UNIVERSIDADE FEDERAL DE OURO PRETO - UFOP
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS - ICEB
DEPARTAMENTO DE COMPUTAÇÃO - DECOM

ANANDA MENDES SOUZA
RÔMULO DE OLIVEIRA CARNEIRO

PESQUISA EXTERNA

RELATÓRIO TRABALHO PRÁTICO I

OURO PRETO
2021

ANANDA MENDES SOUZA
RÔMULO DE OLIVEIRA CARNEIRO

PESQUISA EXTERNA

RELATÓRIO TRABALHO PRÁTICO I

Relatório apresentado à disciplina Estrutura de Dados II da matriz curricular do Curso de Ciência da Computação da Universidade Federal de Ouro Preto, a ser utilizado como parte das exigências do Trabalho Prático I – Pesquisa Externa.

OURO PRETO

2021

SUMÁRIO

1. INTRODUÇÃO

O problema proposto por esse trabalho é o da busca (ou pesquisa) dado um conjunto de elementos, onde cada um é identificado por uma chave, o objetivo é localizar da forma mais eficiente possível, nesse conjunto, o elemento que corresponde a uma chave específica.

Vários métodos e estruturas de dados podem ser empregados para resolver esse problema. Certos métodos de organização/ordenação de dados podem tornar o processo de busca mais eficiente.

Dito isso, quatro estruturas foram implementadas e analisadas, sendo elas: *Acesso Sequencial Indexado, Árvore Binária adequada à memória externa, Árvore B e Árvore B**.

2. OBJETIVO

Realizar um estudo da complexidade de desempenho dos métodos de pesquisa externa exigidos, e como estes algoritmos se portam nos quesitos de número de transferências (entre a memória externa e a memória interna), número de comparações (entre chaves de pesquisa) e tempo de execução (tempo do término de execução menos o tempo do início de execução).

3. METODOLOGIA

Foi desenvolvido um algoritmo completo em linguagem C++ que simula o funcionamento de todos os métodos de pesquisa propostos. O programa foi implementado de tal forma a ser capaz de ser executado, livremente, a partir da seguinte linha de comando no console:

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

onde:

<executável> representa o nome do programa;

<método> representa o método de pesquisa externa a ser executado, podendo ser um número inteiro de 1 a 4, sendo:

[1] – Pesquisa Sequencial Indexada;

[2] – Árvore Binária adequada à memória externa;

[3] – Árvore B;

[4] – Árvore B*;

<quantidade> representa a quantidade de registros do arquivo considerado, um número entre 1 e 2.000.000 registros quaisquer;

<situação> representa a situação de ordem do arquivo, podendo ser um número entre 1 e 3, sendo:

[1] – Arquivo ordenado ascendentemente (Crescente);

[2] – Arquivo ordenado descendentemente (Decrescente);

[3] – Arquivo desordenado aleatoriamente (Aleatório);

<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.

4. DESENVOLVIMENTO

A implementação de cada estrutura foi dividida em arquivos, para se obter um main.cpp (arquivo principal) mais sucinto, objetivo e de fácil entendimento. Todas as estruturas estão contidas em um arquivo cabeçalho (.h) chamado header com os e protótipos dos métodos e um arquivo (.cpp) com as respectivas implementações. Desse modo, serão discutidos o funcionamento, implementação e a complexidade dos seguintes algoritmos:

I. Acesso Sequencial Indexado;
II. Árvore Binária adequada à memória externa;
III. Árvore B;
IV. Árvore B*;

Descrição do experimento

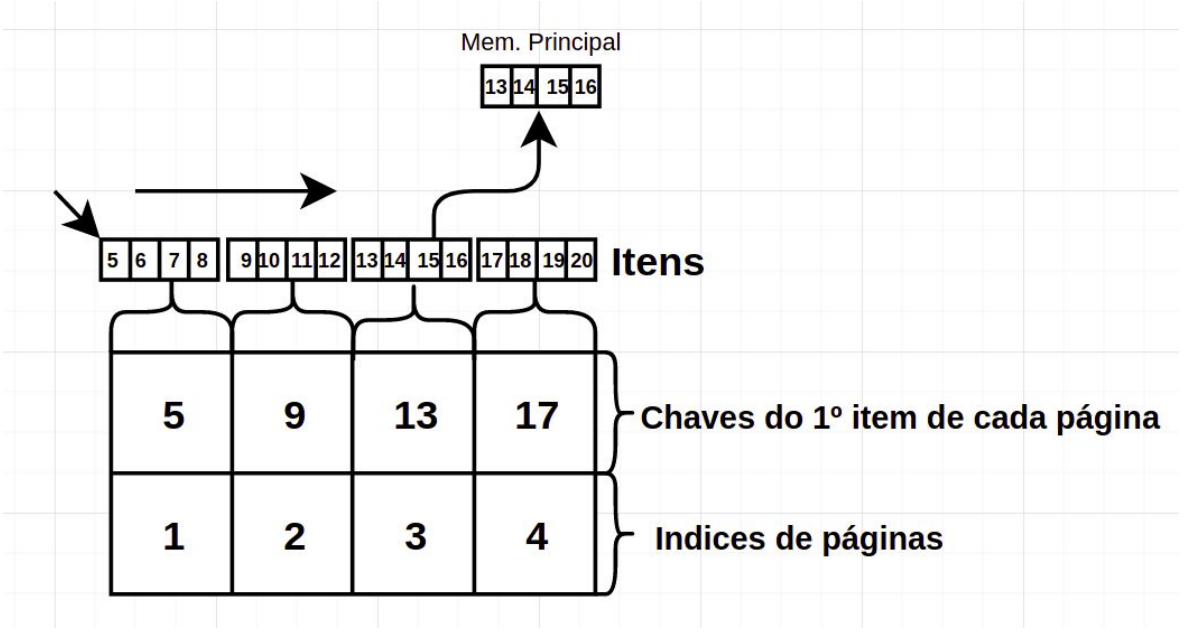
A 1ª fase deste trabalho corresponde à implementação em C++ dos métodos mencionados, considerando arquivos binários de registros quaisquer e memória interna disponível para armazenar os índices necessários, quando for o caso.

A 2ª fase corresponde à análise experimental da complexidade de desempenho dos métodos mencionados, considerando as etapas de criação dos índices necessários, quando for o caso, e da própria pesquisa. Foram realizados experimentos considerando arquivos contendo 100, 1.000, 10.000, 100.000 e 1.000.000 registros quaisquer cujas chaves serão pesquisadas.

Para cada experimento, considerando os mesmos parâmetros (método de ordenação, quantidade de registros, situação de ordem do arquivo), deve-se ocorrer a pesquisa automática de 10 chaves de pesquisa distintas, bem diferenciadas e existentes no arquivo em questão, no intuito de se obter a média de cada um dos quesitos a serem considerados no processo de análise experimental.

Acesso Sequencial Indexado

O acesso sequencial indexado é um dos métodos de pesquisa externa mais simples de serem implementados, ele utiliza um método de paginação para ler o arquivo em intervalos, onde os mesmos são salvos em uma tabela em memória principal e com isso eles auxiliam na busca por uma determinada chave. Exemplo:



No exemplo acima podemos ver um arquivo “Impares.bin”, utilizando o método de acesso sequencial indexado este arquivo é lido em intervalos determinados, no exemplo o intervalo adotado é 10, este intervalo é o tamanho da página. Os valores lidos são armazenados em memória principal em uma tabela. Ao realizar uma pesquisa verifica se a chave buscada é pertencente a um intervalo da tabela quando isto é verdadeiro, ele lê a página inteira do arquivo e realiza uma busca sequencial.

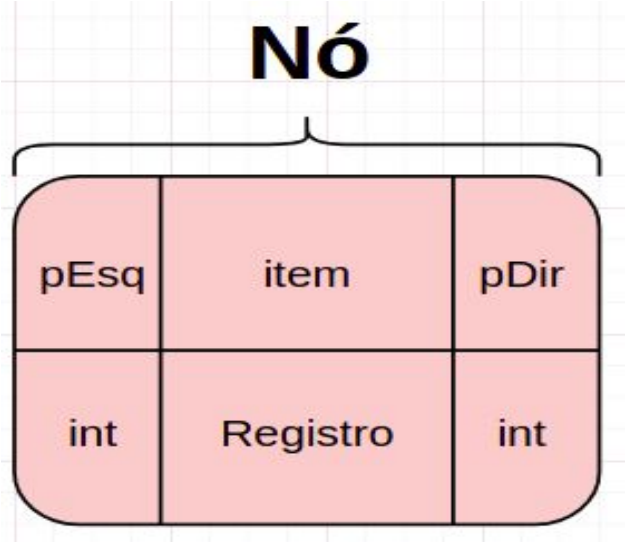
Quesitos para funcionamento do Acesso indexado Sequencial
Na implementação feita neste trabalho, o Acesso Sequencial indexado funciona nas seguintes situações: 1. Arquivo ordenado ascendentemente– para os tamanhos especificados na descrição do trabalho. 2. Arquivo ordenado descendentemente– para os tamanhos especificados na descrição do trabalho. **OBS: Este método não se aplica para arquivos gerados aleatoriamente, pois o arquivo precisa estar ordenado.**

Vantagens e desvantagens

A principal vantagem é a utilização do sistema de paginação que diminui o número de acessos ao arquivo, o que tem um alto custo. No entanto, possui como desvantagem a inserção dos registros em arquivo que devem ser feitos ordenadamente (ascendente ou descendentemente), ou serem tratados com algum tipo de ordenação prévia.

Árvore Binária em memória externa

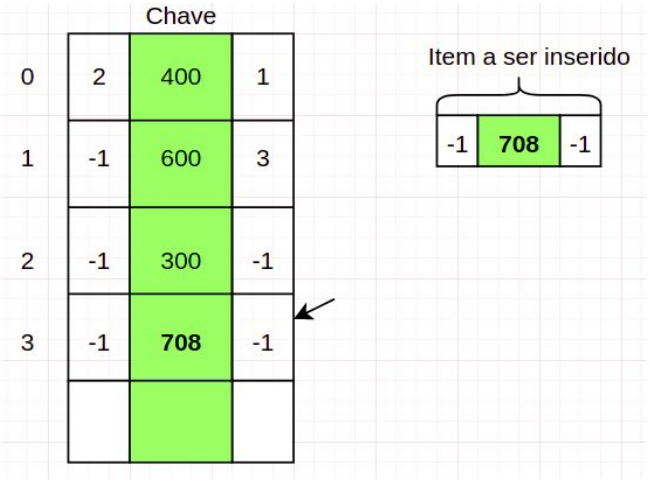
A árvore binária em memória externa consiste em simular uma árvore binária feita em memória principal porém em uma memória secundária para isso simulamos o uso de ponteiros da seguinte forma:



Onde:

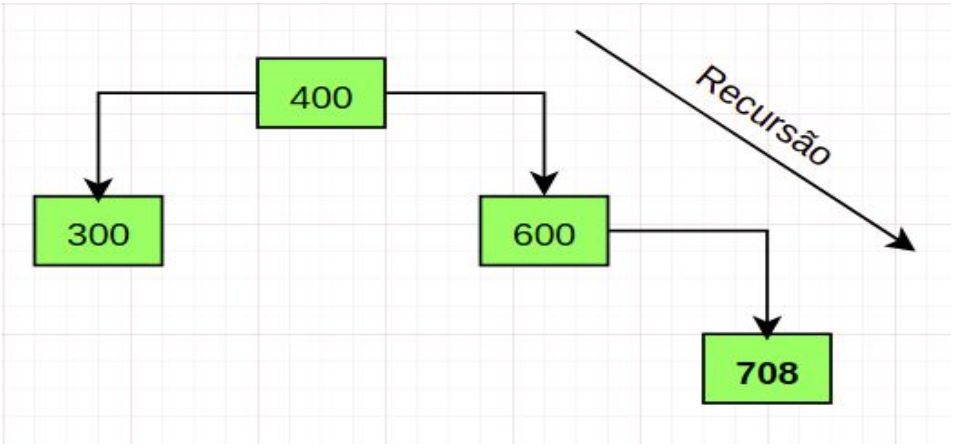
- *Item* é do tipo *Registro* e guarda a chave e os dados do arquivo original.
- *pEsq* e *pDir* são do tipo inteiro e indicam em qual posição do arquivo o dado com chave menor e maior se encontram respectivamente.

Dessa forma, a árvore é lida através do arquivo principal e armazenada em um outro arquivo externo seguindo os critérios de construção de uma árvore como mostra a ilustração abaixo:



Os nodos da árvore são armazenados em disco e os seus apontadores à esquerda e à direita armazenam endereços de disco ao invés de endereços de memória principal.

Um ilustração de como seria a árvore em memória principal:



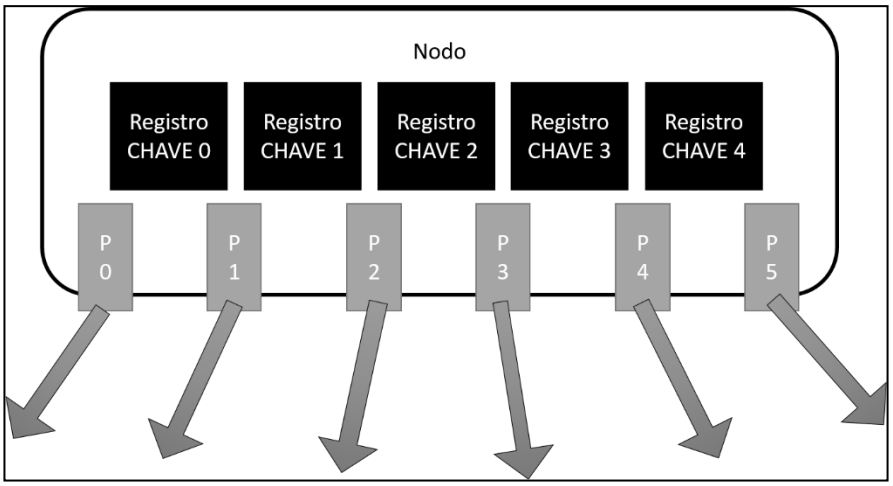
Quesitos para funcionamento da Árvore Binária em memória externa
Na implementação feita neste trabalho, a Árvore Binária em memória externa funciona nas seguintes situações: 1. Arquivo ordenado ascendentemente – para os tamanhos especificados na descrição do trabalho. 2. Arquivo ordenado descendentemente – para os tamanhos especificados na descrição do trabalho. 3. Arquivo ordenado aleatoriamente – para os tamanhos especificados na descrição do trabalho.

Vantagens e desvantagens

A principal vantagem é que muito menos acessos ao disco são realizados, visto que o formato da árvore muda de binário para quartenário, o que reduz pela metade o nº de acessos a disco no pior caso. Já a desvantagem é que a ocupação média das páginas é baixa(ordem de 10%).

Árvore B

A árvore B é uma árvore de pesquisa binária que busca armazenar os registros em nodos, esses nodos possuem um certo tamanho definido e para cada nodo existem ponteiros para outros nodos, esses ponteiros são estabelecidos de acordo com os intervalos dos registros salvos dentro do nodo pai. Assim como na ilustração a seguir:



A construção se dá percorrendo o arquivo onde encontram-se os registros de forma linear e formando a árvore na memória principal. A cada item adicionado, é verificado se não houve quebra nas propriedades da árvore e realizando balanceamentos sempre que necessário, obtendo-se assim uma árvore sempre balanceada.

A pesquisa é realizada da mesma forma que uma árvore binária: procura- se o registro comparando os valores das chaves, caso maior, o algoritmo percorre a subárvore à direita, caso menor, percorre a subárvore à esquerda, até encontrar o registro requisitado ou encontrar um nó folha, indicando que o registro não encontra-se na árvore.

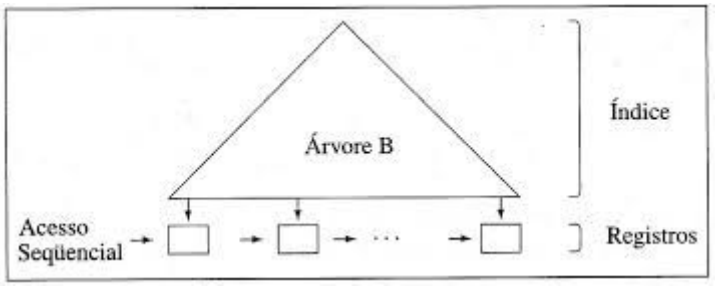
Quesitos para funcionamento da Árvore B
Na implementação feita neste trabalho, a Árvore B funciona nas seguintes situações: 1. Arquivo ordenado ascendentemente – para os tamanhos especificados na descrição do trabalho. 2. Arquivo ordenado descendentemente – para os tamanhos especificados na descrição do trabalho. 3. Arquivo ordenado aleatoriamente – para os tamanhos especificados na descrição do trabalho.

Vantagens e desvantagens

A vantagem é ter em sua estrutura um número maior de itens por nodo, aqui chamados de página, pois reduz o custo de acesso e pesquisa, e é uma árvore balanceada com relação a altura independente da ordem de inserção dos dados. Uma desvantagem é não oferecer solução econômica para indexação por chave secundária ou com alta redundância de valor de chave.

Árvore B*

A árvore B* é uma alternativa para implementação da árvore B, onde todos os itens estão armazenados no último nível e os níveis acima constituem um índice cuja organização é a de uma árvore B, assim há uma separação lógica entre o índice e os itens como mostrada na imagem abaixo:



Na construção é necessário identificar se as páginas são internas ou externas, portanto a estrutura é diferente de uma árvore B. Já as operações dentro de uma árvore B* são bastante similares às da árvore B, na inserção a única diferença é que quando uma folha é dividida em duas, o algoritmo promove uma cópia da chave do meio e leva para a página pai, e deixando o item na folha da direita, e na pesquisa é que ao encontrar a chave desejada em uma página do índice, a pesquisa continua até que se encontre em uma página folha.

Quesitos para funcionamento da Árvore B*
Na implementação feita neste trabalho, a Árvore B* funciona nas seguintes situações: 1. Arquivo ordenado ascendentemente – para os tamanhos especificados na descrição do trabalho. 2. Arquivo ordenado descendentemente – para os tamanhos especificados na descrição do trabalho. 3. Arquivo ordenado aleatoriamente – para os tamanhos especificados na descrição do trabalho.

Vantagens e desvantagens

As vantagens é que como a árvore B* não possui uma página irmã(não faz uso de apontadores nas páginas folha) , é possível armazenar uma quantidade maior de itens nas mesmas, tornando a ordem maior, além disso a árvore B* tem um acesso sequencial mais eficiente e facilita o acesso concorrente ao arquivo. E a desvantagem é que deve-se tomar um cuidado especial com o nó raiz e para ele usar one-to- two splitting (algoritmos mais complexos).

5. TESTES E RESULTADOS

Foram realizados os testes referentes à criação de todas as estruturas na ordem ascendente, descendente e aleatoriamente de dados através da utilização da média de 5 entradas, sendo elas:

- 0, 25, 50, 75 e 99, para estrutura de 100 de registros;
- 0, 250, 500, 750 e 999 para estrutura de 1.000 de registros;
- 0, 2500, 5000, 7500 e 9999 para estrutura de 10.000 de registros;
- 0, 25000, 50000, 75000 e 99999 para estrutura de 100.000 de registros;
- 0, 250.000, 500.000, 750.000 e 999.999 para estrutura de 1.000.000 de registros.

Foram obtidos os seguintes resultados:

ORDENADO ASCENDENTEMENTE

ACESSO SEQUENCIAL INDEXADO					
Nº DE DADOS	100	1.000	10.000	100.000	1.000.000
TRANSFERÊNCIAS	1	1	1	1	1
COMPARAÇÕES	16	128	1252	12.502	125.002
TEMPO EM SEGUNDOS	0,0001	0,00008	0,00008	0,0001226	0,0004

ÁRVORE BINÁRIA EM MEMÓRIA EXTERNA					
Nº DE DADOS	100	1.000	10.000	100.000	1.000.000
TRANSFERÊNCIAS	349	3.285	34.851	--	--
COMPARAÇÕES	50	486	3881	--	--
TEMPO EM SEGUNDOS	0,000 218	0,39	0,0036	--	--

ÁRVORE B					
Nº DE DADOS	100	1.000	10.000	100.000	1.000.000
TRANSFERÊNCIAS	1	1	1	1	1
COMPARAÇÕES	5	8	12	15	18
TEMPO EM SEGUNDOS	0,00008	0,00007	0,00005	0,00005	0,0000482

ÁRVORE B*					
Nº DE DADOS	100	1.000	10.000	100.000	1.000.000
TRANSFERÊNCIAS	1	1	1	1	1
COMPARAÇÕES	8	11	17	22	26
TEMPO EM SEGUNDOS	0,00023	0,00009	0,00005	0,000046	0,000047

ORDENADO DESCENDENTEMENTE					

ACESSO SEQUENCIAL INDEXADO					
Nº DE DADOS	100	1.000	10.000	100.000	1.000.000
TRANSFERÊNCIAS	1	1	1	1	1
COMPARAÇÕES	16	128	1.253	12.503	125.003
TEMPO EM SEGUNDOS	0,00007	0,00012	0,00019	0,00012	0,00042

ÁRVORE BINÁRIA EM MEMÓRIA EXTERNA					
Nº DE DADOS	100	1.000	10.000	100.000	1.000.000
TRANSFERÊNCIAS	348	3.484	34.850	--	--
COMPARAÇÕES	49	486	4851	--	--
TEMPO EM SEGUNDOS	0,018	0,00039	37,7	--	--

ÁRVORE B					
Nº DE DADOS	100	1.000	10.000	100.000	1.000.000
TRANSFERÊNCIAS	1	1	1	1	1
COMPARAÇÕES	5	7	11	13	16
TEMPO EM SEGUNDOS	0,00006	0,00007	0,000058	0,000047	0,000049

ÁRVORE B*					
Nº DE DADOS	100	1.000	10.000	100.000	1.000.000
TRANSFERÊNCIAS	1	1	1	1	1
COMPARAÇÕES	8	10	13	16	21
TEMPO EM SEGUNDOS	0,00008	0,0001	0,00005	0,0000444	0,000043

ORDENADO ALEATORIAMENTE					
ÁRVORE BINÁRIA EM MEMÓRIA EXTERNA					
Nº DE DADOS	100	1.000	10.000	100.000	1.000.000
TRANSFERÊNCIAS	229	2.296	18.202	--	--
COMPARAÇÕES	6	9	14	--	--
TEMPO EM SEGUNDOS	0,00017	0,0000554	0,00007	--	--

ÁRVORE B					
Nº DE DADOS	100	1.000	10.000	100.000	1.000.000
TRANSFERÊNCIAS	1	1	1	1	1
COMPARAÇÕES	5	9	11	13	20
TEMPO EM SEGUNDOS	0,00007 46	0,00006 16	0,00005	0,0000418	0,000042

ÁRVORE B*					
Nº DE DADOS	100	1.000	10.000	100.000	1.000.00 0
TRANSFERÊNCIAS	1	1	1	1	1
COMPARAÇÕES	8	12	16	22	26
TEMPO EM SEGUNDOS	0,000059	0,00006	0,000066	0,000043	0,000049

6. CONSIDERAÇÕES FINAIS

Podemos concluir, através dos resultados obtidos na fase de testes, que a Árvore B e B* foram os métodos mais eficientes durante a execução do programa.

A Árvore Binária em memória externa se mostrou ineficiente para o problema proposto devido a sua demora, número de transferências e comparações durante a execução. Dito isso, ao executar o programa com mais de 100.000 registros o programa demorou horas para ser executado, não sendo necessário realizar mais testes a partir daí, por já obtermos que os outros algoritmos demoraram bem menos, o que satisfazia a solução buscada.

A principal dificuldade do grupo foi justamente a implementação da árvore Binária em memória externa, onde a princípio tentamos construir o algoritmo de maneira recursiva e sem a criação de um segundo arquivo ordenado para a árvore, assim estouramos a fila de recursão disponível.

Por fim, este trabalho mostrou como os métodos de pesquisa em memória são implementados nos ajudando a entender o funcionamento de cada um.

7. CÓDIGOS

Header

```
#ifndef HEADER_H
#define HEADER_H

#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <iomanip>
#include <cstring>
#include <ctime>

//DECLARAÇÕES DE USO GERAL
#define ITENSPAGINA 4
#define ORDEM 2

#define CRESCENTE "Crescente"
#define DECRESCENTE "Decrescente"
#define ALEATORIO "Aleatorio"

using namespace std;

typedef struct {
  int posicao;
  int chave;
} TipoIndice;

typedef struct {
  int chave;
  long int dado1;
  char dado2[501];
} TipoRegistro;

void validaEntradaDeDados(int argc, char **argv, int& metodo, int&
quantidade, int& situacao, int& chave, bool& P);
void algoritmosDePesquisaExterna(string situacao, int quantidade,
TipoIndice* tabela, int chave, int metodo, bool P);
```

```
//FUNÇÃO NECESSÁRIA PARA GERADOR
int criadorDeArquivo (int numRegistros, string cresDecresAlea);

//FUNÇÃO NECESSÁRIA PARA ARQUIVO SEQUENCIAL
int pesquisaAcessoSequencial(TipoIndice tabela[], int tamanho, TipoRegistro
*item, FILE *arquivo, string situacao, int& numero_comparacoes, int&
numero_transferencias);

// STRUCTS E FUNÇÕES NECESSÁRIAS PARA ÁRVORE BINÁRIA
typedef struct No {
  TipoRegistro registro;
  long long int ponteiroEsq, ponteiroDir;
} TipoNo;

typedef TipoNo ArvoreBinaria;

bool criaArvoreBinaria (string caminho);
bool organizaArvoreBinaria (FILE* saida, TipoIndice *ind);
int pesquisaArvoreBinaria(FILE* saida, TipoRegistro *registro, int
*numero_comparacoes);

// STRUCTS E FUNÇÕES NECESSÁRIAS PARA ÁRVORE B
typedef struct TipoPagina *TipoApontador;

typedef struct TipoPagina {
  short quantItens;
  TipoRegistro registro[2*ORDEM];
  TipoApontador ponteiro[2*ORDEM + 1];
} TipoPagina;

bool pesquisaArvoreB (TipoRegistro *registro, TipoApontador apontador, int&
numero_comparacoes);
void insereNaPagina (TipoApontador apontador, TipoRegistro registro,
TipoApontador apontadorDireita, int& numero_comparacoes);
void ins(TipoRegistro registro, TipoApontador apontador, short *cresceu,
TipoRegistro *regRetorno, TipoApontador *apRetorno, int&
numero_comparacoes);
void insereArvoreB (TipoRegistro registro, TipoApontador *apontador, int&
numero_comparacoes);

// STRUCTS E FUNÇÕES NECESSÁRIAS PARA ÁRVORE B*
typedef enum {Interna, Externa} TipoIntExt;

typedef struct Pagina* Apontador;

typedef struct Pagina {
```

```
  TipoIntExt Pt;
  union {
    struct {
      int quantItensInt;
      int chave[2*ORDEM];
      Apontador ponteiro[2*ORDEM + 1];
    } Interna;
    struct {
      int quantItensExt;
      TipoRegistro registro[3*ORDEM];
      Apontador prox;
    } Externa;
  } IntOrExt;
} Pagina;

void InsereNaPagina (Apontador apontador, int registro, Apontador
apontadorDireita, int& numero_comparacoes); // IGUAL ARVORE B
void Ins (TipoRegistro* registro, Apontador apontador, short *cresceu,
TipoRegistro *regRetorno, Apontador *apRetorno, short *Cresceu_No, int&
numero_comparacoes);
void InsereArvoreBStar (TipoRegistro registro, Apontador* apontador, int&
numero_comparacoes);
bool InsereNaFolha (TipoRegistro* registro, Apontador novaPagina, int&
numero_comparacoes);
void imprimeBStar (Apontador arvore);
bool pesquisaArvoreBStar(TipoRegistro *registro, Apontador apontador, int&
numero_comparacoes);

#endif
```

Gerador

```
#include "header.h"

int main () {
  srand(unsigned(time(NULL)));

  criadorDeArquivo(100, CRESCENTE);
  criadorDeArquivo(1000, CRESCENTE);
  criadorDeArquivo(10000, CRESCENTE);
  criadorDeArquivo(100000, CRESCENTE);
  criadorDeArquivo(1000000, CRESCENTE);
  criadorDeArquivo(100, DECRESCENTE);
  criadorDeArquivo(1000, DECRESCENTE);
  criadorDeArquivo(10000, DECRESCENTE);
  criadorDeArquivo(100000, DECRESCENTE);
  criadorDeArquivo(1000000, DECRESCENTE);
```

```
  criadorDeArquivo(100, ALEATORIO);
  criadorDeArquivo(1000, ALEATORIO);
  criadorDeArquivo(10000, ALEATORIO);
  criadorDeArquivo(100000, ALEATORIO);
  criadorDeArquivo(1000000, ALEATORIO);
  return 0;
}
int criadorDeArquivo (int numRegistros, string cresDecresAlea) {
  FILE* arq;
  tipoRegistro registro;
  int i, j, aux;

  //CRIANDO UM NOME PADRONIZADO PARA TODOS OS ARQUIVOS: REGISTROS +
CRESCENTE/DECRESCENTE/ALETORIO + QUANTIDADE
  string arquivo = "registros"+ cresDecresAlea + to_string(numRegistros) +
".bin";

  if ((arq = fopen(arquivo.c_str(), "w+b")) == NULL) {
      return 0;
  }

    if (cresDecresAlea == CRESCENTE){
      for (i = 0; i < numRegistros; i++) {
        registro.chave = i;
        registro.dado1 = rand() % numRegistros;
        for (j = 0; j < 500; j++) {
          aux = 97 + rand() % 26;
          registro.dado2[j] = static_cast<char>(aux);
        }
        fwrite(&registro, sizeof(tipoRegistro), 1, arq);
      }
    }
    else if (cresDecresAlea == DECRESCENTE){
      for (i = numRegistros-1; i >= 0; i--) {
        registro.chave = i;
        registro.dado1 = rand() % numRegistros;
        for (j = 0; j < 500; j++) {
          aux = 97 + rand() % 26;
          registro.dado2[j] = static_cast<char>(aux);
        }
        fwrite(&registro, sizeof(tipoRegistro), 1, arq);
      }
    }
    else if (cresDecresAlea == ALEATORIO){
      for (i = 0; i < numRegistros; i++) {
        registro.chave = rand() % numRegistros;
```

```
        registro.dado1 = rand() % numRegistros;
        for (j = 0; j < 500; j++){
          aux = 97 + rand() % 26;
          registro.dado2[j] = static_cast<char>(aux);
        }
        fwrite(&registro, sizeof(tipoRegistro), 1, arq);
      }
    }

  fclose(arq);
  return 1;
}
```

Acesso Sequencial

```
#include "header.h"

int pesquisaAcessoSequencial (TipoIndice tabela[], int tamanho,
TipoRegistro *registro, FILE *arquivo, string situacao, int&
numero_comparacoes, int& numero_transferencias) {
  TipoRegistro pagina[ITENSPAGINA];
  long deslocamento;
  int i = 0, quantItens;

  // PROCURA PELA PÁGINA ONDE O REGISTRO PODE SE ENCONTRAR
  //EM CASO DECRESCENTE O WHILE MUDA
  if (situacao == DECRESCENTE) {
    while(i < tamanho && tabela[i].chave >= registro->chave) {
      i++;
      numero_comparacoes++;
    }
  }
  else {
    while(i < tamanho && tabela[i].chave <= registro->chave) {
      i++;
      numero_comparacoes++;
    }
  }

  // CASO A CHAVE DESEJADA SEJA MENOR QUE A 1ª CHAVE, O REGISTRO NÃO EXISTE
NO ARQUIVO
  if (i == 0) {
    return 0;
  }
  else {
    // A ÚLTIMA PÁGINA PODE NÃO ESTAR COMPLETA
```

```
    if (i < tamanho) {
      quantItens = ITENSPAGINA;
    }
    else {
      fseek (arquivo, 0, SEEK_END);
      quantItens = (ftell(arquivo) / sizeof(TipoRegistro)) % ITENSPAGINA;
       if(quantItens == 0) {
         quantItens = ITENSPAGINA;
       }
    }

    // LÊ A PÁGINA DESEJADA DO ARQUIVO
    deslocamento = (tabela[i-1].posicao-1) * ITENSPAGINA *
sizeof(TipoRegistro);
    fseek (arquivo, deslocamento, SEEK_SET);
    fread (&pagina, sizeof(TipoRegistro), quantItens, arquivo);
    numero_transferencias++;

    // PESQUISA SEQUENCIAL NA PÁGINA LIDA
    for (i = 0; i < quantItens; i++) {
      numero_comparacoes++;
      if (pagina[i].chave == registro->chave) {
        *registro = pagina[i];
        return 1;
      }
    }
    return 0;
  }
  return 0;
}
```

Árvore Binária

```
#include "header.h"

//CRIANDO UMA ÁRVORE VAZIA
bool criaArvoreBinaria (string caminho) {
    FILE* entrada, *saida;
    //VARIÁVEL AUXILIAR PARA ESCRITA EM ARQUIVO
    ArvoreBinaria arvore_write;
    //VARIÁVEL AUXILIAR PARA LEITURA EM ARQUIVO
    TipoRegistro registro;
    //VARIÁVEL AUXILIAR PARA GUARDAR A POSIÇÃO E CHAVE DO NÓ INSERIDO
    TipoIndice ind;
    ind.posicao = 0;
    bool insira = false;
```

```
    entrada = fopen(caminho.c_str(), "rb");

    saida = fopen("ArvoreBinariaExterna.bin", "w+b");
    if (entrada == NULL || saida == NULL) {
        return false;
    }

    while (fread(&registro, sizeof(TipoRegistro), 1, entrada) == 1){
        //INICIALIZA AS FOLHAS DO NÓ A SER INSERIDO
        arvore_write.ponteiroDir = -1;
        arvore_write.ponteiroEsq = -1;

        arvore_write.registro = registro;
        //SALVA A CHAVE E POSIÇÃO DO NÓ A SER INSERIDO
        ind.chave = arvore_write.registro.chave;

        if (ftell(saida) != 0) {
            ind.posicao = ftell(entrada)/sizeof(ArvoreBinaria);

            //VOLTANDO COM O PONTEIRO PARA O INÍCIO DO ARQUIVO
            rewind(saida);

            //VERIFICA SE O ELEMENTO É UNICO NA ÁRVORE
            if (ind.posicao != 0) {
                insira = organizaArvoreBinaria(saida, &ind);
            }

            //VOLTA O PONTEIRO DO ARQUIVO PRO FINAL PARA CONTINUAR A
INSERÇÃO
            if (fseek(saida, sizeof(ArvoreBinaria)*(ind.posicao), SEEK_SET)
!= 0) {
                break;
            }

            //VERIFICA SE O ELEMENTO EXISTE NA ARVORE
            if(insira) {
                fwrite(&arvore_write, sizeof(ArvoreBinaria), 1, saida);
            }
        }
        else {
            //SE FOR O PRIMEIRO ITEM DA ÁRVORE
            fwrite(&arvore_write, sizeof(ArvoreBinaria), 1, saida);
        }
    }
    fclose(entrada);
    fclose(saida);
```

```
    return true;
}

bool organizaArvoreBinaria (FILE* saida, TipoIndice* ind) {
    //VARIÁVEL AUXILIAR PARA LEITURA EM ARQUIVO
    ArvoreBinaria arvore_read;
    bool inseriu = false;
    //SALVA A POSIÇÃO DO NÓ PAI PARA SOBRESCREVER COM A POSIÇÃO DOS FILHOS
    TipoIndice aux;

    //SE NÓ PAI AINDA NÃO FOI ATRIBUÍDO AO NÓ INSERIDO, LÊ O PRIMEIRO ITEM
    while (!inseriu) {
        aux.posicao = ftell(saida);

        if (aux.posicao != 0) {
            aux.posicao = aux.posicao/sizeof(ArvoreBinaria);
        }

        if (fread(&arvore_read, sizeof(ArvoreBinaria), 1, saida) != 1) {
            break;
        }

        if (ind->chave == arvore_read.registro.chave) {
            inseriu = false;
            break;
        }

        //VAI PARA DIREITA
        if (ind->chave > arvore_read.registro.chave) {
            //VAI PARA DIREITA
            //NÓ NÃO ESTÁ OCUPADO
            if (arvore_read.ponteiroDir == -1) {

                //SOBREESCREVE O -1 COM A POSIÇÃO DO FILHO
                arvore_read.ponteiroDir = ind->posicao;

                //POSICIONA O PONTEIRO DO ARQUIVO PARA SOBREESCREVER O NÓ
PAI COM A POSIÇÃO DO FILHO
                fseek(saida, sizeof(ArvoreBinaria) * aux.posicao,
SEEK_SET);

                fwrite (&arvore_read, sizeof(ArvoreBinaria), 1, saida);
                inseriu = true;
                break;
            }
```

```
            //NÓ OCUPADO = POSICIONA O PONTEIRO NESSE NÓ E CHAMA
RECURSIVAMENTE A FUNÇÃO PARA CONTINUAR PROCURANDO
            else {
                fseek(saida, sizeof(ArvoreBinaria) *
arvore_read.ponteiroDir, SEEK_SET);
            }
        }
        else {
            //VAI PARA ESQUERDA
            if (ind->chave < arvore_read.registro.chave) {
                //VAI PARA ESQUERDA
                //NÓ NÃO ESTÁ OCUPADO
                if (arvore_read.ponteiroEsq == -1) {

                    //SOBREESCREVE O -1 COM A POSIÇÃO DO FILHO
                    arvore_read.ponteiroEsq = ind->posicao;

                    //POSICIONA O PONTEIRO DO ARQUIVO PARA SOBREESCREVER O
NÓ PAI COM A POSIÇÃO DO FILHO
                    fseek(saida, sizeof(ArvoreBinaria) * aux.posicao,
SEEK_SET);
                    fwrite(&arvore_read, sizeof(ArvoreBinaria), 1, saida);
                    inseriu = true;
                    break;
                }
                //NÓ OCUPADO = POSICIONA O PONTEIRO NESSE NÓ E CHAMA
RECURSIVAMENTE A FUNÇÃO PARA CONTINUAR PROCURANDO
                else {
                    fseek(saida, sizeof(ArvoreBinaria) *
arvore_read.ponteiroEsq, SEEK_SET);
                }
            }
            //SE CHEGOU ATÉ AQUI: ARQUIVO VAZIO OU CHAVE IGUAL JÁ NA ÁRVORE
        }
    }
    return inseriu;
}

void leArvore () {
    FILE* saida;
    saida = fopen("ArvoreBinariaExterna.bin", "r+b");
    ArvoreBinaria arvore_read;
    int cont = 0;
    while (fread(&arvore_read, sizeof(ArvoreBinaria), 1, saida) == 1) {
        cout << "[" << cont << "] - " << arvore_read.ponteiroEsq
        << "|" << arvore_read.registro.chave
```

```
        << "|" << arvore_read.ponteiroDir << endl;
        cont++;
    }
    fclose(saida);
}

int pesquisaArvoreBinaria(FILE* saida, TipoRegistro *registro, int
*numero_comparacoes) {
    ArvoreBinaria arvore_read;
    while (fread(&arvore_read, sizeof(ArvoreBinaria), 1, saida) == 1) {
        if (arvore_read.registro.chave == registro->chave) {
            *registro = arvore_read.registro;
            return 1;
        }
        else {
            if (registro->chave > arvore_read.registro.chave) {
                if (arvore_read.ponteiroDir != -1) {
                    fseek(saida, sizeof(ArvoreBinaria) *
arvore_read.ponteiroDir, SEEK_SET);
                }
                else {
                    return 0;
                }
            }
            else {
                if (registro->chave < arvore_read.registro.chave) {
                    if (arvore_read.ponteiroEsq != -1) {
                        fseek(saida, sizeof(ArvoreBinaria) *
arvore_read.ponteiroEsq, SEEK_SET);
                    }
                    else {
                        return 0;
                    }
                }
            }
        }
    }
    return 0;
}
```

Árvore B

```
#include "header.h"
```

```
bool pesquisaArvoreB (TipoRegistro *registro, TipoApontador apontador,int&
numero_comparacoes) {
  long i = 1;
  if (apontador == NULL) {
    cout << "TipoRegistro nao esta presente na arvore\n";
    return false;
  }
  // PESQUISA SEQUENCIAL PARA ENCONTRAR O INTERVALO DESEJADO
  while (i < apontador->quantItens && registro->chave >
apontador->registro[i-1].chave) {
    i++;
    numero_comparacoes++;
  }
  // VERIFICA SE A CHAVE DESEJADA FOI LOCALIZADA
  if (registro->chave == apontador->registro[i-1].chave) {
    numero_comparacoes++;
    *registro = apontador->registro[i-1];
    return true;
  }
  // ATIVAÇÃO RECURSIVA DA PESQUISA EM UMA DAS SUBÁRVORES (ESQUERDA OU
DIREITA)
  if (registro->chave < apontador->registro[i-1].chave){
    numero_comparacoes++;
    pesquisaArvoreB(registro, apontador->ponteiro[i-1],
numero_comparacoes);
  }
  else pesquisaArvoreB(registro, apontador->ponteiro[i],
numero_comparacoes);
}
void insereNaPagina (TipoApontador apontador, TipoRegistro registro,
TipoApontador apontadorDireita, int& numero_comparacoes) {
  short NaoAchouPosicao;
  int k;
  k = apontador->quantItens;
  NaoAchouPosicao = (k > 0);
  while (NaoAchouPosicao) {
    if (registro.chave >= apontador->registro[k - 1].chave) {
      numero_comparacoes++;
      //NaoAchouPosicao == false;
      break;
    }
    apontador->registro[k] = apontador->registro[k -1];
    apontador->ponteiro[k + 1] = apontador->ponteiro[k];
    k--;
    if (k < 1) {
```

```
      NaoAchouPosicao = false;
    }
  }
  apontador->registro[k] = registro;
  apontador->ponteiro[k + 1] = apontadorDireita;
  apontador->quantItens++;
}
void ins(TipoRegistro registro, TipoApontador apontador, short *cresceu,
TipoRegistro *regRetorno, TipoApontador *apRetorno, int&
numero_comparacoes) {
  long i = 1;
  long j;
  TipoApontador ApTemp;
  if (apontador == NULL) {
    *cresceu = true;
    (*regRetorno) = registro;
    (*apRetorno) = NULL;
    return;
  }
  while (i < apontador->quantItens && registro.chave >
apontador->registro[i - 1].chave) {
    numero_comparacoes++;
    i++;
  }
  if (registro.chave == apontador->registro[i - 1].chave) {
    numero_comparacoes++;
    //cout << "Erro: Registro ja esta presente\n";
    *cresceu = false;
    return;
  }
  if (registro.chave < apontador->registro[i - 1].chave) {
    numero_comparacoes++;
    i--;
  }
  ins(registro, apontador->ponteiro[i], cresceu, regRetorno, apRetorno,
numero_comparacoes);
  if (!*cresceu) {
    return;
  }
  // PAGINA TEM ESPACO
  if (apontador->quantItens < 2*ORDEM) {
    insereNaPagina(apontador, *regRetorno, *apRetorno, numero_comparacoes);
    *cresceu = false;
    return;
  }
  // OVERFLOW: PAGINA TEM QUE SER DIVIDIDA
```

```
  ApTemp = (TipoApontador)malloc(sizeof(TipoPagina));
  ApTemp->quantItens = 0;
  ApTemp->ponteiro[0] = NULL;
  if(i < ORDEM + 1) {
    insereNaPagina(ApTemp, apontador->registro[2*ORDEM - 1],
apontador->ponteiro[2*ORDEM], numero_comparacoes);
    apontador->quantItens--;
    insereNaPagina(apontador, *regRetorno, *apRetorno, numero_comparacoes);
  }
  else insereNaPagina(ApTemp, *regRetorno, *apRetorno, numero_comparacoes);
  for(j = ORDEM + 2; j <= 2*ORDEM; j++) {
    insereNaPagina(ApTemp, apontador->registro[j - 1],
apontador->ponteiro[j], numero_comparacoes);
  }
  apontador->quantItens = ORDEM;
  ApTemp->ponteiro[0] = apontador->ponteiro[ORDEM + 1];
  *regRetorno = apontador->registro[ORDEM];
  *apRetorno = ApTemp;
}

void insereArvoreB (TipoRegistro registro, TipoApontador *apontador, int&
numero_comparacoes) {
  short cresceu;
  TipoRegistro regRetorno;
  TipoPagina *apRetorno, *apTemp;
  ins(registro, *apontador, &cresceu, &regRetorno, &apRetorno,
numero_comparacoes);
  // ARVORE CRESCE NA ALTURA DA RAIZ
  if (cresceu) {
    apTemp = (TipoPagina*)malloc(sizeof(TipoPagina));
    apTemp->quantItens = 1;
    apTemp->registro[0] = regRetorno;
    apTemp->ponteiro[1] = apRetorno;
    apTemp->ponteiro[0] = *apontador;
    *apontador = apTemp;
  }
}
```

Árvore B*

```
#include "header.h"

void InsereNaPagina (Apontador apontador, int registro, Apontador
apontadorDireita, int& numero_comparacoes) {
  short NaoAchouPosicao;
  int k;
```

```
  k = apontador->IntOrExt.Interna.quantItensInt;
  //VARIÁVEL QUE MANTÉM O CONTROLE DA POSIÇÃO CORRENTE
  NaoAchouPosicao = (k > 0);

  while(NaoAchouPosicao) {
    if (registro >= apontador->IntOrExt.Interna.chave[k-1]) {
      numero_comparacoes++;
      break;
    }
    //COLOCA O REGISTRO E O APONTADOR NUMA POSIÇÃOA FRENTE
    //("arreda" para a direita)
    apontador->IntOrExt.Interna.chave[k] =
apontador->IntOrExt.Interna.chave[k-1];
    apontador->IntOrExt.Interna.ponteiro[k+1] =
apontador->IntOrExt.Interna.ponteiro[k];
    k--;
    if (k < 1) {
      NaoAchouPosicao = false;
    }
  }

  //ACHOU POSIÇÃO
  apontador->IntOrExt.Interna.chave[k] = registro;
  apontador->IntOrExt.Interna.ponteiro[k+1] = apontadorDireita;
  apontador->IntOrExt.Interna.quantItensInt++;
  return;
}

void Ins (TipoRegistro* registro, Apontador apontador, short *cresceu,
TipoRegistro *regRetorno, Apontador *apRetorno, short *Cresceu_No, int&
numero_comparacoes) {
  long i = 1;
  long j;
  Apontador ApTemp;
  Apontador novo=nullptr;

  TipoRegistro aux;

  if (apontador->Pt == Externa) {
    //SE ENTROU QUER DIZER QUE CHEGOU AO ÚLTIMO NÍVEL
    //VERIFICA SE TEM ESPAÇO NA PÁGINA E TENTA INSERIR NA PÁGINA EXTERNA
    if (apontador->IntOrExt.Externa.quantItensExt < (3*ORDEM)) {
      InsereNaFolha (registro, apontador, numero_comparacoes);
      *cresceu = false;
      *Cresceu_No = false;
      //NÃO PRECISA CRESCER NÍVEL NEM DIVIDIR
```

```
    }
    else {
      //SEPARANDO UM NÓ EM DOIS
      //SOBE PRO PAI A CÓPIA DA CHAVE E DIVIDE
      //regRetorno VAI SALVAR A CHAVE DO PAI
      //ApRetorno VAI SALVAR O NOVO NÓ A DIREITA
      novo = (Pagina*)malloc(sizeof(Pagina));
      novo->Pt = Externa;
      novo->IntOrExt.Externa.quantItensExt=0;
      aux = apontador->IntOrExt.Externa.registro[(3*ORDEM)-1];

      //COLOCA O ÚLTIMO ELEMENTO NA PÁGINA A DIREITA
      InsereNaFolha(&aux, novo, numero_comparacoes);

      //DECREMENTA O NÚMERO DE ITENS DESSA PÁGINA
      apontador->IntOrExt.Externa.quantItensExt--;

      //INSERE O NOVO
      bool inseriu = InsereNaFolha(registro, apontador,
numero_comparacoes);

      if (!inseriu) {
        apontador->IntOrExt.Externa.quantItensExt++;
        delete novo;
        *cresceu = false;
        *Cresceu_No = false;
        return;
      }
      //PASSA METADE PARA NOVA FOLHA
      for (int i = (3*ORDEM)/2; i < (3*ORDEM); i++) {
        InsereNaFolha(&(apontador->IntOrExt.Externa.registro[i]), novo,
numero_comparacoes);
        apontador->IntOrExt.Externa.quantItensExt--;
      }

      regRetorno->chave = novo->IntOrExt.Externa.registro[0].chave;
      *apRetorno = novo;
      apontador->IntOrExt.Externa.prox = novo;
      novo->IntOrExt.Externa.prox = nullptr;
      //APÓS INSERIR NA ÁRVORE O NOVO REGISTRO, A CHAVE QUE PRECISA SUBIR
PARA O NÓ PAI PRECISA SER INSERIDA NA ÁRVORE
      registro->chave = novo->IntOrExt.Externa.registro[0].chave;
      //DIVIDIU O NÓ FOLHA
      *Cresceu_No = true;
      return;
    }
```

```
  }
  else {
    //SE A PÁGINA FOR INTERNA
    while (i < apontador->IntOrExt.Interna.quantItensInt &&
registro->chave > apontador->IntOrExt.Interna.chave[i-1]) {
      numero_comparacoes++;
      i++;
    }
    //VERIFICAR DENTRO DO NÓ FOLHA PARA VER SE AINDA EXISTE
    if (registro->chave == apontador->IntOrExt.Interna.chave[i - 1]) {
      numero_comparacoes++;
      *cresceu = false;
      *Cresceu_No =false;
      return;
    }
    if (registro->chave < apontador->IntOrExt.Interna.chave[i - 1]) {
      numero_comparacoes++;
      i--;
    }
    //ENCONTRA O LUGAR ONDE A CHAVE DEVE ENTRAR
    //ENTRA SE A PRÓXIMA PÁGINA FOR INTERNA
    if(!*Cresceu_No) {
      Ins(registro, apontador->IntOrExt.Interna.ponteiro[i], cresceu,
regRetorno, apRetorno,Cresceu_No, numero_comparacoes);
      if (!*cresceu && !*Cresceu_No) return;
      //VERIFICA SE A PÁGINATEM ESPAÇO
      if (apontador->IntOrExt.Interna.quantItensInt < 2*ORDEM) {
        InsereNaPagina(apontador, regRetorno->chave, *apRetorno,
numero_comparacoes);
        *cresceu = false;
        *Cresceu_No =false;
        return;
      }
      //A PÁGINA ESTÁ CHEIA E DEVE SER DIVIDIDA
      ApTemp = (Apontador) malloc (sizeof(Pagina));
      ApTemp->Pt = Interna;
      ApTemp->IntOrExt.Interna.quantItensInt = 0;
      ApTemp->IntOrExt.Interna.ponteiro[0] = nullptr;

      if (i < ((2*ORDEM)/2) + 1) {
        InsereNaPagina(ApTemp,
apontador->IntOrExt.Interna.chave[(2*ORDEM) - 1],
apontador->IntOrExt.Interna.ponteiro[2*ORDEM], numero_comparacoes);
        apontador->IntOrExt.Interna.quantItensInt--;
        InsereNaPagina(apontador, regRetorno->chave, *apRetorno,
numero_comparacoes);
```

```
      }
      else {
        InsereNaPagina(ApTemp, regRetorno->chave, *apRetorno,
numero_comparacoes);
      }
      for (j = ((2*ORDEM)/2) + 2; j <= 2*ORDEM; j++) {
        InsereNaPagina(ApTemp, apontador->IntOrExt.Interna.chave[j-1],
apontador->IntOrExt.Interna.ponteiro[j], numero_comparacoes);
      }
      apontador->IntOrExt.Interna.quantItensInt = (2*ORDEM)/2;
      ApTemp->IntOrExt.Interna.ponteiro[0] =
apontador->IntOrExt.Interna.ponteiro[((2*ORDEM)/2) + 1];
      regRetorno->chave= apontador->IntOrExt.Interna.chave[(2*ORDEM)/2];

      *apRetorno = ApTemp;
      *cresceu = true;
      *Cresceu_No = false;
      return;
    }
  }
}

void InsereArvoreBStar (TipoRegistro* registro, Apontador* apontador,
int& numero_comparacoes) {
  short cresceu = 0;
  short cresceu_No = 0;
  TipoRegistro regRetorno;
  Pagina *apRetorno = nullptr;
  Pagina *ApTemp = nullptr;
  bool nova = false;

  if (*apontador == NULL) {
    //ALOCA A PRIMEIRA PÁGINA
    ApTemp = (Pagina*)malloc(sizeof(Pagina));
    ApTemp->Pt = Externa;
    ApTemp->IntOrExt.Externa.quantItensExt = 0;
    ApTemp->IntOrExt.Externa.prox = nullptr;
    InsereNaFolha(registro, ApTemp, numero_comparacoes);
    *apontador = ApTemp;
    return;
  }
  else {
    if ((*apontador)->Pt == Externa) {
      nova = true;
    }
```

```
    Ins(registro, *apontador, &cresceu, &regRetorno, &apRetorno,
&cresceu_No, numero_comparacoes);
  }
  //VERIFICA SE É O PRIMEIRO NÓ A SER CRIADO
  if ((cresceu_No && nova) || cresceu) {
    ApTemp = (Pagina*) malloc(sizeof(Pagina));
    ApTemp->Pt = Interna;
    ApTemp->IntOrExt.Interna.quantItensInt = 1;
    ApTemp->IntOrExt.Interna.chave[0] = regRetorno.chave;
    ApTemp->IntOrExt.Interna.ponteiro[0] = *apontador;
    ApTemp->IntOrExt.Interna.ponteiro[1] = apRetorno;
    *apontador = ApTemp;
    return;
  }
}

bool InsereNaFolha (TipoRegistro* registro, Apontador NovaPagina, int&
numero_comparacoes) {
  int quantItens = NovaPagina->IntOrExt.Externa.quantItensExt;
  int k = 0;
  int i = 0;

  while (registro->chave > NovaPagina->IntOrExt.Externa.registro[i].chave
&& i < quantItens) {
    numero_comparacoes++;
    i++;
  }

  if(registro->chave == NovaPagina->IntOrExt.Externa.registro[i].chave &&
quantItens != 0) {
    numero_comparacoes++;
    return false;
  }

  if (i < quantItens) {
    k = quantItens;
    while (k >= 0 && k > i) {
      NovaPagina->IntOrExt.Externa.registro[k] =
NovaPagina->IntOrExt.Externa.registro[k-1];
      k--;
    }
    //INSERE NA POSIÇÃO i
    NovaPagina->IntOrExt.Externa.registro[i] = *registro;
  }
  else {
    //INSERE NO FINAL
```

```
    NovaPagina->IntOrExt.Externa.registro[i] = *registro;
  }

  NovaPagina->IntOrExt.Externa.quantItensExt++;

  return true;
}

void imprimeBStar (Apontador arvore) {
  int i = 0;
  Apontador aux;

  if(arvore == NULL) {
    return;
  }

  if (arvore->Pt == Interna) {
    while (i <= arvore->IntOrExt.Interna.quantItensInt) {
      imprimeBStar(arvore->IntOrExt.Interna.ponteiro[i]);
      if (i != arvore->IntOrExt.Interna.quantItensInt) {
        //cout << arvore->IntOrExt.Interna.chave[i] << endl;
      }
      i++;
    }
  }
  else {
    aux = arvore;
    for (int j = 0; j < aux->IntOrExt.Externa.quantItensExt; j++) {
      cout << aux->IntOrExt.Externa.registro[j].chave << endl;
    }
  }
}

bool pesquisaArvoreBStar(TipoRegistro *registro, Apontador apontador,
int& numero_comparacoes) {
  int i;
  if (apontador->Pt == Interna) {
    i = 1;

    //PESQUISA SEQUENCIAL NA PÁGINA INTERNA
    while (i < apontador->IntOrExt.Interna.quantItensInt &&
registro->chave > apontador->IntOrExt.Interna.chave[i-1]) {
      numero_comparacoes++;
      i++;
    }
```

```
    // ATIVAÇÃO RECURSIVA EM UMA DAS SUBÁRVORES: A PESQUISA SÓ PARA AO
ENCONTRAR UMA PÁGINA FOLHA
    if (registro->chave < apontador->IntOrExt.Interna.chave[i-1]) {
      numero_comparacoes++;
      pesquisaArvoreBStar(registro,
apontador->IntOrExt.Interna.ponteiro[i - 1], numero_comparacoes);
    }
    else
      pesquisaArvoreBStar(registro,
apontador->IntOrExt.Interna.ponteiro[i], numero_comparacoes);
  }

  i = 1;

  // PESQUISA SEQUENCIAL NA PÁGINA FOLHA
  while (i < apontador->IntOrExt.Externa.quantItensExt && registro->chave
> apontador->IntOrExt.Externa.registro[i - 1].chave) {
    numero_comparacoes++;
    i++;
  }
  // VERIFICA SE A CHAVE DESEJADA FOI LOCALIZADA
  if (registro->chave == apontador->IntOrExt.Externa.registro[i -
1].chave) {
    numero_comparacoes++;
    *registro = apontador->IntOrExt.Externa.registro[i - 1];
    return true;
  }
  else if (apontador->IntOrExt.Externa.registro == NULL) {
    cout << "TipoRegistro nao esta presente na arvore" << endl;
    return false;
  }
}
```