**REPORT DOCUMENTATION PAGE**

<table>
<thead>
<tr>
<th>1. REPORT NUMBER</th>
<th>2. GOVT ACCESSION NO.</th>
<th>3. RECIPIENT'S CATALOG NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSWC TR 85-54</td>
<td>AD-4/57</td>
<td>026</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4. TITLE (and Subtitle)</th>
<th>5. TYPE OF REPORT &amp; PERIOD COVERED</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTREE: A FORTRAN CODE FOR A B+ TREE</td>
<td>Final: Fiscal Year '85</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6. PERFORMING ORG. REPORT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>64601N; 50267; 0; 5U15DD</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>7. AUTHOR(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elliot Winston</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8. CONTRACT OR GRANT NUMBER(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>9. PERFORMING ORGANIZATION NAME AND ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naval Surface Weapons Center (R44)</td>
</tr>
<tr>
<td>10901 New Hampshire Avenue</td>
</tr>
<tr>
<td>Silver Spring, MD 20903-5000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10. PROGRAM ELEMENT, PROJECT, TASK AREA &amp; WORK UNIT NUMBERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>64601N; 50267; 0; 5U15DD</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>11. CONTROLLING OFFICE NAME AND ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naval Surface Weapons Center (R44)</td>
</tr>
<tr>
<td>10901 New Hampshire Avenue</td>
</tr>
<tr>
<td>Silver Spring, MD 20903-5000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>12. REPORT DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 April 1985</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>13. NUMBER OF PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>14. MONITORING AGENCY NAME &amp; ADDRESS (if different from Controlling Office)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>15. SECURITY CLASS. (of this report)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNCLASSIFIED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>16. DISTRIBUTION STATEMENT (of this Report)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approved for public release, distribution unlimited</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>18. SUPPLEMENTARY NOTES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>19. KEY WORDS (Continue on reverse side if necessary and identify by block number)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B+ Tree, Database Manager, Node, Leaf, Root</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>20. ABSTRACT (Continue on reverse side if necessary and identify by block number)</th>
</tr>
</thead>
<tbody>
<tr>
<td>This report discusses and documents a FORTRAN code for a B+ Tree, a data structure which if frequently used is the foundation of a database manager.</td>
</tr>
</tbody>
</table>
FOREWORD

This report contains documentation for a FORTRAN implementation of a B+ tree, a data structure which is often used as the foundation of a database manager. Because the code is written in a high-level language, it is basically transportable to any computer with FORTRAN capability (minor modification may be required for compatibility with a host computer's operating system and compiler). The work was done as a first step towards developing a user-friendly, interactive database manager needed by U31 to support studies requiring the extensive use of minefield planning codes.

This work has been supported by the Mine Improvement Program at NSWC under Project S0267.

Approved by:

IRA M. BLATSTEIN, Head
Radiation Division

Accession Form

By:

Distributed: A

Ignoble Sciences

Dist.

1
CONTENTS

INTRODUCTION..........................................................1
SPECIFICATIONS............................................................2
USAGE.................................................................2
RECONFIGURATION.....................................................3
REFERENCES............................................................5
APPENDIX A - SUBROUTINE DOCUMENTATION..........................A-1
APPENDIX B - FORTRAN CODE LISTING...............................B-1
DISTRIBUTION...........................................................(1)

ILLUSTRATION

Figure

1  EXAMPLE OF A B+ TREE..............................................1
INTRODUCTION

A B+ tree is a data structure which is particularly well suited for storing the keys which identify the records in a database. The objective of a B+ tree, hereafter referred to simply as a tree, is to minimize the number of mass storage accesses required to find a specified key. A conceptual representation of a tree, containing the letters \{B,D,E,G,H,L,N,R,S\} as keys, is shown in Figure 1.

A tree corresponds to a file, each node of the tree corresponds to a record in that file, and a pointer between nodes corresponds to the record number of the node to which it points. The tree in Figure 1 has a nodal capacity equal to 2. The nodes on the bottom level, called the leaves of the tree, contain all the keys. Each leaf points to its neighbor on the right, and the rightmost leaf points to zero, indicating that it is the last leaf in the tree.

All key searches begin at the top node, known as the root. To find the letter G, for example, the left node on the second level is searched after the root because G precedes L in alphabetical order. Since G is between E and H, the second leaf from the left is searched next, and the key is found. Thus, the number of accesses required to find a key is equal to one more than the height of the tree. As keys are added to the tree, leaves become full and split in half; as keys are removed, adjacent leaves may merge. A complete description of splitting and merging rules will not be given here, but the interested reader is referred to the excellent introduction presented by Comer; a more analytical discussion can be found in Knuth.
Documentation for a database manager based on the code listed in Appendix B is contained in Winston³.

SPECIFICATIONS

The implementation listed in Appendix B is written in a version of FORTRAN 77 for a DEC VAX/780 computer with the VMS operating system. In particular, file names are at most 9 characters long, and have extenders with as many as 3 characters. A tree called (name) can have as many as 3 files associated with it: (name).KEY, (name).NOD, and (name).REC. The file (name).KEY corresponds to the tree itself, and (name).NOD and (name).REC are node and record stacks, respectively. The numbers of the nodes and data records that are deleted from the tree as a result of key deletions are saved by the stacks and reused as needed. Stack files containing no numbers are automatically erased from the system.

Records corresponding to nodes have a length of 256 bytes, and the maximum tree height is equal to 5. Keys have a maximum length of 20 characters, and the tree can accommodate up to 65,535 keys. Application programs which call BTREE can have up to 10 trees open simultaneously, via the logical device unit numbers 1, 2, ..., 10.

USAGE

A call to BTREE is accomplished by the standard FORTRAN syntax

CALL BTREE (LTR,LDU,A,MAXLEN,IREC,IERR).

A description of the input and output parameters follows:

INPUT:

<table>
<thead>
<tr>
<th>LTR (CHARACTER*1)</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>(A)dd a key to the tree</td>
</tr>
<tr>
<td>C</td>
<td>(C)reate a new tree</td>
</tr>
<tr>
<td>D</td>
<td>(D)elete a key from the tree</td>
</tr>
<tr>
<td>F</td>
<td>(F)irst key in the tree</td>
</tr>
<tr>
<td>G</td>
<td>(G)et first occurrence of a partial key</td>
</tr>
<tr>
<td>O</td>
<td>(O)pen an old tree</td>
</tr>
<tr>
<td>S</td>
<td>(S)uccessor key</td>
</tr>
</tbody>
</table>

LDU (BYTE) unit number under which the tree communicates with mass storage; permissible values are 1, 2, ..., 10

A (CHARACTER*20) full key
(required for LTR = 'A', 'D')
partial key
(required for LTR = 'G')
tree name
(required for LTR = 'C', 'O')
MAXLEN (INTEGER*4) maximum key length; cannot exceed 20
(required for LTR = 'C')

OUTPUT:
A (CHARACTER*20) value of the last key accessed
IREC (INTEGER*4) number of the data record associated with the
last key accessed
IERR (BYTE) ERROR CODE

0 --------------- successful execution of the command
1 --------------- illegal value of the parameter 'LTR'
2 --------------- attempt to create an existent tree
3 --------------- attempt to access a nonexistent tree
4 --------------- attempt to find a nonexistent key
5 --------------- no successor key exists
6 --------------- attempt to insert a key currently in tree

The "get" command is of particular interest because it searches the tree
for the first occurrence of a left-justified string within a key. This
feature is useful when it is desired to access a particular class of keys, all
of which begin with the same string of characters. Setting A equal to the
string and executing a "get" command finds the first key which has A as its
prefix; successive "successor" commands find the remaining keys in the
class. The full key value returned by A should always be checked for its
prefix after each such call. This procedure is especially fast because the
"successor" command usually does not require a mass storage access. Unlike
the "get" command, the "delete" command fails to execute unless the full key
value is specified.

Either an "open" command or a "create" command must be executed prior to
performing any other operations on a particular tree.

BTREE creates trees which contain unique keys only; an attempt to insert
a duplicate key will not execute (IERR = 6). Also, the code prevents the
creation of a new tree having the same name as a tree currently on the system
(IERR = 2).

RECONFIGURATION

It may be desirable to alter some of the specification parameters to make
BTREE mesh more efficiently with a particular application program. These
adjustments are indicated as follows:

(a) To change the maximum key length to k characters, declare the
passing parameter A and the internal variable KEYVAL as
CHARACTER*k;

(b) To change the height of the tree to h, declare the arrays BUF(0:h)
and PATH(0:h);
(c) To change the number of trees open simultaneously to $t$, declare
the arrays HTREC($t$), MAXREC($t$), HTNOD($t$), MAXNOD($t$), ROOT($t$),
MKL($t$), HEIGHT($t$), NAME($t$), MARKS($t$), MANYS($t$), and BLOCS($t$);

(d) To change the size of the nodes to $s$ bytes, change all
CHARACTER*256 declarations to CHARACTER*$s$, assign FULL = $(s-4)/\text{ONE}
in SUBROUTINE BTREE, and assign RECL = $s$ in the OPEN statement in
SUBROUTINE NEWTREE.

While it is possible to increase the maximum number of keys allowed in
the tree, the alterations required by the code are far more intricate and
complex. Changing to a 3-BYTE symbol code for integers would allow up to
16,777,215 keys, but would entail, among other things, rewriting the functions
VAL and SYM and checking all sections of the code dealing with loading
information into the nodes because 3 bytes must be reserved for each integer
instead of 2 bytes. In short, increasing the number of keys is not
recommended. Moreover, specially tailored database programs are usually
developed, or purchased, to maintain such large databases.
REFERENCES


APPENDIX A

SUBROUTINE DOCUMENTATION
SUBROUTINE BTREE

PURPOSE: To control the logic needed to execute the command requested by the value of LTR.

INPUTS:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LTR</td>
<td>CHARACTER*1</td>
<td>command letter</td>
</tr>
<tr>
<td>LDU</td>
<td>BYTE</td>
<td>input/output device number</td>
</tr>
<tr>
<td>A</td>
<td>CHARACTER*20</td>
<td>key name if LTR = 'A', 'D', or 'G'</td>
</tr>
<tr>
<td>A</td>
<td>CHARACTER*20</td>
<td>tree name if LTR = 'C' or 'O'</td>
</tr>
<tr>
<td>MAXLEN</td>
<td>INTEGER*4</td>
<td>maximum key length</td>
</tr>
</tbody>
</table>

OUTPUTS:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IREC</td>
<td>INTEGER*4</td>
<td>number of the data record associated with the last key examined</td>
</tr>
<tr>
<td>IERR</td>
<td>BYTE</td>
<td>error code number</td>
</tr>
<tr>
<td>A</td>
<td>CHARACTER*20</td>
<td>full value of last key examined</td>
</tr>
<tr>
<td>ONE</td>
<td>INTEGER*4</td>
<td>parameter equal to MAXLEN + 2</td>
</tr>
<tr>
<td>FULL</td>
<td>INTEGER*4</td>
<td>maximum number of keys in a node</td>
</tr>
</tbody>
</table>

EXTERNALS:

ADDKEY, NEWTREE, DELKEY, FIRST, GETKEY, OLDTREE, SUCCESSOR
SUBROUTINE MERGE

PURPOSE:  To merge two adjacent nodes into a single node.

INPUTS:

LEAF  LOGICAL  .TRUE. if and only if the current node is a leaf
LFM  INTEGER*4  number of keys in the left node
RTM  INTEGER*4  number of keys in the right node
LFBLOC  CHARACTER*256  left node
RTBLOC  CHARACTER*256  right node
MKL  INTEGER*4  maximum key length
KEYVAL  CHARACTER*20  value of the last key examined
RTNOD  INTEGER*4  number of the right node
IO  BYTE  input/output device number
NAME  CHARACTER*9  name of the current tree
HTNOD  INTEGER*4  height of the node stack
MAXNOD  INTEGER*4  largest number yet assigned to a node
ONE  INTEGER*4  parameter equal to MKL + 2

OUTPUTS:

none

EXTERNALS:

STACK
SUBROUTINE ADJACENT

PURPOSE: To find a node adjacent to the current node.

INPUTS:
- BUF CHARACTER*256 array which contains the nodes in the current path
- LEVEL BYTE level of the current node in the tree
- NOD INTEGER*4 number of the current node
- BLOC CHARACTER*256 current node
- MANY INTEGER*4 number of keys in the current node
- ONE INTEGER*4 parameter equal to MKL + 2

OUTPUTS:
- RTNOD INTEGER*4 number of the node to the right of the current node, if it exists; otherwise, the number of the current node
- RTBLOC CHARACTER*256 right node
- RTM INTEGER*4 number of keys in the right node
- LFNOD INTEGER*4 number of the current node, if no right node exists; otherwise, the number of the node to the left of the current node
- LFBLOC CHARACTER*245 left node
- LFM INTEGER*4 number of keys in the left node
- KEYVAL CHARACTER*20 value of the separator key

EXTERNALS:
- VAL
SUBROUTINE DELKEY

PURPOSE:
To delete a key and update the tree.

INPUTS:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IO</td>
<td>BYTE</td>
<td>input/output device number</td>
</tr>
<tr>
<td>NAME</td>
<td>CHARACTER*9</td>
<td>name of the current tree</td>
</tr>
<tr>
<td>HTREC</td>
<td>INTEGER*4</td>
<td>height of the record stack</td>
</tr>
<tr>
<td>MAXREC</td>
<td>INTEGER*4</td>
<td>largest number yet assigned to a data record</td>
</tr>
<tr>
<td>FULL</td>
<td>INTEGER*4</td>
<td>maximum number of keys in a node</td>
</tr>
<tr>
<td>HEIGHT</td>
<td>BYTE</td>
<td>height of the current tree</td>
</tr>
<tr>
<td>ONE</td>
<td>INTEGER*4</td>
<td>parameter equal to MKL + 2</td>
</tr>
</tbody>
</table>

OUTPUTS:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROOT</td>
<td>INTEGER*4</td>
<td>node number of the root of the updated tree</td>
</tr>
<tr>
<td>HEIGHT</td>
<td>BYTE</td>
<td>height of the updated tree</td>
</tr>
<tr>
<td>HTREC</td>
<td>INTEGER*4</td>
<td>height of the updated record stack</td>
</tr>
<tr>
<td>MAXREC</td>
<td>INTEGER*4</td>
<td>largest number yet assigned to a data record</td>
</tr>
<tr>
<td>HTNOD</td>
<td>INTEGER*4</td>
<td>height of the updated node stack</td>
</tr>
<tr>
<td>MAXNOD</td>
<td>INTEGER*4</td>
<td>largest number yet assigned to a node</td>
</tr>
</tbody>
</table>

EXTERNALS:

GETKEY, STACK, ADJACENT, MERGE, PARENT, SHARE
SUBROUTINE SUCCESSOR

PURPOSE: To search for the key following the last key accessed.

INPUTS:

MARK   INTEGER*4  position in the current node of the last key examined
MANY   INTEGER*4  number of keys in the current node
ONE    INTEGER*4  parameter equal to MKL + 2
BLOC   CHARACTER*256  current node

OUTPUTS:

MARK   INTEGER*4  position of the next key in the node containing it
MANY   INTEGER*4  number of keys in the node containing the next key
NOD    INTEGER*4  number of the node containing the next key
PTR    INTEGER*4  number of the data record associated with the next key
KEYVAL CHARACTER*20  value of the next key
ERR    BYTE    error code number

EXTERNALS:

none
SUBROUTINE FIRST

PURPOSE: To search for the first key in the tree.

INPUTS:
1. IO BYTE input/output device number
2. ROOT INTEGER*4 node number of the tree root
3. HEIGHT BYTE height of the current tree

OUTPUTS:
1. MARK INTEGER*4 position in the current node of the last key examined
2. MANY INTEGER*4 number of keys in the current node
3. NOD INTEGER*4 number of the current node
4. PTR INTEGER*4 number of the data record associated with the last key examined
5. KEYVAL CHARACTER*20 value of the last key examined
6. ERR BYTE error code number

EXTERNALS:
1. VAL
**SUBROUTINE NEWROOT**

**PURPOSE:** To create a new root.

**INPUTS:**
- **LFNOD** INTEGER*4 number of left node
- **RTNOD** INTEGER*4 number of right node
- **KEYVAL** CHARACTER*20 value of the first key in the right node
- **IO** BYTE input/output device number
- **NAME** CHARACTER*9 name of the current tree
- **HTNOD** INTEGER*4 height of the node stack
- **MAXNOD** INTEGER*4 largest number yet assigned to a node

**OUTPUTS:**
- **ROOT** INTEGER*4 node number of the tree root
- **HEIGHT** BYTE height of the current tree

**EXTERNALS:**
- **SYM, STACK**
SUBROUTINE PARENT

PURPOSE: To update a parent node.

INPUTS:

LEVEL BYTE level of node last examined
LFNOD INTEGER*4 number of left node
RTNOD INTEGER*4 number of right node
BUF CHARACTER*256 array which contains the nodes constituent the current path
ONE INTEGER*4 parameter equal to MKL + 2
PATH INTEGER*4 array which contains the node numbers defining the path from the root to the current node
INC INTEGER*4 variable which determines the appropriate update action to be taken on the parent node
IO BYTE input/output device number
MKL INTEGER*4 maximum key length
KEYVAL CHARACTER*20 value of the separator key
PTR INTEGER*4 number of the node to which the separator key points

OUTPUTS:

NOD INTEGER*4 number of the parent node
MANY INTEGER*4 updated number of keys in the parent node
BLOC CHARACTER*256 updated parent node

EXTERNALS:

VAL, SYM
SUBROUTINE SPLIT

PURPOSE: To split a full node into two half-full nodes.

INPUTS:
- IO: BYTE input/output device number
- NAME: CHARACTER*9 name of the current tree
- HTNOD: INTEGER*4 height of the node stack
- MAXNOD: INTEGER*4 largest number yet assigned to a node
- FULL: INTEGER*4 maximum number of keys in a node
- ONE: INTEGER*4 parameter equal to MKL + 2
- BLOC: CHARACTER*256 current node
- LEAF: LOGICAL .TRUE. if and only if the current node is a leaf
- NOD: INTEGER*4 number of the current node

OUTPUTS:
- KEYVAL: CHARACTER*20 value of key in middle of current node
- LFNOD: INTEGER*4 number of left node
- RTNOD: INTEGER*4 number of right node

EXTERNALS: 
- SYM
SUBROUTINE ADDKEY

PURPOSE: To insert a key into the tree.

INPUTS:
ONE INTEGER*4 parameter equal to MKL + 2
FULL INTEGER*4 maximum number of keys in a node
IO BYTE input/output device number
NAME CHARACTER*9 name of the current tree
HTREC INTEGER*4 height of the record stack
MAXREC INTEGER*4 largest number yet assigned to a data record

OUTPUTS:
ERR BYTE error code number

EXTERNALS:
GETKEY, STACK, SYM, PARENT, NEWROOT
SUBROUTINE LOOK

PURPOSE: To search a given node for the pointer to the next node in the path (if \( H < \text{HEIGHT} \)), or to search a leaf for the desired key (if \( H = \text{HEIGHT} \)).

INPUTS:
- \( H \) INTEGER*4 tree level of the current node
- \( \text{MANY} \) INTEGER*4 number of keys in the current node
- \( \text{ONE} \) INTEGER*4 parameter equal to MKL + 2
- \( \text{KEYVAL} \) CHARACTER*20 search string
- \( \text{BLOC} \) CHARACTER*256 current node
- \( \text{IO} \) BYTE input/output device number
- \( \text{HEIGHT} \) BYTE height of the current tree

OUTPUTS:
- \( \text{MARK} \) INTEGER*4 position in the current node of the last key examined
- \( \text{PTR} \) INTEGER*4 number of next node in path if \( H < \text{HEIGHT} \); number of the data record associated with the last key examined if \( H = \text{HEIGHT} \)

EXTERNALS:
- none
EXTERNALS:

LOOK
SUBROUTINE GETKEY

PURPOSE:
To search for the first occurrence of the key having its first MATCH characters equal to the value of KEYVAL.

INPUTS:
- ABC CHARACTER*1 command letter
- KEYVAL CHARACTER*20 search string
- IO BYTE input/output device number
- MKL INTEGER*4 maximum key length
- ROOT INTEGER*4 node number of the tree root
- HEIGHT BYTE height of the current tree

OUTPUTS:
- MATCH INTEGER*4 number of characters in the search string
- PTR INTEGER*4 number of the data record associated with the last key examined
- NOD INTEGER*4 number of the current node
- PATH INTEGER*4 array which contains the node numbers defining the path from the root to the current node
- BLOC CHARACTER*256 current node
- BUF CHARACTER*256 array which contains the nodes in the current path
- MANY INTEGER*4 number of keys in the current node
- MARK INTEGER*4 position in the current node of the last key examined
- KEYVAL CHARACTER*20 value of last key examined
- ERR BYTE error code number
SUBROUTINE OLDTREE

PURPOSE: To open an existing tree called NAME and initialize the associated parameters.

INPUTS:
- IO BYTE input/output device number
- NAME CHARACTER*9 name of the current tree

OUTPUTS:
- HTREC INTEGER*4 height of the record stack
- MAXREC INTEGER*4 largest number yet assigned to a data record
- HTNOD INTEGER*4 height of the node stack
- MAXNOD INTEGER*4 largest number yet assigned to a node
- ROOT INTEGER*4 node number of the tree root
- HEIGHT BYTE height of the current tree

EXTERNALS: none
SUBROUTINE NEWTREE

PURPOSE: To create a tree called NAME and initialize the associated parameters.

INPUTS:
  IO BYTE input/output device number
  NAME CHARACTER*9 name of the current tree

OUTPUTS:
  HTREC INTEGER*4 height of the record stack
  MAXREC INTEGER*4 largest number yet assigned to a data record
  HTNOD INTEGER*4 height of the node stack
  MAXNOD INTEGER*4 largest number yet assigned to a node
  ROOT INTEGER*4 node number of the tree root
  HEIGHT BYTE height of the current tree

EXTERNALS:
  none
SUBROUTINE SHARE

PURPOSE: To equally redistribute keys between adjacent right and left nodes.

INPUTS:

LEAF LOGICAL .TRUE. if and only if the current node is a leaf
LFM INTEGER*4 number of keys in the left node
RTM INTEGER*4 number of keys in the right node
LFBLOC CHARACTER*256 left node
KEYVAL CHARACTER*20 value of separator key
RTBLOC CHARACTER*256 right node
RTNOD INTEGER*4 number of the right node
LFNOD INTEGER*4 number of the left node
ONE INTEGER*4 parameter equal to MKL + 2

OUTPUTS:

KEYVAL CHARACTER*20 value of updated separator key

EXTERNALS:

SYM
SUBROUTINE STACK

PURPOSE: To push or pop either the node stack or the record stack, as required.

INPUTS:

- PTR INTEGER*:4 number pushed onto the stack, if ADD = 1
- IO BYTE input/output device number
- NAME CHARACTER*:9 name of the current tree
- EXT CHARACTER*:3 file name extender which determines which stack, node or record, is to be updated
- ADD BYTE push/pop stack indicator
- HT INTEGER*:4 height of the stack
- MOST INTEGER*:4 largest number yet assigned to a node or data record, depending on which stack is to be updated

OUTPUTS:

- HT INTEGER*:4 height of the updated stack
- MOST INTEGER*:4 largest number yet assigned to a node (if EXT = 'NOD') or data record (if EXT = 'REC')
- PTR INTEGER*:4 number popped from stack, if ADD = -1

EXTERNALS:

SYM, VAL
FUNCTION VAL

PURPOSE: To convert a 2-BYTE symbol into an integer between 0 and 65,635.
(See FUNCTION SYM for the inverse function.)

INPUTS:
A CHARACTER*2 2-BYTE symbol

OUTPUTS:
VAL INTEGER*4 integer between 0 and 65,635 corresponding to A

EXTERNALS:
none
FUNCTION SYM

PURPOSE: To convert an integer between 0 and 65,635 into a 2-BYTE symbol. (See FUNCTION VAL for the inverse function.)

INPUTS:
NUM INTEGER*4 integer between 0 and 65,635

OUTPUTS:
SYM CHARACTER*2 2-BYTE symbol corresponding to NUM

EXTERNALS: none
PROGRAM DRIVER

PURPOSE: To directly examine a tree by means of an interactive diagnostic program; especially effective when used in conjunction with a debug utility program.

INPUTS: none

OUTPUTS: none

EXTERNALS: none
APPENDIX B

FORTRAN CODE LISTING
SUBROUTINE BTREE(LTR, LDU, A, MAXLEN, IREC, IERR)

A 'B+ tree' is a data structure which is particularly well
suited for storing the keys which identify the data records
in a database. Data is rapidly retrieved by minimizing the
class number of mass storage accesses. This implementation allows
a maximum of 65,535 keys, each having a maximum length of 20
characters; the tree has a maximum height of 5 and contains
256-byte nodes. Numbers corresponding to deleted nodes or
records are placed in stacks and reused as needed. Application
programs which call BTREE can have as many as 10 trees open
simultaneously.

For a basic introduction to the subject of B+ trees, see
"The Ubiquitous B Tree" by Douglas Comer, Computing Surveys,
11(1979)121-137; a more complete discussion can be found in
"The Art of Computer Programming, Vol.3: Sorting and Searching"

Complete documentation for BTREE is contained in
"BTREE : A FORTRAN Code for a B+ Tree" by Elliot Winston,
NSWC TR 85-54; a code for a database manager based on BTREE
is the subject of "BOSS : A FORTRAN Code for a Relational
Database Manager" by Elliot Winston, NSWC TR 85-56.

INPUTS:

LTR (CHARACTER*1) COMMAND ACTION
A - (A)DD A KEY TO THE TREE
C - (C)REATE A NEW TREE
D - (D)ELETE A KEY FROM THE TREE
F - GET THE (F)IRST KEY IN THE TREE
G - (G)ET FIRST OCCURRENCE OF A TRUNCATED KEY
O - (O)PEN AN OLD TREE
S - GET THE (S)UCCESSOR KEY

LDU (BYTE) UNIT NUMBER UNDER WHICH THE TREE
COMMUNICATES WITH MASS STORAGE

A (CHARACTER*20) KEY VALUE (LTR = 'A', 'D', 'G');
TREE NAME (LTR = 'C', 'O')
MAXLEN (INTEGER*4) MAXIMUM KEY LENGTH (LTR = 'C')

OUTPUTS:

IREC (INTEGER*4) NUMBER OF THE DATA RECORD ASSOCIATED
WITH THE LAST KEY ACCESSED
A (CHARACTER*20) VALUE OF LAST KEY ACCESSED
C IERR (BYTE) ERROR CODE
C
1 - ILLEGAL VALUE OF PARAMETER 'LTR'
2 - ATTEMPT TO CREATE EXISTENT TREE
3 - ATTEMPT TO ACCESS NONEXISTENT TREE
4 - ATTEMPT TO FIND NONEXISTENT KEY
5 - NO SUCCESSOR EXISTS (LAST KEY IN TREE)
6 - ATTEMPT TO INSERT KEY CURRENTLY IN TREE

C IMPLICIT INTEGER*4 (A-Z)

C COMMON /XXXTREE/
C
1 IO, ERR, MARK, MANY, PTR, NOD, LEVEL, MATCH, ONE, FULL, ABC,
2 LFM, RTM, LFNOD, RTNOD, KEYVAL, BLOC, LFBLOC, RTBLOC,
3 BUF(0:5), PATH(0:5), HTREC(10), MAXREC(10), HTNOD(10),
4 MAXNOD(10), ROOT(10), MKL(10), HEIGHT(10), NAME(10)
BYTE IO, ERR, LEVEL, HEIGHT
CHARACTER ABC*1, NAME*9, KEYVAL*20
CHARACTER*256 BUF, BLOC, LFBLOC, RTBLOC
C
C BYTE IERR, LDU
INTEGER MARKS(10), MANYS(10)
CHARACTER LTR*1, A*20, BLOCS(10)*256
C
C -------------------------------------------------------------
C CONTROL LOGICAL FLOW OF A COMMAND
C -------------------------------------------------------------
C
IO = LDU
ABC = LTR
ERR = 0
IF (ABC .NE. 'C' .OR. ABC .NE. 'O') THEN
ONE = MKL(IO) + 2
FULL = 252/ONE
END IF
IF (ABC .EQ. 'A') THEN
KEYVAL = A
CALL ADDKEY
ELSE IF (ABC .EQ. 'C') THEN
NAME(IO) = A
MKL(IO) = MAXLEN
CALL NEWTREE
ELSE IF (ABC .EQ. 'D') THEN
KEYVAL = A
CALL DELKEY
ELSE IF (ABC .EQ. 'F') THEN
CALL FIRST
ELSE IF (ABC .EQ. 'G') THEN
KEYVAL = A
CALL GETKEY
ELSE IF (ABC .EQ. 'O') THEN
NAME(IO) = A
CALL OLDTREE
ELSE IF (ABC.EQ.'S') THEN
  MARK = MARKS(IO)
  MANY = MANYS(IO)
  BLOC = BLOCS(IO)
  CALL SUCCESSOR
ELSE
  ERR = 1
END IF
A = KEYVAL
IREC = PTR
IERR = ERR
IF (ABC.EQ.'G'.OR.ABC.EQ.'F'.OR.ABC.EQ.'S') THEN
  MARKS(IO) = MARK
  MANYS(IO) = MANY
  BLOCS(IO) = BLOC
END IF
RETURN
END
SUBROUTINE NEWTREE

C IMPLICIT INTEGER*4 (A-Z)

C COMMON /XXXTREE/
  1 IO, ERR, MARK, MANY, PTR, NOD, LEVEL, MATCH, ONE, FULL, ABC,
  2 LFM, RTM, LFNOD, RTNOD, KEYVAL, BLOC, LFBLOC, RTBLOC,
  3 BUF(0:5), PATH(0:5), HTREC(10), MAXREC(10), HTNOD(10),
  4 MAXNOD(10), ROOT(10), MKL(10), HEIGHT(10), NAME(10)
BYTE 10, ERR, LEVEL, HEIGHT
CHARACTER ABC*1, NAME*9, KEYVAL*20
CHARACTER*256 BUF, BLOC, LFBLOC, RTBLOC

C CHARACTER SYM*2, FN*13
LOGICAL*1 THERE

C 201 FORMAT(A256)
202 FORMAT(7I5)

C-------------------------------------------------------------
C CREATE A NEW TREE
C-------------------------------------------------------------

CLOSE(UNIT=IO)
FN = NAME(IO)//'.KEY'
INQUIRE(FILE=FN, EXIST=THERE)
IF (THERE) THEN
  ERR = 2
  RETURN
END IF
OPEN(UNIT=IO, FILE=FN, STATUS='NEW', FORM='FORMATTED',
*  ACCESS='DIRECT', RECL=256)
I = 0
J = 0
BLOC(1:2) = SYM(I)
BLOC(3:4) = SYM(J)
WRITE(IO,201,REC=2) BLOC
J = 2
BLOC(3:4) = SYM(J)
WRITE(IO,201,REC=4) BLOC
HTREC(IO) = 0
MAXREC(IO) = 0
HTNOD(IO) = 0
MAXNOD(IO) = 2
ROOT(IO) = 2
HEIGHT(IO) = 0

C ENTRY POINT FOR 'HEADER'

C ENTRY HEADER
WRITE(IO,202,REC=1) HTREC(IO), MAXREC(IO), HTNOD(IO), MAXNOD(IO),
* ROOT(IO), MKL(IO), HEIGHT(IO)
SUBROUTINE OLDTREE
C
IMPLICIT INTEGER*4 (A-Z)
C
COMMON /XXXTREE/
  1 IO, ERR, MARK, MANY, PTR, NOD, LEVEL, MATCH, ONE, FULL, ABC,
  2 LFM, RTM, LFNOD, RTNOD, KEYVAL, BLOC, LFBLOC, RTBLOC,
  3 BUF(0:5), PATH(0:5), HTREC(10), MAXREC(10), HTNOD(10),
  4 MAXNOD(10), ROOT(10), MKL(10), HEIGHT(10), NAME(10)
BYTE IO, ERR, LEVEL, HEIGHT
CHARACTER ABC*1, NAME*9, KEYVAL*20
CHARACTER*256 BUF, BLOC, LFBLOC, RTBLOC
C
CHARACTER*13 FN
LOGICAL*1 THERE
C
301 FORMAT(715)
C
C-------------------------------------------------------------
C OPEN AND INITIALIZE AN OLD TREE
C-------------------------------------------------------------
C
FN = NAME(IO)'/'.KEY'
INQUIRE(FILE=FN, EXIST=THERE)
IF (THERE) THEN
  CLOSE(UNIT=10)
  OPEN(UNIT=10, FILE=FN, STATUS='OLD', FORM='FORMATTED',
      ACCESS='DIRECT')
  READ(10, 301, REC=1) HTREC(10), MAXREC(10), HTNOD(10), MAXNOD(10),
      ROOT(10), MKL(10), HEIGHT(10)
ELSE
  ERR = 3
  RETURN
END IF
RETURN
END
SUBROUTINE GETKEY

C IMPLICIT INTEGER*4 (A-Z)
C
COMMON /XXXTREE/
1 IO, ERR, MARK, MANY, PTR, NOD, LEVEL, MATCH, ONE, FULL, ABC,
2 LFM, RTM, LFNOD, RTNOD, KEYVAL, BLOC, LFBLOC, RTBLOC,
3 BUF(0:5), PATH(0:5), HTREC(10), MAXREC(10), HTNOD(10),
4 MAXNOD(10), ROOT(10), MKL(10), HEIGHT(10), NAME(10)
BYTE IO, ERR, LEVEL, HEIGHT
CHARACTER ABC*1, NAME*9, KEYVAL*20
CHARACTER*256 BUF, BLOC, LFBLOC, RTBLOC

401 FORMAT(A256)
C
C----------------------------------------
C SEARCH FOR FIRST OCCURRENCE OF A KEY HAVING
C FIRST 'MATCH' CHARACTERS EQUAL TO 'KEYVAL'
C----------------------------------------
C
IF (ABC.EQ.'G') THEN
  J = 20
  DO WHILE (KEYVAL(J:J).EQ.' ')
    J = J - 1
  END DO
  MATCH = J
ELSE
  MATCH = MKL(IO)
END IF
IF PTR = ROOT(IO)
  HEND = HEIGHT(IO)
DO 4015 H=0,HEND
  NOD = PTR
  PATH(H) = NOD
  READ(IO,401,REC=NOD) BLOC
  BUF(H) = BLOC
  MANY = VAL(BLOC(1:2))
  IF (MANY.EQ.0) THEN
    C EMPTY TREE
    MARK = 1
    ERR = 4
    RETURN
  ELSE
    CALL LOOK(H)
  END IF
4015 CONTINUE
4020 IF (MATCH.EQ.MKL(IO).OR.MARK.LT.MANY+1) GO TO 4030
C
C EXTENDED SEARCH FOR A STRICTLY TRUNCATED KEY
C
B-8
J = MANY*ONE + 3
NOD = VAL(BLOC(J:J+1))
IF (NOD.EQ.0) GO TO 4030
PATH(HEND) = NOD
READ(IO,401,REC=NOD) BLOC
BUF(HEND) = BLOC
MANY = VAL(BLOC(1:2))
CALL LOOK(HEND)
GO TO 4020

C

4030 K = (MARK-1)*ONE + 3
PTR = VAL(BLOC(K:K+1))
IF (KEYVAL.EQ.BLOC(K+2:K+MATCH+1)) THEN
  ERR = 0
  KEYVAL = BLOC(K+2:K+MKL(IO)+1)
ELSE
  ERR = 4
END IF
RETURN
END
SUBROUTINE SHARE(LEAF)

C
IMPLICIT INTEGER*4 (A-Z)
C
COMMON /XXXTREE/
1 IO, ERR, MARK, MANY, PTR, NOD, LEVEL, MATCH, ONE, FULL, ABC,
2 LFM, RTM, LFNOD, RTNOD, KEYVAL, BLOC, LFBLOC, RTBLOC,
3 BUF(0:5), PATH(0:5), HTREC(10), MAXREC(10), HTNOD(10),
4 MAXNOD(10), ROOT(10), MKL(10), HEIGHT(10), NAME(10)
BYTE IO, ERR, LEVEL, HEIGHT
CHARACTER ABC*1, NAME*9, KEYVAL*20
CHARACTER*256 BUF, BLOC, LFBLOC, RTBLOC
C
CHARACTER*2 SYM
LOGICAL*1 LEAF
C
201 FORMAT(A256)
C
C-------------------------------------------------------------
C BALANCE ADJACENT NODES
C-------------------------------------------------------------
C
MANY = (LFM + RTM)/2
IF (MANY.EQ.LFM) THEN
WRITE(IO,201,REC=NOD) BLOC
RETURN
END IF
BLOC(1:2) = SYM(MANY)
IF (MANY.LT.LFM) THEN
I = 2 + MANY*ONE
IF (LEAF) THEN
BLOC(3:I) = LFBLOC(3:I)
BLOC(I+1:I+2) = SYM(RTNOD)
WRITE(IO,201,REC=LFNOD) BLOC
J = 2 + LFM*ONE
BLOC(3:J-I+2) = LFBLOC(I+1:J)
ELSE
BLOC(3:I+2) = LFBLOC(3:I+2)
WRITE(IO,201,REC=LFNOD) BLOC
J = 4 + LFM*ONE
K = 4 + (LFM - MANY - 1)*ONE
BLOC(3:K) = LFBLOC(I+ONE+1:J)
BLOC(K+1:K+MKL(IO)) = KEYVAL
END IF
K = 3 + (LFM - MANY)*ONE
MANY = LFM + RTM - MANY
BLOC(K:4+MANY*ONE) = RTBLOC(3:4+RTM*ONE)
KEYVAL = LFBLOC(I+3:1+ONE)
ELSE
I = 2 + LFM*ONE
IF (LEAF) THEN
BLOC(3:I) = LFBLOC(3:I)
END IF

C
SUBROUTINE MERGE(LEAF)

IMPLICIT INTEGER*4 (A-Z)

COMMON /XXXTREE/
1 IO, ERR, MARK, MANY, PTR, NOD, LEVEL, MATCH, ONE, FULL, ABC,
2 LFM, RTM, LFNOD, RTNOD, KEYVAL, BLOC, LFBLOC, RTBLOC,
3 BUF(0:5), PATH(0:5), HTREC(10), MAXREC(10), HTNOD(10),
4 MAXNOD(10), ROOT(10), MKL(10), HEIGHT(10), NAME(10)
BYTE IO, ERR, LEVEL, HEIGHT
CHARACTER ABC*1, NAME*9, KEYVAL*20
CHARACTER*256 BUF, BLOC, LFBLOC, RTBLOC

BYTE ADD
CHARACTER SYM*2, EXT*3
LOGICAL*1 LEAF

201 FORMAT(A256)

C------------------------------------------------------------------------
C MERGE ADJACENT NODES INTO THE LEFT NODE
C------------------------------------------------------------------------

IF (LEAF) THEN
  MANY = LFM + RTM
  BLOC(1:2) = SYM(MANY)
  BLOC(3:2+LFM*ONE) = LFBLOC(3:2+LFM*ONE)
  I = 3 + LFM*ONE
ELSE
  MANY = LFM + RTM + 1
  BLOC(1:2) = SYM(MANY)
  BLOC(3:4+LFM*ONE) = LFBLOC(3:4+LFM*ONE)
  I = 5 + LFM*ONE
  BLOC(I:I+MKL(IO)-1) = KEYVAL
  I = I + MKL(IO)
END IF

BLOC(I:I+1+RTM*ONE) = RTBLOC(3:4+RTM*ONE)
WRITE(IO,201,REC=LFNOD) BLOC
ADD = 1
EXT = 'NOD'
CALL STACK(RTNOD, IO, NAME(IO), EXT, ADD, HTNOD(IO), MAXNOD(IO))
RETURN
END
SUBROUTINE ADJACENT

IMPLICIT INTEGER*4 (A-Z)

COMMON /XXXTREE/ 
  1 IO, ERR, MARK, MANY, PTR, NOD, LEVEL, MATCH, ONE, FULL, ABC,
  2 LFM, RTM, LFNOD, RTNOD, KEYVAL, BLOC, LFBLOC, RTBLOC,
  3 BUF(0:5), PATH(0:5), HTREC(10), MAXREC(10), HTNOD(10),
  4 MAXNOD(10), ROOT(10), MKL(10), HEIGHT(10), NAME(10)
BYTE IO, ERR, LEVEL, HEIGHT
CHARACTER ABC*1, NAME*9, KEYVAL*20
CHARACTER*256 BUF, BLOC, LFBLOC, RTBLOC

501 FORMAT (A256)

FIND ADJACENT NODES

MUCH = VAL(BUF(LEVEL-1)(1:2))
DO 5005 M=1, MUCH
   I = 3 + (M-1)*ONE
   IF (VAL(BUF(LEVEL-1)(I:I+1)).EQ.NOD) GO TO 5010
5005 CONTINUE
   MARK = MUCH
   I = 3 + (MARK-1)*ONE
   RTNOD = NOD
   RTBLOC = BLOC
   RTM = MANY
   LFNOD = VAL(BUF(LEVEL-1)(I:I+1))
   READ(IO,501,REC=LFNOD) LFBLOC
   LFM = VAL(LFBLOC(1:2))
   GO TO 5015
5010 MARK = M
   I = 3 + (MARK-1)*ONE
   LFNOD = NOD
   LFBLOC = BLOC
   LFM = MANY
   RTNOD = VAL(BUF(LEVEL-1)(I+ONE:I+1+ONE))
   READ(IO,501,REC=RTNOD) RTBLOC
   RTM = VAL(RTBLOC(1:2))
5015 KEYVAL = BUF(LEVEL-1)(I+2:I+1+MKL(10))
RETURN
END
IF (MANY.EQ.0) THEN
  ROOT(IO) = PATH(1)
  HEIGHT(IO) = HEIGHT(IO) - 1
  WRITE(IO,802,REC=1) HTREC(IO),MAXREC(IO),HTNOD(IO),
       MAXNOD(IO),ROOT(IO),MKL(IO),HEIGHT(IO)
END IF
WRITE(IO,801,REC=NOD) BLOC
RETURN
ELSE
  M = 2
  LEAF = .FALSE.
  GO TO 8005
END IF
ELSE
  CALL SHARE(LEAF)
  IF (MANY.EQ.LFM) RETURN
  INC = 0
  CALL PARENT(INC)
  WRITE(IO,801,REC=NOD) BLOC
  RETURN
END IF
END
SUBROUTINE DELKEY
C IMPLICIT INTEGER*4 (A-Z)
C
COMMON /XXXTREE/
  1 IO, ERR, MARK, MANY, PTR, NOD, LEVEL, MATCH, ONE, FULL, ABC,
  2 LFM, RTM, LFNOD, RTNOD, KEYVAL, BLOC, LFBLOC, RTBLOC,
  3 BUF(0:5), PATH(0:5), HTREC(10), MAXREC(10), HTNOD(10),
  4 MAXNOD(10), ROOT(10), MKL(10), HEIGHT(10), NAME(10)
BYTE IO, ERR, LEVEL, HEIGHT
CHARACTER ABC*1, NAME*9, KEYVAL*20
CHARACTER*256 BUF, BLOC, LFBLOC, RTBLOC
C BYTE ADD
CHARACTER SYM*2, EXT*3
LOGICAL*1 LEAF
C
801 FORMAT(A256)
802 FORMAT(715)
C-------------------------------------------------------------
C DELETE A KEY FROM THE TREE
C-------------------------------------------------------------
C
CALL GETKEY
IF (ERR.NE.0) RETURN
ADD = 1
EXT = 'REC'
CALL STACK(PTR, IO, NAME(IO), EXT, ADD, HTREC(IO), MAXREC(IO))
L = 3 + (MARK-1)*ONE
R = 4 + MANY*ONE
BLOC(L:R) = BLOC(L+ONE:R+ONE)
MANY = MANY - 1
BLOC(1:2) = SYM(MANY)
IF (MANY.GE.FULL/2.OR.HEIGHT(IO).EQ.0) THEN
  WRITE(IO,801,REC=NOD) BLOC
END IF
C UPDATE TREE
C
LEVEL = HEIGHT(IO)
M = 1
LEAF = .TRUE.
8005 CALL ADJACENT
IF (LFM+RTM.LE.FULL-M) THEN
  CALL MERGE(LEAF)
  INC = -1
  CALL PARENT(INC)
在其版本的过程中，END IF
SUBROUTINE SUCCESSOR

IMPLICIT INTEGER*4 (A-Z)

COMMON /XXXTREE/
1    IO, ERR, MARK, MANY, PTR, NOD, LEVEL, MATCH, ONE, FULL, ABC,
2    LFM, RTM, LFNOD, RTNOD, KEYVAL, BLOC, LFBLOC, RTBLOC,
3    BUF(0:5), PATH(0:5), HTREC(10), MAXREC(10), HTNOD(10),
4    MAXNOD(10), ROOT(10), MKL(10), HEIGHT(10), NAME(10)
    BYTE IO, ERR, LEVEL, HEIGHT
    CHARACTER ABC*1, NAME*9, KEYVAL*20
    CHARACTER*256 BUF, BLOC, LFBLOC, RTBLOC

801 FORMAT(A256)

GET THE NEXT KEY IN SEQUENCE FOLLOWING THE LAST
KEY ACCESSED UNDER THE SAME INPUT/OUTPUT NUMBER

IF (MARK.LT.MANY) THEN
    MARK = MARK + 1
    GO TO 8010
ELSE
    I = 3 + MANY*ONE
    NOD = VAL(BLOC(1:I+1))
    IF (NOD.EQ.0) THEN
        ERR = 5
        RETURN
    ELSE
        READ(IO, 801, REC=NOD) BLOC
        MANY = VAL(BLOC(1:2))
        MARK = 1
    END IF
END IF

8010 K = 3 + (MARK-1)*ONE
    PTR = VAL(BLOC(K:K+1))
    KEYVAL = BLOC(K+2:K+MKL(IO)+1)
    RETURN
END
SUBROUTINE FIRST

IMPLICIT INTEGER*4 (A-Z)

COMMON /XXXTREE/  
  1  IO, ERR, MARK, MANY, PTR, NOD, LEVEL, MATCH, ONE, FULL, ABC,  
  2  LFM, RTM, LFNOD, RTNOD, KEYVAL, BLOC, LFBLOC, RTBLOC,  
  3  BUF(0:5), PATH(0:5), HTREC(10), MAXREC(10), HTNOD(10),  
  4  MAXNOD(10), ROOT(10), MKL(10), HEIGHT(10), NAME(10)  
 BYTE IO, ERR, LEVEL, HEIGHT  
 CHARACTER ABC*1, NAME*9, KEYVAL*20  
 CHARACTER*256 BUF, BLOC, LFBLOC, RTBLOC

701 FORMAT(A256)

SEARCH FOR THE FIRST KEY IN THE TREE

NOD = ROOT(IO)  
HEND = HEIGHT(IO)  
DO 7005 H = 0, HEND  
   READ(IO,701,REC=NOD) BLOC  
   IF (H.EQ.HEND) GO TO 7010  
   NOD = VAL(BLOC(3:4))
7005 CONTINUE  
7010 MARK = 1  
MANY = VAL(BLOC(1:2))

EMPTY TREE

IF (MANY.EQ.0) THEN  
   ERR = 4  
   RETURN  
END IF  
PTR = VAL(BLOC(3:4))  
KEYVAL = BLOC(5:4+MKL(IO))  
RETURN  
END
SUBROUTINE NEWROOT

C
IMPLICIT INTEGER*4 (A-Z)

C
COMMON /XXXTREE/
1 IO, ERR, MARK, MANY, PTR, NOD, LEVEL, MATCH, ONE, FULL, ABC,
2 LFM, RTM, LFNOD, RTNOD, KEYVAL, BLOC, LFBLOC, RTBLOC,
3 BUF(0:5), PATH(0:5), HTREC(10), MAXREC(10), HTNOD(10),
4 MAXNOD(10), ROOT(10), MKL(10), HEIGHT(10), NAME(10)
BYTE IO, ERLEVEL, HEIGHT
CHARACTER A8C*1, NAME*9, KEYVAL*20
CHARACTER*256 BUF, BLOC, LFBLOC, RTBLOC

C
BYTE ADD
CHARACTER SYM*2, EXT*3

C
701 FORMAT(A256)
702 FORMAT(715)

C-----------------------------------------------
C CREATE A NEW ROOT
C-----------------------------------------------

MANY = 1
BLOC(1:2) = SYM(MANY)
BLOC(3:4) = SYM(LFNOD)
BLOC(5:2+ONE) = KEYVAL
BLOC(3+ONE:4+ONE) = SYM(RTNOD)
ADD = -1
EXT = 'NOD'
CALL STACK(PTR, IO, NAME(IO), EXT, ADD, HTNOD(IO), MAXNOD(IO))
WRITE(IO, 701, REC=PTR) BLOC
ROOT(IO) = iTR
HEIGHT(IO) = HEIGHT(IO) + 1
 WRITE(IO, 702, REC=1) HTREC(IO), MAXREC(IO), HTNOD(IO), MAXNOD(IO),
 * ROOT(IO), MKL(IO), HEIGHT(IO)
RETURN
END
BLOC(1:2) = SYM(MANY)
RETURN
END
SUBROUTINE PARENT(INC)

C
IMPLICIT INTEGER*4 (A-Z)

C
COMMON /XXXTREE/
1 IO, ERR, MARK, MANY, PTR, NOD, LEVEL, MATCH, ONE, FULL, ABC,
2 LFM, RTM, LFNOD, RTNOD, KEYVAL, BLOC, LFBLOC, RTBLOC,
3 BUF(0:5), PATH(0:5), HTREC(10), MAXREC(10), HTNOD(10),
4 MAXNOD(10), ROOT(10), MKL(10), HEIGHT(10), NAME(10)

BYTE IO, ERR, LEVEL, HEIGHT
CHARACTER ABC*1, NAME*9, KEYVAL*20
CHARACTER*256 BUF, BLOC, LFBLOC, RTBLOC

C
CHARACTER*2 SYM

C-------------------------------------------------------------
C UPDATE A PARENT NODE
C-------------------------------------------------------------

LEVEL = LEVEL - 1
NOD = PATH(LEVEL)
BLOC = BUF(LEVEL)
MANY = VAL(BLOC(1:2))

DO 6005 M=1, MANY
    I = 3 + (M-1)*ONE
    IF(VAL(BLOC(I:I+1)).EQ.LFNOD) GO TO 6010
6005 CONTINUE

M = MANY + 1

L = 5 + (M-1)*ONE
R = 4 + MANY*ONE
IF (INC.EQ.-1) THEN
    DELETE SEPARATOR FROM PARENT NODE
    IF (M.LT.MANY) BLOC(L:R) = BLOC(L+ONE:R+ONE)
    ELSE IF (INC.EQ.0) THEN
        UPDATE VALUE OF SEPARATOR IN PARENT NODE
        BLOC(L:L+MKL(IO)-1) = KEYVAL
    ELSE
        INSERT SEPARATOR INTO PARENT NODE
        IF (M.LE.MANY) BLOC(L+ONE:R+ONE) = BLOC(L:R)
        BLOC(L:L+MKL(IO)-1) = KEYVAL
        BLOC(L+MKL(IO):L+ONE-1) = SYM(RTNOD)
    END IF
MANY = MANY + INC

END IF
SUBROUTINE SPLIT(LEAF)
C
IMPLICIT INTEGER*4 (A-Z)
C
COMMON /XXXTREE/
  1 IO, ERR, MARK, MANY, PTR, NOD, LEVEL, MATCH, ONE, FULL, ABC,
  2 LFM, RTM, LFNOD, RTNOD, KEYVAL, BLOC, LFBLOC, RTBLOC,
  3 BUF(0:5), PATH(0:5), HTREC(10), MAXREC(10), HTNOD(10),
  4 MAXNOD(10), ROOT(10), MKL(10), HEIGHT(10), NAME(10)
BYTE IO, ERR, LEVEL, HEIGHT
CHARACTER ABC*1, NAME*9, KEYVAL*20
CHARACTER*256 BUF, BLOC, LFBLOC, RTBLOC
C
BYTE ADD
CHARACTER SYM*2, EXT*3
LOGICAL*1 LEAF
C
601 FORMAT(A256)
C
--- SPLIT A FULL NODE INTO TWO HALF-FULL NODES ---
C
ADD = -1
EXT = 'NOD'
CALL STACK(PTR, IO, NAME(IO), EXT, ADD, HTNOD(IO), MAXNOD(IO))
MANY = FULL/2
LFNOD = NOD
LFBLOC(1:2) = SYM(MANY)
I = 2 + MANY*ONE
KEYVAL = BLOC(I+3:I+ONE)
IF (LEAF) THEN
LFBLOC(3:I) = BLOC(3:I)
LFBLOC(I+1:I+2) = SYM(PTR)
WRITE(IO, 601, REC=NOD) LFBLOC
MANY = FULL - MANY
ELSE
LFBLOC(3:I+2) = BLOC(3:I+2)
WRITE(IO, 601, REC=NOD) LFBLOC
I = I + ONE
MANY = FULL - 1 - MANY
ENDIF
RTNOD = PTR
RTBLOC(1:2) = SYM(MANY)
RTBLOC(3:4+MANY*ONE) = BLOC(I+1:4+FULL*ONE)
WRITE(IO, 601, REC=PTR) RTBLOC
RETURN
END
IF (LEVEL.GT.0) THEN
   INC = 1
   CALL PARENT(INC)
   IF (MANY.LT.FULL) THEN
      WRITE(10,401,REC=NOD) BLOC
      RETURN
   ELSE
      LEAF = .FALSE.
      GO TO 4005
   END IF
ELSE
   CALL NEWROOT
END IF
RETURN
END
SUBROUTINE ADDKEY

C IMPLICIT INTEGER*4 (A-Z)
C
COMMON /XXXTREE/
1 I0, ERR, MARK, MANY, PTR, NOD, LEVEL, MATCH, ONE, FULL, ABC,
2 LFM, RTM, LFNOD, RTNOD, KEYVAL, BLOC, LFBLOC, RTBLOC,
3 BUF(0:5), PATH(0:5), HTREC(10), MAXREC(10), HTNOD(10),
4 MAXNOD(10), ROOT(10), MKL(10), HEIGHT(10), NAME(10)
BYTE I0, ERR, LEVEL, HEIGHT
CHARACTER ABC*1, NAME*9, KEYVAL*20
CHARACTER*256 BUF, BLOC, LFBLOC, RTBLOC

BYTE ADD
CHARACTER SYM*2, EXT*3
LOGICAL*1 LEAF

C
C 401 FORMAT(A256)
C
C-------------------------------------------------------------
C INSERT A KEY INTO THE TREE
C-------------------------------------------------------------
C
CALL GETKEY
IF (ERR.EQ.0) THEN
  ERR = 6
  RETURN
ELSE
  ERR = 0
END IF
ADD = -1
EXT = 'REC'
CALL STACK(PTR, I0, NAME(I0), EXT, ADD, HTREC(I0), MAXREC(I0))
L = 3 + (MARK-1)*ONE
R = 4 + MANY*ONE
BLOC(L+ONE:R+ONE) = BLOC(L:R)
BLOC(L:L+1) = SYM(PTR)
BLOC(L+2:L+ONE-1) = KEYVAL
MANY = MANY + 1
BLOC(1:2) = SYM(MANY)
IF (MANY.LT.FULL) THEN
  WRITE(I0, 401, REC=NOD) BLOC
  RETURN
END IF
C
C UPDATE TREE
C
LEVEL = HEIGHT(I0)
LEAF = .TRUE.
4005 CALL SPLIT(LEAF)
SUBROUTINE LOOK(H)

C IMPLICIT INTEGER*4 (A-Z)

C COMMON /XXXTREE/
1  IO, ERR, MARK, MANY, PTR, NOD, LEVEL, MATCH, ONE, FULL, ABC,
2  LFM, RTM, LFNOD, RTNOD, KEYVAL, BLOC, LFBLOC, RTBLOC,
3  BUF(0:5), PATH(0:5), HTREC(10), MAXREC(10), HTNOD(10),
4  MAXNOD(10), ROOT(10), MKL(10), HEIGHT(10), NAME(10)

BYTE 10, ERR, LEVEL, HEIGHT
CHARACTER ABC*1, NAME*9, KEYVAL*20
CHARACTER*256 BUF, BLOC, LFBLOC, RTBLOC

C CHARACTER*20 TRY

C-------------------------------------------------------------
C SEARCH A GIVEN NODE
C-------------------------------------------------------------
C

DO 5015 M=1, MANY
   K = 5 + (M-1)*ONE
   TRY = BLOC(K:K+MKL(IO)-1)
   IF (TRY.GE.KEYVAL) GO TO 5020
5015 CONTINUE
   MARK = MANY + 1
   GO TO 5025
5020 IF (H.EQ.HEIGHT(IO)) THEN
   MARK = M
   ELSE IF (TRY.EQ.KEYVAL) THEN
   MARK = M + 1
   ELSE
   MARK = M
   END IF
   END IF
5025 K = (MARK-1)*ONE + 3
   PTR = VAL(BLOC(K:K+1))
RETURN
END
J = 2 + (MANY-LFM)*ONE
BLOC(1+1:I+J-2) = RTBLOC(3:J)
K = I + J - 1
BLOC(K:K+1) = SYM(RTNOD)
ELSE
BLOC(3:I+2) = LFBLOC(3:I+2)
BLOC(1+3:I+ONE) = KEYVAL
J = 2 + (MANY-1-LFM)*ONE
I = I + ONE
BLOC(1+1:I+J) = RTBLOC(3:J+2)
END IF
WRITE(IO,201,REC=LFNOD) BLOC
K = 3 + (MANY - LFM)*ONE
MANY = LFM + RTM - MANY
BLOC(3:4+MANY*ONE) = RTBLOC(K:4+RTM*ONE)
KEYVAL = RTBLOC(J+3:J+ONE)
END IF
BLOC(1:2) = SYM(MANY)
WRITE(IO,201,REC=RTNOD) BLOC
RETURN
END
SUBROUTINE STACK(PTR, IO, NAME, EXT, ADD, HT, MOST)

C IMPLICIT INTEGER*4 (A-Z)

C CHARACTER SYM*2, EXT*3, NAME*9, FN*13, B*128

BYTE ADD, IO

C 901 FORMAT(A128)

C-------------------------------------------------------------

C PUSH/POP (ACCORDING TO ADD = 1, -1) THE RECORD/NODE
C STACK (ACCORDING TO EXT = 'REC', 'NOD')

C-------------------------------------------------------------

C IF (HT.GT.0.OR.ADD.EQ.1) THEN
CLOSE UNIT=10
FN = NAME//'.KEY'
OPEN (UNIT=10, FILE=FN, STATUS='UNKNOWN', FORM='FORMATTED',
ACCESS='DIRECT', RECL=128)

I = 1 + HT/64
J = MOD(HT, 64)
K = 2*J
HT = HT + ADD
IF (ADD.EQ.1) THEN
IF (K.GT.0) READ(10, 901, REC=I) B
B(K+1:K+2) = SYM(PTR)
WRITE(10, 901, REC=I) B
ELSE
IF (J.EQ.0) THEN
I = I - 1
K = 126
ELSE
K = K - 2
END IF
READ(10, 901, REC=I) B
PTR = VAL(B(K+1:K+2))
END IF
IF (HT.EQ.0) THEN
CLOSE UNIT=10
ELSE
CLOSE UNIT=10, STATUS='DELETE'
END IF
FN = NAME//'.KEY'
OPEN (UNIT=10, FILE=FN, STATUS='OLD', FORM='FORMATTED',
ACCESS='DIRECT')
ELSE
MOST = MOST + 1
PTR = MOST
END IF

ENTRY POINT OF 'HEADER' IS LOCATED IN 'NEWTREE'

B-25
CALL HEADER
RETURN
END
FUNCTION VAL(A)

C IMPLICIT INTEGER*4 (A-Z)
C CHARACTER*2 A
C
C-------------------------------------------------------------
C CONVERT CODED 2-BYTE SYMBOL INTO AN
C INTEGER BETWEEN 0 AND 65535
C-------------------------------------------------------------
C
VAL = ICHAR(A(1:1))
J = ICHAR(A(2:2))
K = 2**8
DO 1005 I = 0, 7
   VAL = VAL + IBITS(J, I, 1) * K
   K = 2 * K
1005 CONTINUE
RETURN
END
CHARACTER*2 FUNCTION SYM(NUM)
C
IMPLICIT INTEGER*4 (A-Z)
C
-------------------------------------------------------------
C CONVERT AN INTEGER BETWEEN 0 AND 65535
C INTO A CODED 2-BYTE SYMBOL
C-------------------------------------------------------------
C
DO 2010 J=1,2
   SUM = 0
   K = 1
   DO 2005 I=0,7
      SUM = SUM + IBITS(NUM,(J-1)*8+I,1)*K
      K = 2*K
   2005 CONTINUE
   SYM(J:J) = CHAR(SUM)
2010 CONTINUE
RETURN
END
PROGRAM DRIVER

CHARACTER LTR*1,A*20

10 FORMAT(A1)
11 FORMAT(A25)
21 FORMAT(10X,' ILLEGAL VALUE OF LTR')
22 FORMAT(10X,' TREE CURRENTLY EXISTS')
23 FORMAT(10X,' NONEXISTENT TREE')
24 FORMAT(10X,' CANNOT FIND DESIRED KEY VALUE')
25 FORMAT(10X,' NO SUCCESSOR KEY EXISTS')
26 FORMAT(10X,' KEY CURRENTLY EXISTS IN TREE')

C-------------------------------------------------------------
C PROGRAM TO DIRECTLY EXAMINE A TREE
C-------------------------------------------------------------
C
WRITE(6,*)' ENTER LOGICAL UNIT NUMBER'
READ(5,*) LDU
100 WRITE(6,*)

WRITE(6,*)' A - add D - delete G - get key'
WRITE(6,*)' F - first S - successor key'
WRITE(6,*)' O - open C - create tree'
WRITE(6,*)' ENTER LETTER'
READ(5,10) LTR
IF (LTR.EQ.'A'.OR.LTR.EQ.'D'.OR.LTR.EQ.'G') THEN
   WRITE(6,*)' ENTER KEY VALUE'
   READ(5,11) A
ELSE IF (LTR.EQ.'O'.OR.LTR.EQ.'C') THEN
   WRITE(6,*)' ENTER TREE NAME'
   READ(5,11) A
   IF (LTR.EQ.'C') THEN
      WRITE(6,*)' ENTER LENGTH OF PRIMARY KEY'
      READ(5,*) MAXLEN
   END IF
END IF
CALL BTREE(LTR,LDU,A,MAXLEN,IREC,IERR)

C IF (IERR.EQ.0) THEN
GO TO 200
ELSE IF (IERR.EQ.1) THEN ! ILLEGAL VALUE OF 'LTR'
   WRITE(6,21)
ELSE IF (IERR.EQ.2) THEN ! TREE CURRENTLY EXISTS
   WRITE(6,22)
ELSE IF (IERR.EQ.3) THEN ! NONEXISTENT TREE
   WRITE(6,23)
ELSE IF (IERR.EQ.4) THEN ! CANNOT FIND KEY
   WRITE(6,24)
ELSE IF (IERR.EQ.5) THEN ! NO SUCCESSOR KEY
   WRITE(6,25)
ELSE IF (IERR.EQ.6) THEN ! KEY CURRENTLY EXISTS IN TREE
   WRITE(6,26)
END IF
WRITE(6,*)' ' REQUEST VOIDED'
WRITE(6,*)' ' C
WRITE(6,*)' DO YOU WISH TO EXIT? (Y/N)'
READ (5,10) LTR
IF(LTR.EQ.'N') GO TO 100
STOP
END
DISTRIBUTION

Copies

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314 12

Commander
Naval Sea Systems Command
Attn: PMS-407E 1
Washington, D. C. 20362

Library of Congress
Attn: Gift & Exchange Division 4
Washington, D. C. 20540

Internal Distribution
E231 9
E232 3
R44 (E. Winston) 25
U31 1

(1/2)
LEVEL = HEIGHT(10)
M = 1
LEAF = .TRUE.
8005 CALL ADJACENT
IF (LFM+RTM.LE.FULL-M) THEN
  CALL MERGE(LEAF)
  INC = -1
  CALL PARENT(INC)
  IF (MANY.GE.FULL/2.OR.LEVEL.EQ.0) THEN
    B-19
BLOC(3:K) = LFBLoc(I+ONE+1:J)
BLOC(K+1:K+MKL(10)) = KEYVAL
END IF
K = 3 + (LFM - MANY)*ONE
MANY = LFM + RTM - MANY
BLOC(K:4+MANY*ONE) = RTBLoc(3:4+RTM*ONE)
KEYVAL = LFBLoc(I+3:1+ONE)
ELSE
I = 2 + LFM*ONE
IF (LEAF) THEN
   BLOC(3:I) = LFBLoc(3:I)
END IF
ELSE
MOST = MOST + 1
PTR = MOST
END IF

ENTRY POINT OF 'HEADER' IS LOCATED IN 'NEWTREE'

B-25
ELSE IF (IERR.EQ.3) THEN
WRITE(6,23)  ! NONEXISTENT TREE
ELSE IF (IERR.EQ.4) THEN
WRITE(6,24)  ! CANNOT FIND KEY
ELSE IF (IERR.EQ.5) THEN
WRITE(6,25)  ! NO SUCCESSOR KEY
ELSE IF (IERR.EQ.6) THEN
WRITE(6,26)  ! KEY CURRENTLY EXISTS