SCC0503 - Algoritmos e Estruturas de Dados II - Trabalho I - Árvore B

Guilherme Filipe Feitosa dos Santos NUSP: 11275092
Pedro Fernando Christofoletti dos Santos NUSP: 11218560

Relatório:

Para realizar o trabalho de Árvore B e cumprir os requisitos solicitados organizamos o projeto em 5 arquivos:

- ## Arquivo 1

"Main.c": Arquivo onde rodamos a aplicação, nele estamos criando e validando o arquivo de texto contendo os registros dos estudantes e o índice de árvore de B. Além disso, a main cuida do sistema de menu utilizando de um simples Switch para selecionar a opção desejada e também ao final da execução realiza o fechamento dos arquivos.

Bibliotecas utilizadas:

```
#include <stdio.h>
#include <stdlib.h>
#include "b_tree.h"
```

Criando e validando os arquivos de dados e da árvore:

```
FILE *data_file = fopen("dados.txt","a+b");
FILE *b_tree_index = fopen("index.dat","a+b");
```

```
if (checkFile(data_file))return 0;
if (checkFile(b_tree_index))return 0;
```

Menu para navegação:

```c
do {
  printMenu();
  scanf("%d",&op);
  // system("clear");
  switch(op){
    case 1: { //Create data file and b tree
      b_tree_setup(data_file,b_tree_index);
      break;
    }
    case 2:{ //Insert
      b_tree_insertRegistry(data_file,b_tree_index);
      break;
    }
    case 3:{ //Search by USP number
      b_tree_searchRegistry(data_file, b_tree_index);
      break;
    }
  }
} while(op != 0);
```

Fechando arquivos no final da execução:

```c
closeFile(data_file);
closeFile(b_tree_index);
```

- ## Arquivo 2

"b_tree.h": Arquivo de cabeçalho das funções contidas no arquivo "b_tree.c" , nesse cabeçalho são disponibilizadas apenas as funções que permitimos que o usuário acesse, no caso, a main. Além disso nele estão contidos os #defines e as structs. As funções possuem retornos de inteiros pois esses retornos são os próprios #defines, através desse retorno o usuário pode saber se a função obteve sucesso ou erro e o motivo do erro.

#defines:

```
#define SUCCESS 1
#define INVALID_FILE -1
#define INVALID_B_TREE_INDEX -2
#define RRN_NOT_FOUND -3
#define RRN_EXISTENT -4
#define EMPTY_FILE -5
```

***Além desses, também há os defines para o tamanho das strings dos registros e de ordem da árvore:**

```
#define SIZE 100
#define ORDER 170
```

Struct KEY e Struct B_TREE_NODE:

```
typedef int keyType;

typedef struct {
  keyType NUSP; // 4 bytes
  long RRN; // 8 bytes
} KEY; // TOTAL = 16 bytes (A struct tambem ocupa espaco na memoria)

typedef struct {
  int counter; // 4 bytes
  KEY keys[ORDER-1]; // 16 * 169 = 2704 bytes
  long children[ORDER]; // 8 * 170 = 1360 bytes
} B_TREE_NODE; // TOTAL = 4079 bytes (A struct tambem ocupa espaco na memoria)
```

***A ordem da árvore foi calculada com base no objetivo de chegar o mais próximo possível de uma página de disco de 4kb.**

Struct STUDENT:

```
typedef struct {
  char name[SIZE];
  keyType numUSP;
  char lastname[SIZE];
  char course[SIZE];
  float grade;
} STUDENT;
```

Cabeçalho das funções disponibilizadas para main:

```c
void printMenu();
void closeFile(FILE *);
int checkFile(FILE *);
int b_tree_setup(FILE *, FILE *);
int b_tree_insertRegistry(FILE *,FILE *);
int b_tree_searchRegistry(FILE *, FILE *);
```

- **Arquivo 3**

"b_tree.c": Arquivo onde estão implementadas tanto as funções do arquivo de cabeçalho quanto as funções auxiliares usadas apenas nesse arquivo e que não são disponibilizadas, essas funções auxiliares sempre começam com "__" na frente do nome. As funções internas (__) não requerem tratamentos ou retornos de erro pois são chamadas apenas pelas funções principais nas quais já são checados possíveis conflitos.

Bibliotecas utilizadas:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <time.h>
#include "b_tree.h"
```

Menu:

```c
void printMenu(){
    printf("\n                    MENU\n\n");
    printf("=============================================");
    printf("\n\nType the option number of your choice\n\n");
    printf("1- Create data file and B Tree\n");
    printf("2- Register\n");
    printf("3- Search\n");
    printf("=============================================");
    printf("\n\nType 0 to exit\n\n");
}
```

***A função irá printar o menu na tela e através do switch na main é possível escolher as opções desejadas. Existe uma função que garante que o usuário não registre ou procure antes de ter criado a árvore e o arquivo de dados. Segue a função:**

Checa se já foram criados os arquivos da árvore e de dados e confere se já foram preenchidos:

```c
// Check if b_tree and data file are filled with data
int __checkfilesize(FILE *file, FILE *b_tree_index){
  fseek(file, 0, SEEK_END);
  fseek(b_tree_index, 0, SEEK_END);
  if(ftell(file) == 0 && ftell(b_tree_index) == 0) return 1;
  else return 0;
}
```

***Essa função vai até o final de cada arquivo e confere com o ftell se o ponteiro é igual a 0, se for igual a 0 quer dizer que o arquivo está vazio e a árvore ou arquivo de dados não foram criadas, caso contrário a função retornará 0 e permitirá a inserção ou busca.**

Confere se o arquivo foi aberto e/ou criado com sucesso:

```c
int checkFile(FILE *file){
  if(file == NULL){
    printf("ERROR, cannot open file");
    return 1;
  }
  return 0;
}
```

Cria e popula árvore e arquivo de dados:

```c
int b_tree_setup(FILE *file, FILE *b_tree_index){

  if(!file) return INVALID_FILE;
  if(!b_tree_index) return INVALID_B_TREE_INDEX;
  if(!__checkfilesize(file,b_tree_index)){
    printf("You have alredy set up the environment!\n");
    return SUCCESS;
  }

  STUDENT student;
  long root_RRN;
  int i = 0, total;
  KEY key;

  total = (ORDER / 2) * ORDER + 1;
  printf("Wait while the %d examples are loaded...\n\n", total);

  fclose(b_tree_index);
  b_tree_index = fopen("index.dat","rb+");

  // Inicializa arquivo de indexacao por b_tree
  __b_tree_create(b_tree_index);

  while(i < total){
    strcpy(student.name, __generateName(i));
    strcpy(student.lastname, __generateName(i+1));
    strcpy(student.course, __generateCourse(i));
    student.grade = __generateGrade(i);
    student.numUSP = i;

    __printRegistry(student);

    //Inserting in file
    fseek(file, 0, SEEK_END);
    fwrite(&student,sizeof(STUDENT),1,file);

    //Inserting in B_tree
    key.NUSP = i;
    key.RRN = i;
    fseek(b_tree_index, 0, SEEK_SET);
    fread(&root_RRN, sizeof(long), 1, b_tree_index);
    // printf("--------------->>>>>Inserting %d\n", i);
    __b_tree_insert(b_tree_index, key, root_RRN, NULL, 0);

    i++;
  }

  printf("Build Successful!\n\n");

  return SUCCESS;

}
```

***A cada itereção do while um novo registro é adicionado tanto ao arquivo de dados quanto no arquivo da árvore B, o loop roda (ORDER / 2) * ORDER + 1 vezes a fim de garantir que existam ao menos dois níveis na árvore.**

Para gerar os dados foram utilizadas as seguintes funções:

Gera nomes:

```
char * __generateName(int i){
  srand(time(NULL)+i);
  char consonants[21] = "bcdfghjklmnpqrstvwxyz";
  char vowels[5] = "aeiou";
  char *str = (char*) malloc(SIZE*sizeof(char));
  int pos=0;
  int syllables = (rand() % 3) + 2;
  for (int i=0;i<syllables;i++){
    str[pos] = consonants[(rand() % 21)];
    str[pos+1] = vowels[(rand() % 5)];
    pos+=2;
  }
  str[0] = toupper(str[0]);
  return str;
}
```

***Essa função concatena uma consoante e uma vogal aleatoriamente utilizando da função rand com a seed setada para o clock do sistema somado ao número da iteração do loop a fim de garantir que não se repitam os nomes pois a função pode rodar mais de uma vez no mesmo segundo, ocasionando em uma seed igual, ou seja, o rand daria o mesmo resultado que o anterior. O nome pode variar entre 2 até 5 sílabas.**

Gera cursos:

```
char * __generateCourse(int i){
  char courses[7][100] = {"Bacharelado em Matematica","Licenciatura em Matematica","Bacharelado em Matematica Aplicada e
  Computacao Cientifica","Bacharelado em Ciencias de Computacao","Bacharelado em Sistemas de Informacao","Bacharelado em
  Estatistica e Ciencia da Dados","Engenharia de Computacao"};
  srand(time(NULL)+i);
  int pos = rand() % 7;
  char *str = (char*) malloc(SIZE*sizeof(char));
  strcpy(str,courses[pos]);
  return str;
}
```

***Seguindo o mesmo princípio da aleatoriedade, essa função retorna um dos cursos armazenados em uma matriz de char que foi preenchida manualmente.**

Gera notas:

```
float __generateGrade(int i){
  srand(time(NULL)+i);
  float grade = rand() % 11;
  return grade;
}
```

***Novamente utilizamos a função rand para gerar uma nota de 0 a 10 para ser retornada.**

OBS: Utilizamos a função gera nome tanto para o nome quanto sobrenome. Além disso, não criamos uma função para gerar o número usp pois ele está sendo gerado sequencialmente conforme a árvore e o arquivo de dados está sendo criado.

Printa um aluno:

```
void __printRegistry(STUDENT student){
  printf("\nName: %s\n",student.name);
  printf("Last Name: %s\n",student.lastname);
  printf("NUSP: %d\n",student.numUSP);
  printf("Course: %s\n",student.course);
  printf("Grade: %.2f\n\n",student.grade);
}
```

Função auxiliar interna para comparar dois inteiros:

```
int __compare(int a, int b){
  if (a==b) return 0;
  else if (a<b) return -1;
  else return 1;
}
```

Recebe e insere o registro de um aluno:

```c
// #Insert a student in data file and b_tree_index;
int b_tree_insertRegistry(FILE *file, FILE *b_tree_index){

  if(!file) return INVALID_FILE;
  if(!b_tree_index) return INVALID_B_TREE_INDEX;
  if(__checkfilesize(file,b_tree_index)){
    printf("Please create file and B Tree before register or search\n");
    return EMPTY_FILE;
  }

  STUDENT student;
  //Reading input
  printf("Name: ");scanf(" %[^\n]s",student.name);
  printf("Last Name: ");scanf(" %[^\n]s",student.lastname);
  printf("NUSP: ");scanf("%d",&student.numUSP);
  printf("Course: ");scanf(" %[^\n]s",student.course);
  printf("Grade: ");scanf("%f",&student.grade);

  KEY key;
  key.NUSP = student.numUSP;
  fseek(file, 0, SEEK_END);
  key.RRN = ftell(file) / sizeof(STUDENT);

  //Inserting in file
  fwrite(&student, sizeof(STUDENT), 1, file);

  long root_RRN;
  fseek(b_tree_index, 0, SEEK_SET);
  fread(&root_RRN, sizeof(long), 1, b_tree_index);

  //Inserting in B_tree
  __b_tree_insert(b_tree_index, key, root_RRN, NULL, 0);

  printf("\nSuccessful registration!\n\n");

  return SUCCESS;
}
```

Busca um aluno pelo número NUSP:

```c
//# Search a registry by USP number
int b_tree_searchRegistry(FILE *file, FILE *b_tree_index){

  if(!file) return INVALID_FILE;
  if(!b_tree_index) return INVALID_B_TREE_INDEX;
  if(__checkfilesize(file,b_tree_index)){
    printf("Please create file and B Tree before register or search\n");
    return EMPTY_FILE;
  }

  long root_RRN, byteoffset, RRN;
  STUDENT student;
  keyType nusp;

  printf("NUSP: ");scanf("%d",&nusp);
  printf("Searching...\n");

  fseek(b_tree_index, 0, SEEK_SET);
  fread(&root_RRN, sizeof(long), 1, b_tree_index);

  RRN = __b_tree_search(b_tree_index, root_RRN, nusp);

  if(RRN == RRN_NOT_FOUND){
    printf("NUSP not found\n\n");
  } else {
    byteoffset = RRN * sizeof(STUDENT);
    fseek(file, byteoffset, SEEK_SET);
    fread(&student, sizeof(student), 1, file);
    __printRegistry(student);
  }

  return SUCCESS;
}
```

***O RRN é descoberto utilizando de uma função interna. Após descoberto posicionamos o ponteiro do arquivo de dados no lugar desejado, o registro é lido e em seguida printado na tela.**

Função interna que descobre o RRN:

```c
int __b_tree_search(FILE *b_tree_index, long RRN, int key){

  if (RRN == -1) return RRN_NOT_FOUND;
  else{
    int i = 0;
    B_TREE_NODE *node = (B_TREE_NODE *) malloc(sizeof(B_TREE_NODE));
    fseek(b_tree_index, ((sizeof(B_TREE_NODE) * RRN) + sizeof(long)), SEEK_SET);
    fread(node, sizeof(B_TREE_NODE), 1, b_tree_index);

    while(i<node->counter){
      int cmp = __compare(node->keys[i].NUSP, key);
      if(!cmp){
        return node->keys[i].RRN;
      } else if(cmp<0){ // key > node->keys->nusp
        i++;
      } else { // key < node->keys->nusp
        RRN = node->children[i];
        return __b_tree_search(b_tree_index, RRN, key);
      }
    }
    RRN = node->children[i];
    return __b_tree_search(b_tree_index, RRN, key);
  }
}
```

***Essa função começa lendo o primeiro nó da árvore inteiro (o RRN do primeiro nó foi disponibilizado através da leitura do cabeçalho de arquivo da árvore na função principal de busca). Após ler o primeiro nó, ela procura dentre todas as chaves do nó se alguma é igual ou maior que a chave passada pelo usuário, caso encontre alguma maior isso nos diz que aquela chave não está nesse nó e um novo nó é lido a partir do descendente da esquerda da primeira chave maior e o processo começa novamente, caso encontre uma chave igual é retornado o seu RRN, caso não encontre é porque chegou em algum nó folha com descendente -1.**

Função interna que esvazia nó a partir de uma posição:

Esta função usa seu parâmetro pos para inserir o sinal de -1 como forma de indicação de espaço vazio dentro de um node. Isso auxilia as outras funções a perceberem os espaços que estão vazios.

```c
// Insere -1 nas posicoes a partir de pos
int __b_tree_empty_node_beginning_in_pos(B_TREE_NODE *node, int pos){
  if(!node) return INVALID_NODE;
  for(int i = pos; i < ORDER - 1; i++){
    node->keys[i].NUSP = -1;
    node->keys[i].RRN = -1;
    node->children[i+1] = -1;
  }
  if(pos == 0) node->children[0] = -1;
  return SUCCESS;
}
```

Função auxiliar que insere uma key em uma posição do nó:

Diferente da função apresentada mais à frente que busca o nó folha onde a key deva ser inserida, esta função tem por objetivo encaixar a key exatamente na posição especificada. Para isso, a função verifica se o nó especificado está vazio, com espaço livro ou lotado, sendo neste último caso necessário realizar a manobra do split.

```c
// Insere key em um nó
int __b_tree_put_key_in_node(FILE *b_tree_index, B_TREE_NODE *node,
        long node_RRN, KEY *key, long RRN, int pos, KEY **over_key){
  long over_RRN = 0;
  if(node->counter == 0){ // Nó vazio
    node->keys[0].NUSP = key->NUSP;
    node->keys[0].RRN = key->RRN;
    node->children[0] = -1;
    node->counter = 1;

    // Zera espaços inutilizados
    __b_tree_empty_node_beginning_in_pos(node, 1);

    *over_key = NULL; // Nenhuma KEY precisara ser reposicionada
  } else if(node->counter < (ORDER - 1)){ // Nó com espaço
    // Insere em pos
    memcpy(&node->keys[pos+1], &node->keys[pos], (node->counter - pos) * sizeof(KEY));
    memcpy(&node->children[pos+2], &node->children[pos+1], (node->counter - pos) * sizeof(long));
    node->keys[pos].NUSP = key->NUSP;
    node->keys[pos].RRN = key->RRN;
    node->children[pos+1] = RRN;
    node->counter++;

    *over_key = NULL; // Nenhuma KEY precisara ser reposicionada
  } else { // Nó cheio
    // Split
    int new_size = (ORDER - 1) / 2, i = 0, cmp;
    B_TREE_NODE *new_children, *to_insert;
```

```
    // Separa nodes
    new_children = (B_TREE_NODE *) calloc(1, sizeof(B_TREE_NODE));
    memmove(new_children->keys, &node->keys[ORDER - 1 - new_size], new_size * sizeof(KEY));
    memmove(new_children->children, &node->children[ORDER - 1 - new_size], (new_size + 1) * sizeof(long));
    node->children[ORDER - 1 - new_size] = -1;

    // Zera espaços inutilizados
    __b_tree_empty_node_beginning_in_pos(node, ORDER - 1 - new_size);
    __b_tree_empty_node_beginning_in_pos(new_children, new_size);

    // Reajusta tamanhos
    new_children->counter = new_size;
    node->counter -= new_size;

    // Verifica de onde vai promover
    if(pos < node->counter) to_insert = node;
    else {
      pos -= node->counter;
      to_insert = new_children;
    }

    // Insere no local correto para descobrir quem sera promovido
    memcpy(&to_insert->keys[pos+1], &to_insert->keys[pos],
      (to_insert->counter - pos) * sizeof(KEY));
    memcpy(&to_insert->children[pos+2], &to_insert->children[pos+1],
      (to_insert->counter - pos) * sizeof(long));
    to_insert->keys[pos].NUSP = key->NUSP;
    to_insert->keys[pos].RRN = key->RRN;
    to_insert->children[pos+1] = RRN;
    to_insert->counter++;
```

```
    // Insere no local correto para descobrir quem sera promovido
    memcpy(&to_insert->keys[pos+1], &to_insert->keys[pos],
      (to_insert->counter - pos) * sizeof(KEY));
    memcpy(&to_insert->children[pos+2], &to_insert->children[pos+1],
      (to_insert->counter - pos) * sizeof(long));
    to_insert->keys[pos].NUSP = key->NUSP;
    to_insert->keys[pos].RRN = key->RRN;
    to_insert->children[pos+1] = RRN;
    to_insert->counter++;

    // Define o promovido e reajusta o tamanho do no
    *over_key = (KEY*) calloc(1, sizeof(KEY)); // Uma KEY precisara ser reposicionada
    fseek(b_tree_index, 0, SEEK_END);
    over_RRN = (ftell(b_tree_index) - sizeof(long)) / sizeof(B_TREE_NODE);
    if(to_insert == node){
      memmove(*over_key, &node->keys[node->counter - 1], sizeof(KEY));
      node->keys[node->counter - 1].NUSP = -1;
      node->keys[node->counter - 1].RRN = -1;
      node->children[node->counter] = -1;
    }else{
      memmove(*over_key, &new_children->keys[0], sizeof(KEY));
      memmove(&new_children->keys[0], &new_children->keys[1],  (new_children->counter - 1) * sizeof(KEY));
      memmove(&new_children->children[0], &new_children->children[1],  (new_children->counter) * sizeof(KEY));
    }
    to_insert->counter--;
    to_insert->keys[to_insert->counter].NUSP = -1;
    to_insert->keys[to_insert->counter].RRN = -1;
    to_insert->children[to_insert->counter + 1] = -1;

    // Atualiza arquivo de indexacao
    fwrite(new_children, sizeof(B_TREE_NODE), 1, b_tree_index);
  }
  fseek(b_tree_index, ((sizeof(B_TREE_NODE) * node_RRN) + sizeof(long)), SEEK_SET);
  fwrite(node, sizeof(B_TREE_NODE), 1, b_tree_index);
  return over_RRN;
}
```

Função interna que cria novo nó e o define como root:

A seguinte função tem por objetivo facilitar a criação e definição no arquivo de um novo nó raiz. Isso pois essa ocorrência pode ser considerada como um caso à parte que deve ser tratado de uma maneira específica.

```c
int __b_tree_new_root_node(FILE *b_tree_index, long left_RRN, long right_RRN, KEY *over_key){
  B_TREE_NODE *new_root;
  long root_RRN;

  // Cria novo no raiz
  new_root = (B_TREE_NODE *) calloc(1, sizeof(B_TREE_NODE));

  // Preenche novo no raiz
  new_root->keys[0].NUSP = over_key->NUSP;
  new_root->keys[0].RRN = over_key->RRN;
  new_root->children[0] = left_RRN;
  new_root->children[1] = right_RRN;
  new_root->counter = 1;
  __b_tree_empty_node_beginning_in_pos(new_root, 1);

  // Insere novo no raiz
  fseek(b_tree_index, 0, SEEK_END);
  root_RRN = (ftell(b_tree_index) - sizeof(long)) / sizeof(B_TREE_NODE);
  fwrite(new_root, sizeof(B_TREE_NODE), 1, b_tree_index);

  // Atualiza RRN do no raiz no arquivo
  fseek(b_tree_index, 0, SEEK_SET);
  fwrite(&root_RRN, sizeof(long), 1, b_tree_index);

  return SUCCESS;
}
```

Função interna que libera espaço alocado para over_key auxiliar:

```c
void __b_tree_free_over_key(KEY **over_key){
  free(over_key);
  free(*over_key);
}
```

Função interna que descobre o RRN:

Função recursiva que percorre um nó verificando a posição relativa onde uma nova chave deve ser inserida. Possui algumas verificações auxiliares que a permitem identificar os possíveis cenários necessários para uma inserção, a citar quando uma chave é “promovida” dentro de um nó raiz que ficou cheio. Além disso, possibilita ao programa aproveitar a referência de posição encontrada para evitar processamentos e buscas repetidas.

```c
int __b_tree_insert(FILE *b_tree_index, KEY key, long node_RRN, KEY **over_key, short level){
  if(over_key == NULL) { // Inicializa over_key auxiliar para primeira chamada da funcao recursiva
    over_key = (KEY **) malloc(sizeof(KEY*));
    *over_key = (KEY *) malloc(sizeof(KEY));
  }

  int i = 0;
  long offset = ((sizeof(B_TREE_NODE) * node_RRN) + sizeof(long));
  B_TREE_NODE *node = (B_TREE_NODE *) calloc(1, sizeof(B_TREE_NODE));
  fseek(b_tree_index, offset, SEEK_SET);
  fread(node, sizeof(B_TREE_NODE), 1, b_tree_index);

  while(i<node->counter){
    int cmp = __compare(node->keys[i].NUSP, key.NUSP);
    if(cmp==0) return RRN_EXISTENT;
    else if(cmp<0) i++; // key > node->keys->nusp
    else if(cmp>0) break; // key < node->keys->nusp
  }

  long over_RRN = -1;
  if(node->counter == 0 || node->children[i] == -1){ // Nó folha
    over_RRN = __b_tree_put_key_in_node(b_tree_index, node, node_RRN, &key, -1, i, over_key);
    if(over_RRN){ // Realizou split e promoveu alguma chave
      if(level == 0){ // Nó raiz
        __b_tree_new_root_node(b_tree_index, node_RRN, over_RRN, *over_key);
        if(level == 0) __b_tree_free_over_key(over_key);
        return 0;
      }
      if(level == 0) __b_tree_free_over_key(over_key);
      return over_RRN;
    }
    if(level == 0) __b_tree_free_over_key(over_key);
    return 0;
  } else { // Nó "galho"
    over_RRN = __b_tree_insert(b_tree_index, key, node->children[i], over_key, level+1);
    if(over_RRN){ // Verifica se houve split no nó seguinte
      memcpy(&key, *over_key, sizeof(KEY));
      over_RRN = __b_tree_put_key_in_node(b_tree_index, node, node_RRN, &key, over_RRN, i, over_key);
      if(over_RRN && level == 0){ // Nó raiz
        __b_tree_new_root_node(b_tree_index, node_RRN, over_RRN, *over_key);
        if(level == 0) __b_tree_free_over_key(over_key);
        return 0;
      }
      if(level == 0) __b_tree_free_over_key(over_key);
      return over_RRN;
    }
    if(level == 0) __b_tree_free_over_key(over_key);
    return 0;
  }
}
```

- # Arquivo 4

“dados.txt”: Arquivo onde são armazenados os registros dos alunos, sempre um novo registro é adicionado ao final do arquivo e todos com tamanho fixo. O arquivo foi escrito de forma binária a fim de facilitar a implementação.

- # Arquivo 5

“index.dat”: Arquivo onde é armazenada a Árvore B contendo cada nó uma página de disco de tamanho aproximado padrão 4Kb de memória. O arquivo foi escrito de forma binária a fim de facilitar a implementação.