 1;

        resultados->pesquisa.comparacoes +=1;
        while (i < Pag->UU.U0.ni && x->Chave > Pag->UU.U0.ri[i - 1]){
            i++;
            resultados->pesquisa.comparacoes +=1;
        }

        resultados->pesquisa.comparacoes +=1;
        if (x->Chave < Pag->UU.U0.ri[i - 1]) return Pesquisa(x, &Pag->UU.U0.pi
            [i - 1], resultados);

        else return Pesquisa(x, &Pag->UU.U0.pi[i], resultados);

    }

    i = 1;

    resultados->pesquisa.comparacoes +=1;
    while (i < Pag->UU.U1.ne && x->Chave > Pag->UU.U1.re[i - 1].Chave){
        i++;
        resultados->pesquisa.comparacoes +=1;
    }
    resultados->pesquisa.comparacoes +=1;
    if (x->Chave == Pag->UU.U1.re[i - 1].Chave){
        *x = Pag->UU.U1.re[i - 1];
        return true;
```

```
36      }
37      else return false;
38  }
39
40  void InsereNaPaginaExterna(TipoApontadorEstrela Ap, TipoRegistroEstrela Reg,
        Resultados *resultados){
41      bool NaoAchouPosicao;
42      int k;
43
44      k = Ap->UU.U1.ne;
45      NaoAchouPosicao = (k > 0);
46
47      while(NaoAchouPosicao){
48          resultados->preProcessamento.comparacoes +=1;
49          if(Reg.Chave > Ap->UU.U1.re[k-1].Chave){
50              NaoAchouPosicao = false;
51              break;
52          }
53          Ap->UU.U1.re[k] = Ap->UU.U1.re[k-1];
54          k--;
55          if(k < 1)
56              NaoAchouPosicao = false;
57      }
58
59      Ap->UU.U1.re[k] = Reg;
60      Ap->UU.U1.ne++;
61  }
```

Código 4: Partes relevantes do codigo da arvore b* de pesquisa.

3 Experimentos e Resultados

Para a realização dos experimentos e obtenção dos resultados, foram utilzadas 10 chaves aleatórias presentes nos arquivos. Dessa forma, para realizar os testes, foi utilizada seguinte diretiva no terminal:

```
./(nome_do_executavel) (metodo_desejado) (quantidade_de_itens_no_arquivo) -a [-P]
```

Assim, o -a"irá substituir o valor da chave que será pesquisada e dará inicio aos testes automatizados. Ressalta-se que o parâmetro -P ainda é opcional.

3.1 Utilizando um arquivo de 100 registros:

Arquivo Ordenado Crescente:

Método	Acesso Indexado	ABP	Árvore B	Árvore B*
TM Pré Processamento	0.000769 s	0.007236 s	0.000974 s	0.001223 s
TM Pesquisa	0.000281 s	0.000460 s	0.000002 s	0.000002 s
TM Total	0.001050 s	0.007697 s	0.000976 s	0.001225 s
Trans Média Pré Processamento	11	505	1	1
Trans Média Pesquisa	1	32	0	0
Trans Média Total	12	537	1	1
Comp Pré Processamento	0	504	1650	1611
Comp Média Pesquisa	9	32	13	11
Comp Total	9	536	1663	1622

Arquivo Ordenado Descrescente:

Método	ABP	Árvore B	Árvore B*
TM Pré Processamento	0.006512 s	0.001523 s	0.001110 s
TM Pesquisa	0.000510 s	0.000002 s	0.000002 s
TM Total	0.007023 s	0.001525 s	0.001111 s
Trans Média Pré Processamento	505	1	1
Trans Média Pesquisa	38	0	0
Trans Média Total	543	1	1
Comp Pré Processamento	504	1382	1122
Comp Média Pesquisa	38	13	11
Comp Total	542	1395	1133

Arquivo Aleatório:

Método	ABP	Árvore B	Árvore B*
TM Pré Processamento	0.006933 s	0.000995 s	0.001091 s
TM Pesquisa	0.000450 s	0.000001 s	0.000002 s
TM Total	0.007384 s	0.000996 s	0.001093 s
Trans Média Pré Process amento	74	1	1
Trans Média Pesquisa	7	0	0
Trans Médi a Total	81	1	1
Comp Pré Proces samento	74	1444	1284
Comp Média Pesquisa	7	12	10
Comp Total	81	1456	1294

3.2 Utilizando um arquivo de 1.000 registros:

Arquivo Ordenado Crescente:

Método	Acesso Indexado	ABP	Árvore B	Árvore B*
TM Pré Processamento	0.001487 s	0.106009 s	0.001642 s	0.002962 s
TM Pesquisa	0.000298 s	0.000810 s	0.000002 s	0.000004 s
TM Total	0.001785 s	0.106818 s	0.001644 s	0.002966 s
Trans Média Pré Processamento	101	50050	1	1
Trans Média Pesquisa	1	388	0	0
Trans Média Total	102	50438	1	1
Comp Pré Processamento	0	50049	26844	24754
Comp Média Pesquisa	44	388	20	18
Comp Total	44	50437	26864	24772

Arquivo Ordenado Descrescente:

Método	ABP	Árvore B	Árvore B*
TM Pré Processamento	0.104345 s	0.001316 s	0.001257 s
TM Pesquisa	0.001058 s	0.000002 s	0.000003 s
TM Total	0.105404 s	0.001318 s	0.001260 s
Trans Média Pré Processamento	50050	1	1
Trans Média Pesquisa	573	0	0
Trans Média Total	50623	1	1
Comp Pré Processamento	50049	20277	15663
Comp Média Pesquisa	573	22	17
Comp Total	50622	20299	15680

Arquivo Aleatório:

Método	ABP	Árvore B	Árvore B*
TM Pré Processamento	0.057500 s	0.001392 s	0.002097 s
TM Pesquisa	0.000465 s	0.000002 s	0.000003 s
TM Total	0.057965 s	0.001394 s	0.002100 s
Trans Média Pré Processamento	1175	1	1
Trans Média Pesquisa	11	0	0
Trans Média Total	1186	1	1
Comp Pré Processamento	1175	21710	18630
Comp Média Pesquisa	11	20	17
Comp Total	1186	21730	18647

3.3 Utilizando um arquivo de 10.000 registros:

Arquivo Ordenado Crescente: 10000 crescente

Método	Acesso Indexado	ABP	Árvore B	Árvore B*
TM Pré Processamento	0.000232 s	0.788930 s	0.003525 s	0.003736 s
TM Pesquisa	0.000004 s	0.000548 s	0.000001 s	0.000001 s
TM Total	0.000236 s	0.789478 s	0.003526 s	0.003737 s
Trans Média Pré Processamento	1001	5000500	1	1
Trans Média Pesquisa	1	3710	0	0
Trans Média Total	1002	5004210	1	1
Comp Pré Processamento	0	5000499	372462	331396
Comp Média Pesquisa	462	3710	29	24
Comp Total	462	5004209	372491	331420

Arquivo Ordenado Descrescente:

Método	ABP	Árvore B	Árvore B*
TM Pré Processamento	0.771310 s	0.006242 s	0.004338 s
TM Pesquisa	0.000539 s	0.000004 s	0.000004 s
TM Total	0.771848 s	0.006245 s	0.004342 s
Trans Média Pré Processamento	5000500	1	1
Trans Média Pesquisa	3699	0	0
Trans Média Total	5004199	1	1
Comp Pré Processamento	5000499	265262	198478
Comp Média Pesquisa	3699	28	22
Comp Total	5004198	265290	198500

Arquivo Aleatório:

Método	ABP	Árvore B	Árvore B*
TM Pré Processamento	0.915479 s	0.005188 s	0.005405 s
TM Pesquisa	0.000872 s	0.000004 s	0.000003 s
TM Total	0.916352 s	0.005192 s	0.005409 s
Trans Média Pré Processamento	16444	1	1
Trans Média Pesquisa	16	0	0
Trans Média Total	16460	1	1
Comp Pré Processamento	16444	291459	241919
Comp Média Pesquisa	16	28	23
Comp Total	16460	291487	241942

3.4 Utilizando um arquivo de 100.000 registros:

Arquivo Ordenado Crescente:

Método	Acesso Indexado	ABP	Árvore B	Árvore B*
TM Pré Processamento	0.002904 s	76.047467 s	0.035649 s	0.038739 s
TM Pesquisa	0.000019 s	0.007291 s	0.000002 s	0.000002 s
TM Total	0.002923 s	76.054758 s	0.035651 s	0.038741 s
Trans Média Pré Processamento	10001	70508270	1	1
Trans Média Pré Processamento	10001	70508270	1	1
Trans Média Pesquisa	1	49743	0	0
Trans Média Total	10002	70558013	1	1
Comp Pré Processamento	0	70508270	4773621	4151872
Comp Média Pesquisa	4786	49743	34	28
Comp Total	4786	70558013	4773655	4151900

Arquivo Ordenado Descrescente:

Método	ABP	Árvore B	Árvore B*
TM Pré Processamento	81.167299 s	0.030376 s	0.027173 s
TM Pesquisa	0.005187 s	0.000003 s	0.000006 s
TM Total	81.172486 s	0.030379 s	0.027180 s
Trans Média Pré Processamento	70508270	1	1
Trans Média Pesquisa	34041	0	0
Trans Média Total	70542311	1	1
Comp Pré Processamento	70508270	3272785	2398487
Comp Média Pesquisa	34041	35	27
Comp Total	70542311	3272820	2398514

Arquivo Aleatório:

Método	ABP	Árvore B	Árvore B*
TM Pré Processamento	9.718731 s	0.126443 s	0.079030 s
TM Pesquisa	0.001166 s	0.000005 s	0.000004 s
TM Total	9.719897 s	0.126447 s	0.079035 s
Trans Média Pré Processamento	209911	1	1
Trans Média Pesquisa	20	0	0
Trans Média Total	209931	1	1
Comp Pré Processamento	209911	3645398	2979939
Comp Média Pesquisa	20	34	26
Comp Total	209931	3645432	2979965

3.5 Utilizando um arquivo de 1.000.000 registros:

Arquivo Ordenado Crescente:

Método	Acesso Indexado	ABP	Árvore B	Árvore B*
TM Pré Processamento	0.037642 s	5950.975419 s	0.385793 s	0.365280 s
TM Pesquisa	0.000150 s	0.082550 s	0.000007 s	0.000004 s
TM Total	0.037792 s	5951.057969 s	0.385800 s	0.365284 s
Trans Média Pré Processamento	100001	178429366	1	1
Trans Média Pesquisa	1	647886	0	0
Trans Média Total	100002	179077252	1	1
Comp Pré Processamento	0	178429366	58178793	49941311
Comp Média Pesquisa	50314	647886	45	35
Comp Total	50314	179077252	58178838	49941346

Arquivo Ordenado Descrescente:

Método	ABP	Árvore B	Árvore B*
TM Pré Processamento	7297.271855 s	0.315965 s	0.251982 s
TM Pesquisa	0.067900 s	0.000003 s	0.000004 s
TM Total	7297.339755 s	0.315968 s	0.251985 s
Trans Média Pré Processamento	17842 9366	1	1
Trans Média Pesquisa	471531	0	0
Trans Média Total	178900897	1	1
Comp Pré Processamento	178429366	38905591	28103683
Comp Média Pesquisa	471531	40	33
Comp Total	178900897	38905631	28103716

Arquivo Aleatório:

Método	ABP	Árvore B	Árvore B*
TM Pré Processamento	110.011771 s	1.358603 s	1.078745 s
TM Pesquisa	0.001540 s	0.000006 s	0.000007 s
TM Total	110.013311 s	1.358609 s	1.078752 s
Trans Média Pré Processamento	2561478	1	1
Trans Média Pesquisa	26	0	0
Trans Média Total	2561504	1	1
Comp Pré Processamento	2561477	44030122	35289222
Comp Média Pesquisa	26	40	34
Comp Total	2561503	44030162	35289256

4 Análise dos testes

4.1 Acesso Sequencial Indexado

É notório que, antes da análise dos dados obtidos através da utilização deste método de pesquisa, é válido ressaltar que tal método foi melhorado devido à implementação de uma pesquisa binária e não uma pesquisa sequencial, na qual é comumente usada neste procedimento.

Em relação às informações obtidas mediante a execução de testes em arquivos com diferentes tamanhos de registros, nota-se que ha uma abundância de transferências no pre-processamento e uma pequena quantidade de transferências na pesquisa, já que os registros para as pesquisas são transferidos para a memória principal. Com relação ao número de comparações media, observa-se que, este numero aumenta a cada tamanho de arquivo, e também pelo fato dos registros de teste selecionados aleatoriamente.

Diante dos fatos, pode-se concluir que o acesso sequencial indexado, apesar do arquivo ter que estar ordenado para se obter uma pesquisa, tal método mostra-se eficiente, principalmente quando é implementado melhorias em tal procedimento, como a pesquisa binaria.

4.2 Árvore Binária de Pesquisa (ABP)

Antes de analisar os resultados da Árvore Binaria de Pesquisa, é necessário estar ciente que os dados de pré-processamento, ou seja, criação do aqrquivo com a ávore devidamente gerada, possui valores que estão, de certa forma, com um certo vies.

Em outras palavras, no funcionamento da ABP, a primeira coisa que ocorre é a tentativa de recuperar uma árvore já criada para evitar a sua recriação. Dessa forma, caso o seu arquivo não exista, ele será criado, o que ocasionará em um tempo final de pré-processamento muito alto. Assim, para um teste unitário, o valor será mostrado explicitamente. Todavia em um teste automatizado, o valor sofrerá com algumas alterações, uma vez que, para a primeira chave, a árovre não estará montada, mas para as próximas sim. Logo, o tempo de pré processamento dos itens seguintes serão menores, pois todos os dados já estão montados, sendo nececssário somente a pesquisa direta no arquivo.

Com um arquivo de 100 registros, percebe-se a semelhança entre o número de transações e comparações tanto de forma crescente quanto descrescente, pois como ambos não são bons casos para a árvore, os valores fazem sentido estarem próximos. Já em relação ao tempo, esta é uma medida mais complexa de ser analisada, pois como todos ficaram com diferenças nos milésimos de segundo, a sua análise pode sofrer muita interferência da situação atual do computador, como o número de tarefas sendo executadas ao mesmo tempo e outras razões externas.

Para o arquivo de 1000 e de 10000 itens, os valores da crescente e derescente seguem o mesmo padrão do arquivo de 100 registros. Porém para os itens aleatórios, podemos perceber que o número de comparações diminuem, pois não estamos mais com os piores casos de inserção e pesquisa de uma árvore binária, fazendo com que todos os valores sejam diminuidos.

Indo para o arquivo de 100 mil itens, podemos começar a ver diferenças extremamente significativas. Ainda que os dados crescentes e decrescentes sejam diferentes, podemos veridicar uma semelhança nas transferências e comparações, o tempo possui uma diferença de cerca de 5 segundos de um para outro (76 segundos crescente e 81 decrescente). Todavia, quando analisamos os itens que estão dispostos de forma aleatória, vemos o poder da ABP, pois a tempo caiu para cerca de 9.72 segundos e de 70 milhões de transferências e comparações para cerca de 200 mil.

Por fim, para 1 milhão de itens, percebemos o mesmo comportamento que 100 mil itens, ou seja, uma queda abrupta da ordenação crescente e descrescente para a aleatória, saindo de 170 milhões de comparações e transferências para cerca de 2,5 milhões.

4.3 Árvore B

A Árvore B quando está trabalhando com arquivos pequenos, não é possível notar uma diferença considerável em nenhum dos parâmetros de análise em relação a tempo. Todos os resultados obtidos seja com arquivo ordenado crescente, descrescente ou aleatório, apresentam valores extremamente próximos. Por exemplo o tempo médio total no arquivo de 1.000 registros

ordenado crescente (0.001644 s), usando o arquivo ordenado decrescente (0.001318 s), e por fim com o arquivo aleatório o tempo médio obtido em nossos teste foi de (0.001394 s).

Porém quando se trata de arquivos relativamente grandes como por exemplo um que contenha 1.000.000 registros, nesta estrutura de dados, os arquivos crescente e decrescente apresentam valores muito próximos, pois visto que, o tempo médio total no arquivo aleatório foi consideravelmente maior. Em nossos testes os números obtidos confirmam essa superioridade de arquivos ordenados. No ordenado crescente de 1.000.000 registros o tempo médio total foi de (0.385800 s), e no ordenado decrescente (0.315968 s), valores que apresentam um certo grau de proximidade. Por fim o resultado obtido no arquivo aleatório foi de (1.358609 s).

Analisando o parâmetro de comparações médio total, é notório que nesta estrutura de dados os arquivos ordenados crescentemente o número de comparações é significavelmente maior. Os dados obtidos comprovam esse fato. Com o arquivo de 100 registros crescente foram obtidos no total (1663) comparações, no arquivo ordenado decrescentemente de mesmo tamanho (1395) e por fim o arquivo aleatório (1456). E no nosso teste com o arquivo com o maior número de registros foram obtidos os seguintes dados: crescente (58178838) comparações totais, decrescente (38905631) e por último o arquivo aleatório (44030162). Dos variados tamanhos de arquivos testados, em todos se repetiu o mesmo padrão.

Com os dados obtidos em nossos teste, concluímos que o método de pesquisa "Árvore B", em relação a comparações o arquivo decrescente leva vantagem, pois em todos os testes ele teve menos comparações, e o pior caso observado foram observados quando os dados estavam ordenados crescentemente, e análago a isso vimos que o arquivo aleatório através desses parâmetros é o caso médio.

4.4 Árvore B*

Os testes realizados com o método, árvore b*, revelaram alguns comportamentos com relação aos tempos de processamento, quantidade de transferências, e quantidade de comparações em todos os tamanhos de arquivos (100,1000,10000,100000,1000000). Os Tempos de pré-processamento aumentaram de uma maneira mais acentuada nos arquivos de 100000 e de 1000000 de dados, entretanto, o tempo de pesquisa se manteve praticamente instantâneo em todos os tamanhos e tipos de arquivo.

Com relação a quantidade de transferências, nas simulações houve apenas uma transferência, pois todos os arquivos eram carregados em memória principal em primeira análise.

Com relação a quantidade de comparações realizadas, assim como no tempo de pré-processamento, quase todas as comparações ocorrem no processo de criação da árvore, e conforme o tamanho do arquivo aumenta, o números de comparação também aumenta, Porém o número de comparações no momento de pesquisa de alguma chave se manteve consistentemente baixo independente do tipo e do tamanho do arquivo.

Dadas estas informações, concluímos que o método de pesquisa "árvore b*" ´ é extremamente eficiente na pesquisa em arquivos de qualquer tamanho. Entretanto deverá ser implementado somente quando o arquivo em questão tiver um grande volume de dados, pois sua implementação é mais complexa em comparação aos outros métodos e não valeria a pena implementá-lo para um arquivo com um baixo volume de dados que outros métodos de implementação mais simples, conseguiriam apresentar um bom resultado.

A respeito das melhorias no codigo, o codigo da arvore b* e uma adaptação do codigo da árvore b, a pesquisa da árvore b* foi alterada para procurar elementos apenas nos nós folha, e a inserção foi alterada para identificar quando temos um nó interno e um nó externo para realizar os processos de divisão corretamente, obedecendo as propriedades da árvore b* de permanecer com todos os dados nos nós externos.

5 Considerações Finais

Após a finalização de todo o trabalho, podemos notar que as maiores dificuldades que tivemos foi em relação ao desenvolvimento da Árvore B*, uma vez que a mesma não possui um código pré-implementado, assim, ficamos muito tempo tentando entender como ela funcionaria a partir da adaptação do código da Árvore B. Ademais, também tivemos problemas em sua implementação no momento em que iríamos aumentar o tamanho da árvore interna, por causa de erros de código.

Depois, para realizar os testes, nos deparamos com a demora de execução dos métodos, principalmente a árvore de pesquisa binária para os arquivos com 100 mil e 1 milhão de registros, pois era necessário lê-los e criar um novo com os valores de ponteiros corretos. Ademais, outro problema enfretado foi com o computador de um dos membros que estava, originalmente realizando os testes, pois o seu SSD não possuia espaço para gerar todos os arquivos necessários, assim tivemos que dividir as máquinas que executaram os testes.

Por fim, após toda a difucldade de executar os testes, conseguimos ver e entender bem a implementação de todos os métodos. Sendo que vale a pena ressaltar o fato de que foi possível visualizar a árvore B* da mesma forma que vemos em testes de mesa, pois para tentar solucionar os problemas com a sua criação, fizemos um método para exibir todos os valores, presentes nela durante a sua criação. Dessa forma, foi possivel solidificar o nosso entendimento sobre a mesma.