%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%

```
/* bt.h
  header file for btree programs
 */

#define MAXKEYS 4  //ordem ímpar
#define MINKEYS MAXKEYS/2
#define NIL    (-1)
#define NOKEY   '@'
#define NO      0
#define YES     1

typedef struct {
      short keycount;        // number of keys in page
      char  key[MAXKEYS];     // the actual keys
      short child[MAXKEYS+1];  // ptrs to rrns of descendants
} BTPAGE;

#define PAGESIZE sizeof(btpage)

extern short root;           // rrn of root page
extern int btfd;            // file descriptor of btree file
extern int infd;            // file descriptor of input file

/* prototypes */
btclose ();
btopen ();
btread (short rrn, BTPAGE *page_ptr);
btwrite (short rrn, BTPAGE *page_ptr);
create_root (char key, short left, short right);
short create_tree();
short getpage ();
short getroot ();
insert (short rrn, char key, short *promo_r_child, char *promo_key);
ins_in_page (char key,short r_child, BTPAGE *p_page);
pageinit (BTPAGE *p_page);
putroot(short root);
search_node (char key, BTPAGE *p_page, short *pos);
split(char key, short r_child, BTPAGE *p_oldpage, char *promo_key, short *promo_r_child, BTPAGE *p_newpage);
```

%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%

```
/* driver.c
  Driver for btree tests
       Open or creates b-tree file.
       Get next key and calls insert to insert key in tree.
       If necessary creates new root.
 */

#include <stdio.h>
#include "bt.h"

int main()
{
```

```
    int promoted;      // boolean: tells if a promotion from below
    short root,        // rrn of root page
        promo_rrn;   // rrn promoted from below
    char promo_key,    // key promoted from below
        key;         // next key to insert in tree

    if (btopen())
    {
      root = getroot();
    }
    else
    {
      root = create_tree();
    }
    while ((key = getchar()) != 'q')
    {
      promoted = insert(root, key, &promo_rrn, &promo_key);
      if (promoted)
        root = create_root(promo_key, root, promo_rrn);
    }
    btclose();
}
```

%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%

```
/* insert.c
  Contains insert() function to insert a key into a btree.
  Calls itself recursively until bottom of tree is reached.
  Then insert key node.
  If node is out of room,
    - calls split() to split node
    - promotes middle key and rrn of new node
 */

#include "bt.h"

insert (short rrn, char key, short *promo_r_child, char *promo_key)
{
     BTPAGE page,        // current page
          newpage;      // new page created if split occurs
     int found, promoted;  // boolean values
     short pos,
          p_b_rrn;       // rrn promoted from below
     char p_b_key;        // key promoted from below

     if (rrn == NIL)
     {
          *promo_key = key;
          *promo_r_child = NIL;
          return(YES);
     }
     btread(rrn, &page);
     found = search_node ( key, &page, &pos);
     if (found)
     {
          printf ("Error: attempt to insert duplicate key: %c \n\007", key);
```

```
            return(0);
        }
        promoted = insert(page.child[pos], key, &p_b_rrn, &p_b_key);
        if (!promoted)
        {
            return(NO);
        }
        if(page.keycount < MAXKEYS)
        {
            ins_in_page(p_b_key, p_b_rrn, &page);
            btwrite(rrn, &page);
            return(NO);
        }
        else
        {
            split(p_b_key, p_b_rrn, &page, promo_key, promo_r_child, &newpage);
            btwrite(rrn, &page);
            btwrite(*promo_r_child, &newpage);
            return(YES);
        }
}
```

%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%

```
/* btio.c
   Contains  btree functions that directly involve file I/O:
 */

 #include <stdio.h>
 #include "bt.h"
 #include "fileio.h"

 int btfd;        // global file descriptor for "btree.dat"

btopen()
{
     btfd = open("btree.dat", O_RDWR);  //trocar por fopen()
     return (btfd > 0);
}

btclose()
{
     close(btfd);
}

short getroot()
{
    short root;
    long lseek();

    lseek(btfd, 0L, 0);      //trocar por fseek()
    if (read(btfd, &root, 2) == 0) //trocar por fread()
    {
        printf("Error: Unable to get root. \007\n");
        exit(1);
    }
    return (root);
```

```
}

putroot(short root)
{
    lseek(btfd, 0L, 0); //trocar por fseek()
    write(btfd, &root, 2);  //trocar por fwrite()
}

short create_tree()
{
    char key;

    btfd = creat("btree.dat",PMODE);
    close (btfd);
    btopen();
    key = getchar();
    return (create_root(key, NIL, NIL));
}

short getpage() //checar quando da primeira página!!!
{
    long lseek(), addr;
    addr = lseek(btfd, 0L, 2) - 2L;   //trocar por fseek(); 2L (cabeçalho)!
    return ((short) addr / PAGESIZE);
}

btread (short rrn, BTPAGE *page_ptr)
{
     long lseek(), add;

     addr  = (long)rrn * (long)PAGESIZE + 2L; 2L (cabeçalho)!
     lseek(btfd, addr, 0);  //trocar por fseek()
     return(read(btfd, page_ptr, PAGESIZE)); //trocar por fread()
}

btwrite(short rrn, BTPAGE *page_ptr)
{
     long lseek(), addr;

     addr = (long)rrn * (long)PAGESIZE +2L; 2L (cabeçalho)!
     lseek(btfd, addr, 0);  //trocar por fseek()
     return(write(btfd, page_ptr, PAGESIZE)); //trocar por fwrite()
}

%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%
/* btutil.c
  Contains utility function for btree program

 */

#include "bt.h"

create_root(char key, short left, short right)
{
     BTPAGE page;
```

```
    short rrn;
    rrn = getpage();
    pageinit (&page);
    page.key[0] = key;
    page.child[0] = left;
    page.child[1] = right;
    page.keycount = 1;
    btwrite(rrn, &page);
    putroot(rrn);
    return(rrn);
}

pageinit(BTPAGE *p_page)
{
    int j;
    for (j = 0; j < MAXKEYS; j++){
       p_page->key[j] = NOKEY;
       p_page->child[j] = NIL;
    }
    p_page->child[MAXKEYS] = NIL;
}

search_node(char key, BTPAGE *p_page, short *pos)
{
    int i;

    for (i = 0; i < p_page->keycount && key > p_page->key[i]; i++);

    *pos = i;

    if (*pos < p_page->keycount && key == p_page->key[*pos])
       {
          return(YES);
       }
    else
       {
          return(NO);
       }
}

ins_in_page(char key,short r_child, BTPAGE *p_page)
{
    int j;
    for(j = p_page-> keycount; key < p_page->key[j-1] && j > 0; j--){
        p_page->key[j] = p_page->key[j-1];
        p_page->child[j+1] = p_page->child[j];
    }
    p_page->keycount++;
    p_page->key[j] = key;
    p_page->child[j+1] = r_child;
}
    somente para ordem ímpar!!
split(char key, short r_child, BTPAGE *p_oldpage, char *promo_key, short *promo_r_child, BTPAGE *p_newpage)
{
    int j;
    short mid;
    char workkeys[MAXKEYS+1];
```

```
    short workchil[MAXKEYS+2];

    for (j = 0; j < MAXKEYS; j++){
       workkeys[j] = p_oldpage->key[j];
       workchil[j] = p_oldpage->child[j];
    }
    workchil[j] = p_oldpage->child[j];
    for (j = MAXKEYS; key < workkeys[j-1] && j > 0; j--){
       workkeys[j] = workkeys[j-1];
       workchil[j+1] = workchil[j];
    }
    workkeys[j] = key;
    workchil[j+1] = r_child;

    *promo_r_child = getpage();
    pageinit(p_newpage);
    for (j = 0; j < MINKEYS; j++){
       p_oldpage->key[j] = workkeys[j];
       p_oldpage->child[j] = workchil[j];
       p_newpage->key[j] = workkeys[j+1+MINKEYS];
       p_newpage->child[j] = workchil[j+1+MINKEYS];
       p_oldpage->key[j+MINKEYS] = NOKEY;
       p_oldpage->child[j+1+MINKEYS] = NIL;
    }
    p_oldpage->child[MINKEYS] = workchil[MINKEYS];
    p_newpage->child[MINKEYS] = workchil[j+1+MINKEYS];
    p_newpage->keycount = MAXKEYS - MINKEYS;
    p_oldpage->keycount = MINKEYS;
    *promo_key = workkeys[MINKEYS];
}
```