Universidade Federal de Ouro Preto

Departamento de Computação

Augusto Guilarducci (20.1.4012)

Caio Monteiro (20.1.4110)

Paulo Correa (20.1.4036)

Pedro Lucas Damasceno (20.1.4003)

Trabalho Prático I – Pesquisa Externa

Implementação e análise experimental de métodos de pesquisa em memória secundária

Ouro Preto, Minas Gerais

2021

SUMÁRIO

- # Introdução

Este relatório se refere ao primeiro trabalho prático da disciplina Estrutura de Dados 2, realizado durante o período 21.1, cujo objetivo consiste na implementação e estudo da complexidade de desempenho dos seguintes métodos de pesquisa externa apresentados em sala de aula:

1. Acesso Sequencial Indexado
2. Árvore Binária Externa
3. Árvore B Externa
4. Árvore B* Externa

Dividimos tanto a implementação quanto a etapa de análise, sendo cada membro responsável por um método. A princípio, desenvolvemos a modularização e o fluxo de execução para organizar e possibilitar que cada membro tivesse autonomia para desenvolver separadamente a sua parte no seu tempo. Em seguida, implementamos a criação dos arquivos necessários para testagem, funções de uso geral (verificação de parâmetros de execução, impressão de registros) e outras diretivas do enunciado do trabalho. Ademais, utilizamos também um sistema de controle de versões cujo repositório pode ser visualizado no link https://github.com/augustofgui/grupo-21.1 .

Para a análise experimental dos métodos, utilizamos a função *clock* da biblioteca *time.h* para realizar o cálculo de tempo de execução e variáveis inteiras para somar as comparações e leituras. Tanto as etapas de criação dos índices quanto a própria pesquisa foram contabilizadas, e as chaves pesquisadas foram determinadas por '*1 + rand () % (nro_registros – 1)*'. Os outputs utilizados para a construção dos gráficos estão localizados no diretório "análise experimental" do repositório.

- **Acesso Sequencial Indexado**

1. Definição de itens por página

Com a intenção de otimizar o método em geral e ilustrar um contexto mais próximo do real, optamos por dinamizar o tamanho das páginas de acordo com o tamanho da entrada. Como o número de registros está bem definido no enunciado do trabalho, não nos preocupamos com grandezas diferentes, haja vista que a verificação de parâmetros ao início da execução impede que esses casos vão adiante. No caso de um arquivo de 1.000.000 registros, o consumo máximo de memória do programa será de aproximadamente 2.9MB(vetor de 500 registros), um valor bastante razoável se comparado à dimensão do arquivo de entrada.

```
switch (n_Registros)
    {
    case 100:
        return 5;
    case 1000:
        return 50;
    case 10000:
        return 100;
    case 100000:
        return 200;
    case 1000000:
        return 500;
    }
```

2. Definição da tabela e preenchimento

O código de exemplo fornecido lê cada registro e utiliza uma variável auxiliar para realizar um somatório e selecionar as chaves a serem adicionadas ao índice da tabela. Alteramos essa função de forma que apenas o inteiro referente ao índice daquela página seja lido no arquivo.

```
int aux;

    for (int i = 0; fread(&aux, sizeof(int), 1, arquivo_Binario);
i++)
    {
        tabela_Indice[i] = aux;
        fseek(arquivo_Binario, (itens_Pagina * sizeof(Registro)) -
sizeof(int), SEEK_CUR);
    }
rewind(arquivo_Binario);
```

Após a leitura do inteiro, o ponteiro, na posição seguinte, é deslocado na quantidade de 1 página – 1 inteiro, posicionando-o sobre a primeira chave da página seguinte. Ao final do processo, o ponteiro é reiniciado na posição inicial do arquivo.

3. Busca pelo índice

Iniciamos tratando casos onde o registro obviamente não se encontra no arquivo. Em um arquivo crescente, por exemplo, se o registro for menor que a primeira chave, então a busca pode ser interrompida imediatamente. Como o enunciado também inclui a possibilidade de arquivos decrescentes, utilizamos a mesma lógica para excluir esse tipo de caso.

```
switch (n_Situacao)
    {
    case 1:
        if (tabela_Indice[0] > n_Chave)
            return -1;
        for (aux = 0; aux < tam_Tabela; aux++)
        {
            if (tabela_Indice[aux] == n_Chave)
                return aux;
            if (tabela_Indice[aux] > n_Chave)
                return aux - 1;
        }
        return aux - 1;
        break;
    case 2:
        if (tabela_Indice[0] < n_Chave)
            return -1;
        for (aux = 0; aux < tam_Tabela; aux++)
        {
            if (tabela_Indice[aux] == n_Chave)
                return aux;
            if (tabela_Indice[aux] < n_Chave)
                return aux - 1;
        }
        return aux - 1;
        break;
    }
```

Em seguida, percorremos a tabela sequencialmente comparando os índices com a chave pesquisada. No caso de um arquivo decrescente, se a chave for maior do que o índice atualmente comparado, então ela só pode estar localizada na página anterior. Se a iteração for concluída, retornamos o índice da última página independentemente, pois se a chave existir no arquivo, então ela só pode pertencer àquela página.

4. Leitura da página e busca em memória principal

Uma vez encontrado o índice da página, calculamos o deslocamento, posicionamos o ponteiro do arquivo de acordo e realizamos a leitura dos dados. Em memória principal, outra pesquisa sequencial é realizada e a iteração é interrompida se a chave pesquisada for encontrada. Note que não há preocupação quanto à possibilidade de página incompleta, pois a dinamização do tamanho da página impede que isso aconteça para as possibilidades de entradas enunciadas no trabalho. Todavia, se essa possibilidade fosse real, bastaria utilizar a função sizeof para enumerar a quantidade de elementos do vetor e dividi-la pelo tamanho referente ao seu tipo (int, nesse caso). Linguagens de mais alto nível possuem ferramentas ainda melhores para tratar casos dessa natureza (em PHP, por exemplo, poderíamos utilizar a função foreach para iterar o vetor sem a preocupação de acessar uma posição inexistente, ou a função count para obter mais objetivamente a grandeza do vetor).

```
    Registro aux[itens_Pagina];
    long int desloc_Ponteiro = itens_Pagina * indice_Pagina *
sizeof(Registro);
    fseek(arquivo_Binario, desloc_Ponteiro, SEEK_SET);
    fread(aux, sizeof(Registro), itens_Pagina, arquivo_Binario);

    for (int i = 0; i < itens_Pagina; i++)
    {
        if (aux[i].chave == n_Chave)
        {
            printf("Registro encontrado!\n");
            imprimir_Registro(aux[i]);
            return 1;
        }
```

5. Análise experimental

Observamos que o número de comparações é variável, haja vista que depende do quão distante o índice da página e a própria chave estão em relação ao início dos vetores que as armazenam. Já o número de transferências é constante para o mesmo número de registros, pois a obtenção da tabela de índices efetua (*nro_registros / tam_tabela*) transferências de inteiros (chaves) e a pesquisa sequencial requer a transferência de uma página da grandeza de (*itens_pagina*).

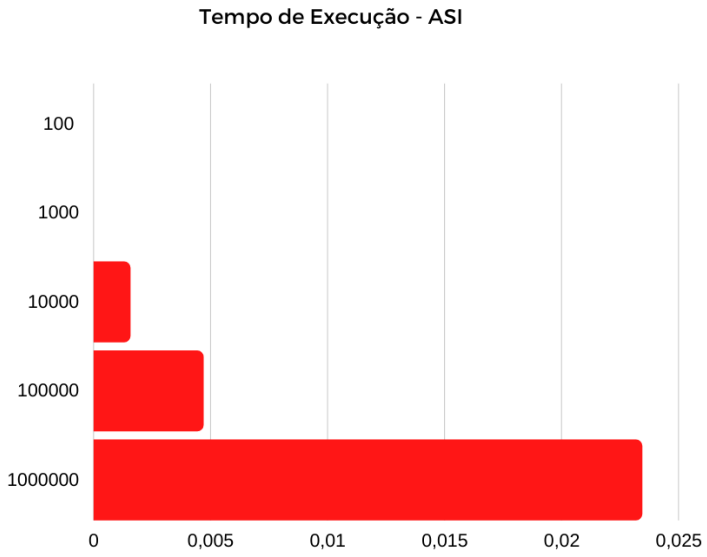

Figura 1 - Número de registros e tempo de execução (em milissegundos)

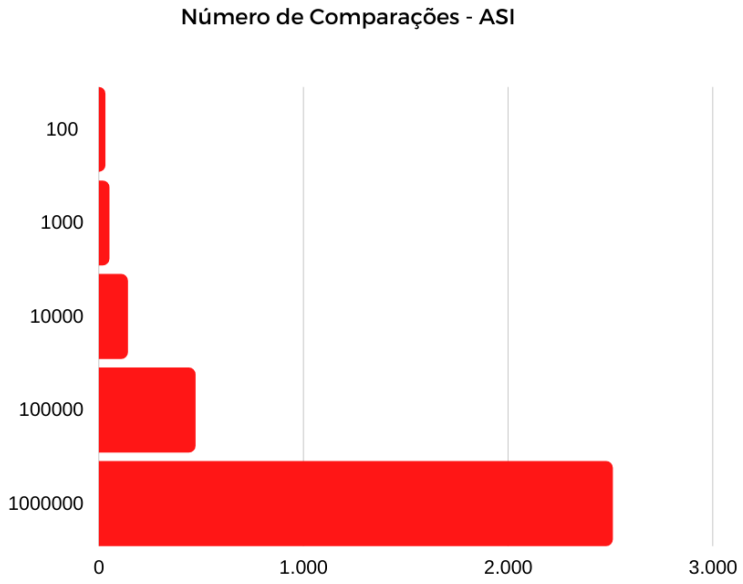

Figura 2 - Número de registros e número de comparações realizadas

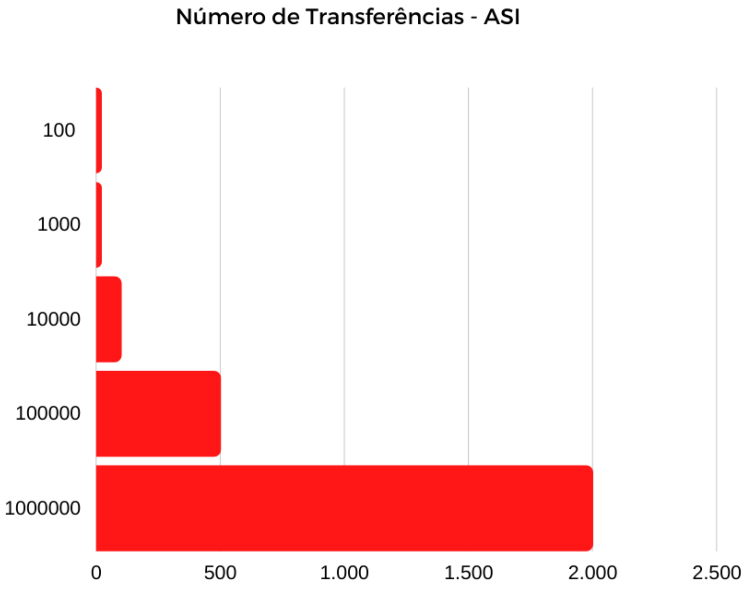

Figura 3 - Número de registros e número de transferências realizadas

- # Árvore Binária Externa

1. Pesquisa em uma Árvore Binária

A realização da pesquisa na árvore binária se dá de uma forma recursiva onde primeiramente checa se o nó atual aponta nulo, representando o fim de um caminho, retornando zero caso seja indicando que o item não foi encontrado, depois checa se o item a ser procurado se encontra no nó atual retornando 1 caso sim, e por último verifica se o item procurado é menor ou maior do que o item do nó atual para realizar o mesmo processo no nó da esquerda ou da direita respectivamente. As linhas de código com "arvore_binaria_comparacoes++;" são contadores da quantidade de comparações feitas no programa.

```
int Buscar (TNo *pRaiz, Registro *x){
    arvore_binaria_comparacoes++;
    //If que checa de a árvore chegou no fim sem encontrar o
valor
      if (pRaiz == NULL)
            return 0;
    arvore_binaria_comparacoes++;
    //If que checa se o atual valor no nó é o valor procurado
      if (x->chave == pRaiz->item){
        x->chave=pRaiz->item;
            return 1;
    }
    arvore_binaria_comparacoes++;
    //If que checa qual o próximo nó de acordo com o valor
      if (x->chave < pRaiz->item)
            return Buscar(pRaiz->pEsq, x);
      else
            return Buscar(pRaiz->pDir, x);
}
```

2. Caminhamento em uma Árvore Binária

Realizamos o caminhamento de maneira recursiva, passando o apontador inicial da árvore, caso o apontador seja nulo, nada será impresso. O caminhamento é passando por todos os nós a esquerda da raiz, pela raiz e por todos os nós a direita da raiz. As linhas de código com "arvore_binaria_comparacoes++;" são contadores da quantidade de comparações feitas no programa.

```
void Infixa(TNo *p){
    arvore_binaria_comparacoes++;
    if (p == NULL )
        return ;
    Infixa(p->pEsq);
    printf(" %d",p->item);
    Infixa(p->pDir);
}
```

3. Inserção em Árvore Binária

Para o item ser inserido, primeiro checamos se a árvore está vazia ou não para poder criar o nó raiz. Depois que a árvore tiver seu primeiro nó é criado uma variável auxiliar que aponta para a arvore original. Nessa variável auxiliar é feita uma pesquisa dentro de um while para determinar em qual nó o item x a ser inserido deve ficar. Dentro do while são feitas duas comparações para determinar se x pertence a esquerda ou à direita do nó atual. Caso o x já esteja na árvore a função retornará 0. Caso a auxiliar aponte nulo, á posição do novo nó terá sido encontrada. Á função TNo_Cria é responsável por criar um novo nó com o item desejado.

```
int TArvore_Insere (TNo **pRaiz , int x){
    arvore_binaria_comparacoes++;
    //If que checa se árvore está vazia ou não para colocar o primeiro nó
    if (*pRaiz==NULL) {
        *pRaiz = TNo_Cria(x);
        return 1;
    }
    TNo **pAux;
    pAux = pRaiz;
    //While para que pAux aponte para o lugar onde o novo nó deve ficar
    while (*pAux!=NULL) {
        arvore_binaria_comparacoes++;
        if (x<(*pAux)->item)
            pAux = &((*pAux)->pEsq);
        else {
            arvore_binaria_comparacoes++;
            if (x>(*pAux)->item)
                pAux = &((*pAux)->pDir);
            else return 0;
        }
    }
    *pAux = TNo_Cria(x);
    return 1;
}
//Função auxiliar para criar um novo nó quando necessário
TNo *TNo_Cria (int x){
    TNo *pAux = (TNo*)malloc(sizeof(TNo));
    pAux->item = x;
    pAux->pEsq = NULL;
    pAux->pDir = NULL;
    return pAux;
}
```

4. Execução da Árvore Binária

Iniciamos a árvore, fazemos um loop terminado no fim do arquivo binário, que conta a quantidade de transferências realizadas e insere os registros na árvore.

```
void TArvore_Inicia (TNo
**pRaiz){
    *pRaiz = NULL ;
}
```

```
TNo* arvore;
    TArvore_Inicia(&arvore);

    Registro *reg = (Registro*) malloc
(sizeof(Registro));
    //While responsável por colocar os valores
do arquivo binário na árvore
    while (!feof(arquivo_binario)) {
        arvore_binaria_transferencias++;
        fread(reg, sizeof(Registro), 1,
arquivo_binario);
        TArvore_Insere(&arvore, reg->chave);
    }
```

Depois, nós realizamos a pesquisa na árvore e criamos uma condição para quando o registro for encontrado, o usuário será avisado imprimindo uma mensagem, além da quantidade de transferências e comparações que foram realizadas durante todo o método de execução. Caso a condição não for ativada, ou seja, o registro não ser encontrado, é retornado o valor de 0.

```
Registro *pesquisa = (Registro*) malloc (sizeof(Registro));
    pesquisa->chave = nro_chave;

    int i = 0;
    arvore_binaria_comparacoes++;
    //If que checa se um valor foi encontrado
    if (Buscar(arvore, pesquisa)) {
        printf("Registro encontrado!\n");
        printf("N° de transferências: %d\n",
arvore_binaria_transferencias);
        printf("N° de comparações: %d\n",
arvore_binaria_comparacoes);
        if (print_registro)
            imprimir_registro(*pesquisa);
        return 1;
    }
    return 0;
```

5. Análise Experimental

Os testes foram feitos alternando o registro e os arquivos (ordenado crescente, ordenado decrescente, desordenado aleatoriamente). Infelizmente, não foi possível realizar os testes de 1.000.000 de registros nem os testes de 100.000 para os casos ordenados crescentes e decrescentes, pois o computador não rodou em nenhuma situação.

a. Tempo de Execução

O tempo de execução aumenta consideravelmente, de forma proporcional à quantidade de registros dentro de um arquivo e, além disso, por se tratar de uma Árvore Binária, ela pode ficar desbalanceada dependendo da forma que os registros são inseridos nela.

Como na forma ordenada tanto crescente quanto decrescente o valor de cada registro é constantemente maior ou menor então a árvore ficará desbalanceada pois só terá valores a direita no caso da crescente ou valores a esquerda no caso da decrescente, logo o tempo de pesquisa será maior.

Vale adicionar que o tempo de pesquisa da decrescente é levemente menor do que a da crescente, pois como nesse caso a árvore é composta apenas de nós a esquerda, e a função começa a varredura pela esquerdam, o item desejado é encontrado mais rápido.

Já na forma desordenada aleatoriamente, tem uma chance razoável de que a raiz seja um número próximo da média de todos os valores, logo a árvore ficará mais balanceada e os tempos de pesquisa serão menores.

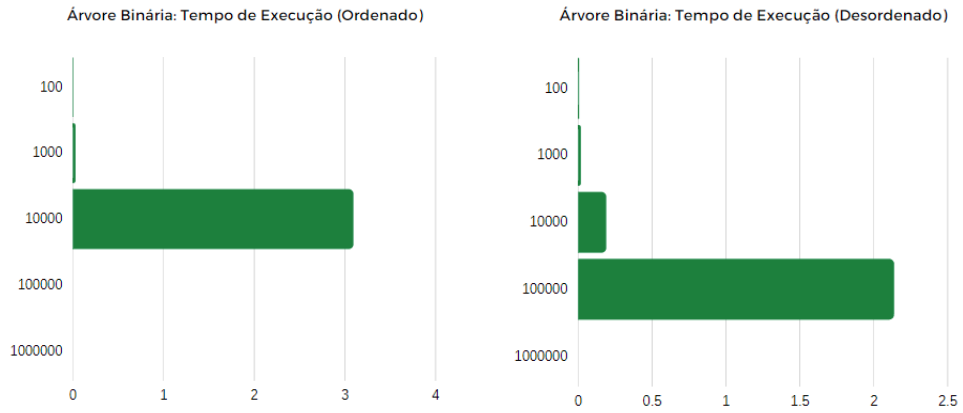

Figura 1 e 2 - Número de registros e tempo de execução (em segundos)

b. Quantidade de Comparações

Assim como o tempo de execução, a quantidade de comparações aumenta consideravelmente, de forma proporcional à quantidade de registros dentro de um arquivo.

Nos casos onde os registros são inseridos de forma ordenada, pode se observar que o número médio de comparações da forma crescente (100036501 para 10000 registros) é aproximadamente o dobro do número médio de comparações da forma decrescente (50028505 para 10000 registros). Isso ocorre, pois, a varredura pela árvore sempre começa pela esquerda e a árvore resultante da forma decrescente é uma árvore com nós sempre a esquerda.

Assim como o tempo de execução a quantidade de comparações de uma árvore resultante da forma desordenada será menor em geral.

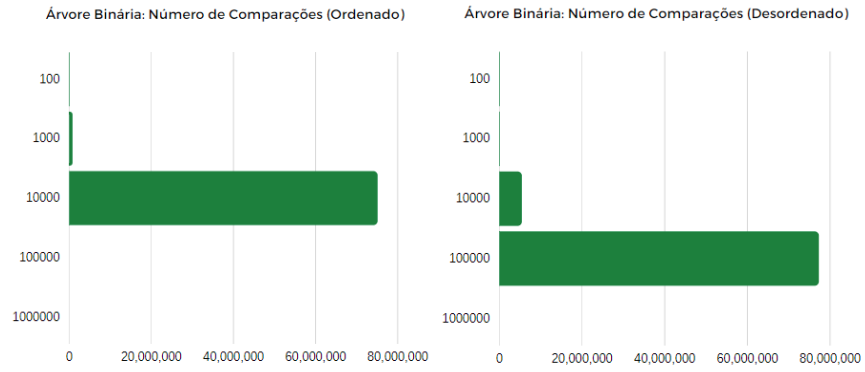

Figura 3 e 4 - Número de registros e número de comparações realizadas

c. Quantidade de Transferências

O número de transferências é sempre constante, independentemente da maneira do teste, ela sempre será igual a quantidade de registros na Árvore Binária.

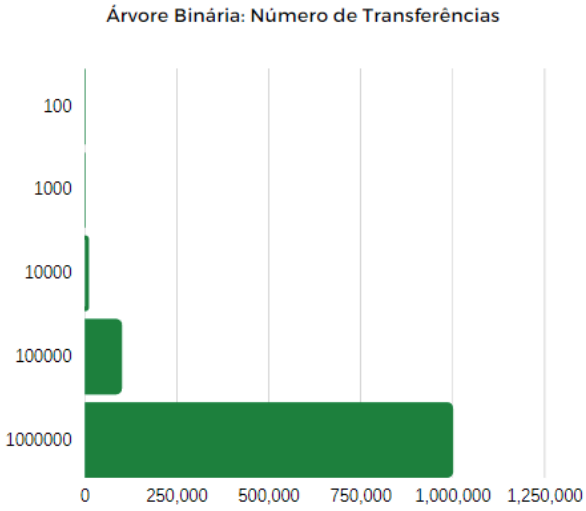

Figura 5 - Número de registros e número de transferências realizadas

- # Árvore B

1. Pesquisa na Árvore B

Para realizar a pesquisa na Árvore B, primeiro nós comparamos a chave com as outras chaves que estão na página raiz até que a chave seja encontrada ou o intervalo no qual ela se encaixa. Caso a chave não fosse localizada, o apontador iria para a subárvore do intervalo encontrado. Em seguida, o processo era repetido recursivamente até achar a chave ou atingir uma página folha (apontador nulo).

```
int Pesquisa (Registro *x, Apontador Ap) {
    long i = 1;
    ab_comparacoes++;
    if (Ap == NULL){
        return 0;
    }
    while (i < Ap->n && x->chave > Ap->r[i-1].chave) i++;
    ab_comparacoes++;
    if (x->chave == Ap->r[i-1].chave){
        *x = Ap->r[i-1];
        return 1;
    }
    ab_comparacoes++;
    if (x->chave < Ap->r[i-1].chave) {
        return Pesquisa (x, Ap->p[i-1]);
    }
    else {
        return Pesquisa (x, Ap->p[i]);
    }
    return 0;
}
```

2. Caminhamento em Árvore B

Realizamos o caminhamento de maneira recursiva, passando o apontador inicial da árvore, caso o apontador seja nulo, nada será impresso. Assim o loop será acionado, e enquanto a condição do while for aceita, cada item da primeira página vai ser selecionado, e em seguida, imprimindo seus filhos à esquerda, então todo processo é por ele (caminhamento à esquerda). Se o valor de i for diferente da quantidade de itens, então será impressa a chave daquela posição.

```
void Imprime(Apontador arvore) {
    int i = 0;
    ab_comparacoes++;
    if (arvore == NULL)
        return;
    while (i <= arvore->n) {
        Imprime(arvore->p[i]);
        ab_comparacoes++;
        if (i != arvore->n)
            printf("'%d' \n",
arvore->r[i].chave);
        i++;
    }
    printf("\n");
}
```

3. Inserção em Árvore B

Para o item ser inserido, primeiro nós localizamos a página onde ele deve ser inserido, caso encontre uma página válida (menos de **2m** itens), ele é inserido nela. Caso encontre uma página que esteja cheia, outra página é criada para a divisão de itens, caso a página pai também esteja cheia, a divisão se propaga.

```
// A página tem espaço.
    ab_comparacoes++;
    if (Ap->n < MM){
        InsereNaPagina (Ap, *RegRetorno, *ApRetorno);
        *Cresceu = 0;
        return;
    }
    // A página tem que ser dividida (Overflow)
    ApTemp = (Apontador)malloc(sizeof(Pagina));
    ApTemp->n = 0;
    ApTemp->p[0] = NULL;
    ab_comparacoes++;
    if (i < M+1){
        InsereNaPagina(ApTemp, Ap->r[MM-1], Ap->p[MM]);
        Ap->n --;
        InsereNaPagina(Ap, *RegRetorno, *ApRetorno);
    } else
        InsereNaPagina(ApTemp, *RegRetorno, *ApRetorno);

    for (j = M + 2; j <= MM; j++) {
        InsereNaPagina(ApTemp, Ap->r[j-1], Ap->p[j]);
    }

    Ap->n = M;
    ApTemp->p[0] = Ap->p[M+1];
    *RegRetorno = Ap->r[M];
    *ApRetorno = ApTemp;
```

Depois, no pior caso, o processo de divisão pode propagar-se até a raiz da árvore B e, assim, sua altura aumenta. (Única forma de aumentar a altura de uma árvore B: divisão da raiz).

```
void Ins (Registro Reg, Apontador Ap,
short *Cresceu, Registro *RegRetorno,
Apontador *ApRetorno){
    long i = 1; long j;
    Apontador ApTemp;
    ab_comparacoes++;
    if (Ap == NULL) {
        *Cresceu = 1;
        (*RegRetorno) = Reg;
        (*ApRetorno) = NULL;
        return;
    }
```

```
Ins(Reg, *Ap, &Cresceu, &RegRetorno, &ApRetorno);
    if (Cresceu == 1) { // Caso a árvore tenha crescido
        ApTemp = (TipoPagina *) malloc (sizeof(TipoPagina));
        ApTemp->n = 1;
        ApTemp->r[0] = RegRetorno;
        ApTemp->p[1] = ApRetorno;
        ApTemp->p[0] = *Ap;
        *Ap = ApTemp;
    }
```

4. Execução da Árvore B

Primeiramente, fizemos um loop - que se encerra ao final do arquivo criado -, que tem a função de contar a quantidade de transferências realizadas, além de inserir os registros na árvore.

```
    while
(!feof(arquivo_binario))
{

ab_transferencias++;
        fread(reg,
sizeof(Registro), 1,
arquivo_binario);
        Insere(*reg,
&arv);
    }
```

Depois, nós fazemos a pesquisa na Árvore B e criamos uma condição para quando o registro for encontrado, o usuário será avisado imprimindo uma mensagem, além da quantidade de transferências e comparações que foram realizadas durante todo o método de execução. Caso a condição não for ativada, ou seja, o registro não ser encontrado, é retornado o valor de 0.

```
ab_comparacoes++;
    if (Pesquisa(pesquisa, arv)) {
        printf("Registro encontrado!\n");
        printf("N° de transferências: %d\n",
ab_transferencias);
        printf("N° de comparações: %d\n", ab_comparacoes);
        if (print_registro)
            imprimir_registro(*pesquisa);
        return 1;
    }

    return 0;
```

5. Análise experimental

Os testes foram feitos alternando o registro e os arquivos (ordenado crescente, ordenado decrescente, desordenado aleatoriamente). Infelizmente, não foi possível realizar os testes de 1.000.000 de registros, pois o computador não rodou em nenhuma situação.

a. Tempo de execução

O tempo de execução aumenta consideravelmente, de forma proporcional à quantidade de registros dentro de um arquivo, entretanto, por se tratar de uma Árvore B, ela é sempre balanceada. Então, independentemente da maneira em que os registros estiverem ordenados, o tempo praticamente não se altera, considerando a mesma quantidade de registros nos 3 casos (ordenado crescente, ordenado decrescente, desordenado aleatoriamente).

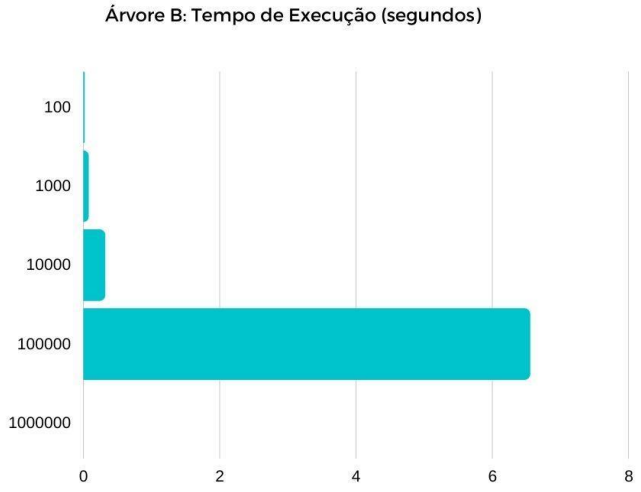

Figura 4 - Número de registros e tempo de execução (em segundos)

b. Quantidade de Comparações

Assim como o tempo de execução, a quantidade de comparações aumenta consideravelmente, de forma proporcional à quantidade de registros dentro de um arquivo,

entretanto, por se tratar de uma Árvore B, ela é sempre balanceada. Então, independentemente da maneira em que os registros estiverem ordenados, o número de comparações praticamente não se altera, considerando a mesma quantidade de registros nos 3 casos (ordenado crescente, ordenado decrescente, desordenado aleatoriamente).

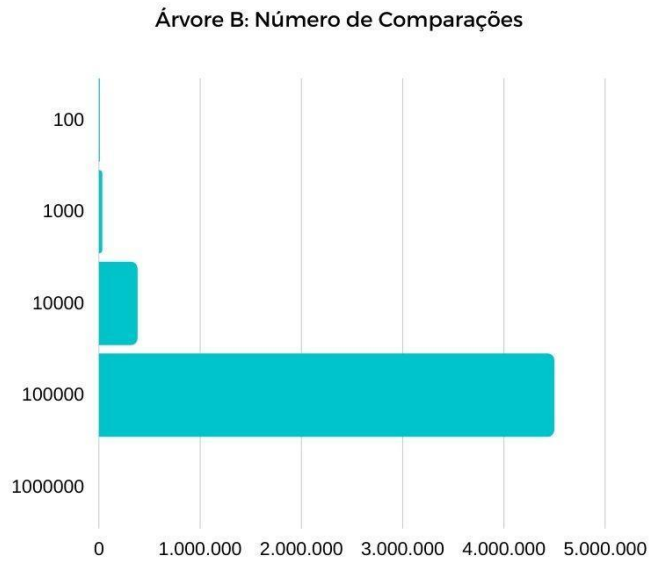

Figura 5 - Número de registros e comparações realizadas

c. Quantidade de Transferências

O número de transferências é sempre constante, independentemente da maneira do teste, ela sempre será igual a quantidade de registros na Árvore B.

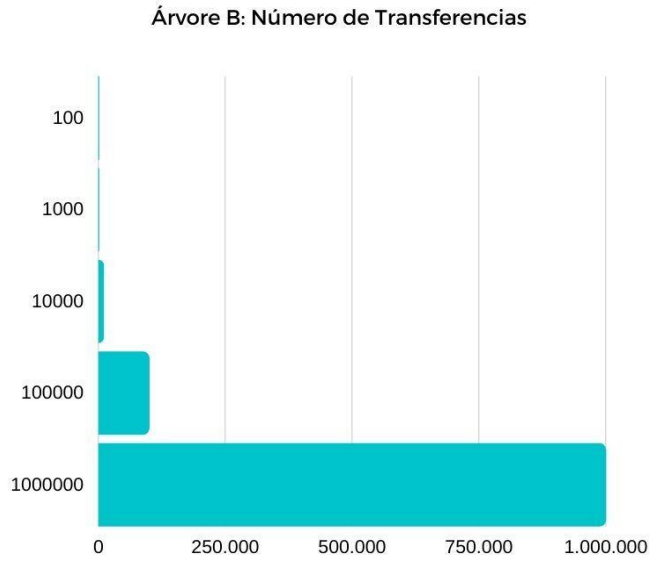

Figura 6 - Número de registros e transferências realizadas

- **Árvore B***

A Árvore B segue um modelo parecido com a Árvore B*, utilizamos as funções já desenvolvidas para o modelo anterior como base. Ainda sim, é necessário alterar algumas partes da lógica do programa, dividindo algumas funções entre: fluxo para um página externa e fluxo para uma página interna. Isso o ocorre pois a estrutura de páginas da Árvore B* é da seguinte forma:

```
#define N 4
#define NN N*2
typedef enum {Interna, Externa} TipoPaginaBX;
typedef struct PaginaBX * ApontadorBX;
typedef struct PaginaBX{
    TipoPaginaBX tipo;
      union {
        struct { // pagina index
            int ni;
            int ri[NN];
            ApontadorBX pi[NN + 1];
        } U0;

        struct { // folha
            int ne;
            Registro re[NN];
        } U1;
    } UU;
} PaginaBX;
```

A Página da Árvore B*, a TAD PaginaBX, recebe uma variável para definir o seu tipo, entre Interna e Externa, para que o fluxo seja separado. Assim, o programa pode acessar a estrutura correta por meio da union de cada caso, U0 ou U1, página interna de index e folha, respectivamente. Logo várias funções tem seu fluxo separado entre Interno e Externo:

```
int ins_bx( Registro registro, ApontadorBX
raiz, int * bx_comparacoes, Registro *
reg_out, ApontadorBX * pag_out){
    if(raiz->tipo == Externa)
        return ins_externa_bx(registro, raiz,
bx_comparacoes, reg_out, pag_out);
    else
        return ins_interna_bx(registro, raiz,
bx_comparacoes, reg_out, pag_out);
}
```

- Divisão em dois terços

Outra peculiaridade na Árvore B* é o fato de suas páginas, no caso de estarem cheias, sofrem uma divisão de ⅔, ou dois terços. Isso acontece pois o programa tenta balancear a árvore de modo que cada página fique em média dois terços cheias. Para isso, o código redistribui as chaves para preencher duas páginas vizinhas e depois divide entre três páginas. Desse modo, o espaço é utilizado de maneira eficiente.

- Análise Experimental

Nos testes de mesa, a Árvore B* se mostrou bastante eficiente, mantendo o mesmo tempo de execução independente da organização do arquivo. Em todos os casos, crescente, decrescente e aleatório, a estrutura realiza a pesquisa no mesmo tempo nos 3 casos (um mesmo valor leva o mesmo tempo em cada tipo de arquivo).

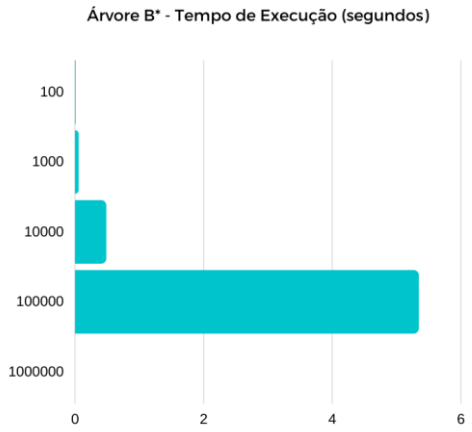

Figura 7 - Número de registros e tempo de execução

Já na quantidade de comparações, o programa tem diferenças caso o arquivo seja aleatório ou não, uma vez que a ordem de entrada diferente muda a ordem de alteração e criação de páginas.

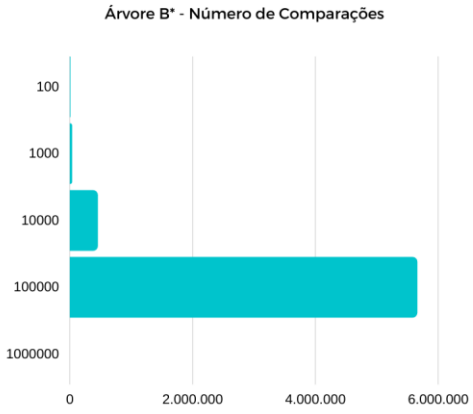

Figura 8 - Tempo de execução e número de comparações

Sobre o número de transferências, a Árvore B* é bastante eficiente, de modo que o número de acessos ao arquivo é sempre igual à quantidade de registros do arquivo. Independente do caso. Portanto, as transferências são minimizadas, mas sendo possível reduzi-las ainda mais caso seja implementado uma leitura de páginas, ou de vários registros, de uma vez, reduzindo ainda mais a quantidade de transferências.