described in papers by Ouskel and Scheuermann (1981) and Robinson (1981). Other approaches to secondary indexing include the use of *tries* and *grid files*. Tries are covered in many texts on files and data structures, including Knuth (1973b) and Loomis (1983). Grid files are covered thoroughly in Nievergelt et al. (1984).

An interesting early paper on the use of dynamic tree structures for processing files is "The Use of Tree Structures for Processing Files," by Sussenguth (1963). Wagner (1973) and Keehn and Lacy (1974) examine the index design considerations that led to the development of VSAM. VSAM uses an index structure very similar to a B-tree, but appears to have been developed independently of Bayer and McCreight's work. Readers interested in learning more about AVL trees will find a good, approachable discussion of the algorithms associated with these trees in Standish (1980). Knuth (1973b) takes a more rigorous, mathematical look at AVL tree operations and properties.

C Programs to Insert Keys into a B-Tree

The C program that follows implements the insert program described in the text. The only difference between this program and the one in the text is that this program builds a B-tree of order five, whereas the one in the text builds a B-tree of order four. Input characters are taken from standard I/O, with *q* indicating end of data.

The program requires the use of functions from several files:

driver.c	Contains the main program, which parallels the driver program described in the text very closely.
insert.c	Contains *insert()*, the recursive function that finds the proper place for a key, inserts it, and supervises splitting and promotions.
btio.c	Contains all support functions that directly perform I/O. The header files *fileio.h* and *stdio.h* must be available for inclusion in *btio.c*.
btutil.c	Contains the rest of the support functions, including the function *split()* described in the text.

All the programs include the header file called *bt.h*.

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

#define  MAXKEYS  4
#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;
```

(continued)

```
#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.c...
     Driver for btree tests:

          Opens or creates b-tree file.
          Gets next key and calls insert to insert key in tree.
          If necessary, creates a new root.
*/
#include <stdio.h>
#include "bt.h"

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())         /* try to open btree.dat and get root */
        root = getroot();
    else                  /* if btree.dat not there, create it  */
        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

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

/* insert() ...
Arguments:
      rrn:            rrn of page to make insertion in
      *promo_r_child: child promoted up from here to next level
      key:            key to be inserted here or lower
      *promo_key:     key promoted up from here to next level
*/
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) {          /* past bottom of tree... "promote"*/
        *promo_key = key;      /* original key so that it will be */
        *promo_r_child = NIL;/* inserted at leaf level            */
        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);
    }
```

(continued)

```
    promoted = insert(page.child[pos], key, &p_b_rrn, &p_b_key);
    if (!promoted)
      return (NO);  /* no promotion */
    if (page.keycount < MAXKEYS) {
      ins_in_page(p_b_key, p_b_rrn, &page);/* OK to insert key and */
      btwrite(rrn, &page);                 /* pointer in this page. */
      return (NO);  /* no promotion */
    }
    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);   /* promotion */
    }
}
```

Btio.c

```
/* btio.c...
    Contains btree functions that directly involve file i/o:

    btopen() -- open file "btree.dat" to hold the btree.
    btclose() -- close "btree.dat"
    getroot() -- get rrn of root node from first two bytes of btree.dat
    putroot() -- put rrn of root node in first two bytes of btree.dat
    create_tree() -- create "btree.dat" and root node
    getpage() -- get next available block in "btree.dat" for a new page
    btread() -- read page number rrn from "btree.dat"
    btwrite() -- write page number rrn to "btree.dat"
*/
#include "stdio.h"
#include "bt.h"
#include "fileio.h"

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

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

btclose()
{
    close(btfd);
}

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

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

putroot(short root)
{
      lseek(btfd, 0L, 0);
      write(btfd, &root, 2);
}

short create_tree()
{
    char key;

    btfd = creat("btree.dat",PMODE);
    close(btfd);        /* Have to close and reopen to insure  */
    btopen();           /* read/write access on many systems.  */
    key = getchar();    /* Get first key.  */
    return (create_root(key, NIL, NIL));
}

short getpage()
{
    long lseek(), addr;
    addr = lseek(btfd, 0L, 2) - 2L;
    return ((short) addr / PAGESIZE);
}

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

      addr = (long)rrn * (long)PAGESIZE + 2L;
      lseek(btfd, addr, 0);
      return ( read(btfd, page_ptr, PAGESIZE) );
}

btwrite(short rrn, BTPAGE *page_ptr)
{
    long lseek(), addr;
    addr = (long) rrn * (long) PAGESIZE + 2L;
    lseek(btfd, addr, 0);
    return (write(btfd, page_ptr, PAGESIZE));
}
```

Btutil.c

```
/* btutil.c...
      Contains utility functions for btree program:

      create_root() -- get and initialize root node and insert one key
      pageinit() -- put NOKEY in all "key" slots and NIL in "child" slots
      search_node() -- return YES if key in node, else NO. In either case,
                  put key's correct position in pos.
      ins_in_page() -- insert key and right child in page
      split() -- split node by creating new node and moving half of keys to
                  new node. Promote middle key and rrn of new node.
 */
#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)   /* p_page: pointer to a 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)
                /* pos: position where key is or should be inserted  */
{
    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);  /* key is in page */
    else
        return (NO);   /* key is not in page */
}

ins_in_page(char key, short r_child, BTPAGE *p_page)
{
    int i;
    for (i = p_page->keycount; key < p_page->key[i-1] && i > 0; i--) {
        p_page->key[i] = p_page->key[i-1];
        p_page->child[i+1] = p_page->child[i];
    }
    p_page->keycount++;
    p_page->key[i] = key;
    p_page->child[i+1] = r_child;
}

/* split ()
Arguments:
      key:           key to be inserted
      promo_key:     key to be promoted up from here
      r_child:       child rrn to be inserted
      promo_r_child: rrn to be promoted up from here
      p_oldpage:     pointer to old page structure
      p_newpage:     pointer to new page structure
*/
split(char key, short r_child, BTPAGE *p_oldpage, char *promo_key,
                                short *promo_r_child, BTPAGE *p_newpage)
{
    int   i;
    short mid;                  /* tells where split is to occur          */
    char  workkeys[MAXKEYS+1];/* temporarily holds keys, before split    */
    short workch[MAXKEYS+2];  /* temporarily holds children, before split*/

    for (i=0; i < MAXKEYS; i++) {       /* move keys and children from  */
      workkeys[i] = p_oldpage->key[i];  /* old page into work arrays    */
      workch[i] = p_oldpage->child[i];
    }
    workch[i] = p_oldpage->child[i];
    for (i=MAXKEYS; key < workkeys[i-1] && i > 0; i--){/* insert new key*/
        workkeys[i] = workkeys[i-1];
        workch[i+1] = workch[i];
    }
    workkeys[i] = key;
    workch[i+1] = r_child;

*promo_r_child = getpage();          /* create new page for split,   */
pageinit(p_newpage);                 /* and promote rrn of new page  */
```

(continued)

```
    for (i = 0; i < MINKEYS; i++) {   /* move first half of keys and  */
      p_oldpage->key[i] = workkeys[i];  /* children to old page, second */
      p_oldpage->child[i] = workch[i];  /*  half to new page            */
      p_newpage->key[i] = workkeys[i+1+MINKEYS];
      p_newpage->child[i] = workch[i+1+MINKEYS];
      p_oldpage->key[i+MINKEYS] = NOKEY;     /* mark second half of old  */
      p_oldpage->child[i+1+MINKEYS] = NIL;  /* page as empty             */
    }
    p_oldpage->child[MINKEYS] = workch[MINKEYS];
    p_newpage->child[MINKEYS] = workch[i+1+MINKEYS];
    p_newpage->keycount = MAXKEYS - MINKEYS;
    p_oldpage->keycount = MINKEYS;
    *promo_key = workkeys[MINKEYS];          /* promote middle key */
}
```


Pascal Programs to Insert Keys into a B-Tree

The Pascal program that follows implements the insert program described in the text. The only difference between this program and the one in the text is that this program builds a B-tree of order five, whereas the one in the text builds a B-tree of order four. Input characters are taken from standard I/O, with q indicating end of data.

The main program includes three nonstandard compiler directives:

```
{$B-}
{$I btutil.prc}
{$I insert.prc}
```

The $B− instructs the Turbo Pascal compiler to handle keyboard input as a standard Pascal file.

The $I directives instruct the compiler to include the files *btutil.prc* and *insert.prc* in the main program. These two files contain functions needed by the main program. So the B-tree program requires the use of functions from three files:

driver.pas	Contains the main program, which closely parallels the driver program described in the text.
insert.prc	Contains *insert()*, the recursive function that finds the proper place for a key, inserts it, and supervises splitting and promotions.
btutil.prc	Contains all other support functions, including the function *split()* described in the text.

Driver.pas

```
PROGRAM btree (INPUT,OUTPUT);
{
        Driver for B-tree tests:
```

(continued)