

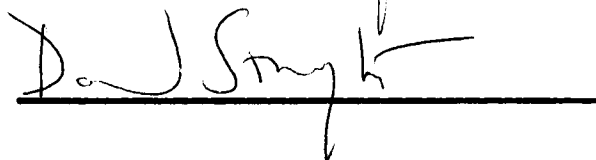
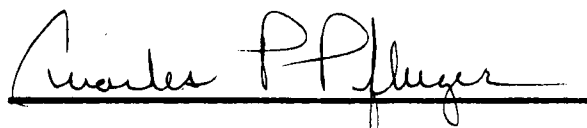
To the Graduate Council:

I am submitting herewith a thesis written by Harry S. Cook, III entitled "A B-tree Implementation of MUMPS Global Arrays Using Variable Length Keys." I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

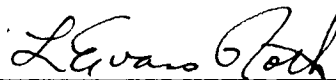


Kevin C. O'Kane, Major Professor

We have read this thesis
and recommend its acceptance:



Accepted for the Council:



Vice Chancellor
Graduate Studies and Research

A B-TREE IMPLEMENTATION
OF
MUMPS GLOBAL ARRAYS USING VARIABLE LENGTH KEYS

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Harry S. Cook, III.

March 1981

3049611

To my parents, Harry S. Cook, Jr. and Mary B. Cook

ACKNOWLEDGMENTS

The author wishes to thank Dr. Kevin C. O'Kane, Dr. Charles P. Pfleeger, and Dr. David W. Straight for their time, guidance, friendship, and service on this thesis committee, as well as throughout my graduate studies. The author would like to extend special appreciation to Dr. Kevin O'Kane for the creation of the MUMPS interpreter in which this project is encompassed, and for his experience, encouragement, and patients in making the thesis a reality.

The author would like to make special mention to Mr. Thomas B. Boaz, Jr. for his diagram artwork. Also, to Ms. Danna Carter for proof reading some of the earlier versions of this paper.

ABSTRACT

MUMPS, (Massachusetts General Hospital Utility Multi-Programming System), was first developed in the mid 1960's as a high-level interpretive programming language. MUMPS was designed to facilitate the creation of conversational programs that would permit dynamic data base interaction in the clinical environment. The files that contain the data base are hierarchically organized tree file structures called "global arrays".

As with any data base management facility, the requirement to service an interactive user is highly dependent upon the speed with which a given token of information can be stored, deleted, and retrieved from its peripheral storage location. This paper describes a B-tree data structure implementation of MUMPS global arrays with variable length keys.

The B-tree data structure was first developed by Dr. R. Bayer and Dr. E. M. McCrieght in the early 1970's. This data structure allows any random number of key insertions or deletions to be performed upon the tree structure without disturbing the excellent benefits gained from having a balanced tree. Their work dealt with a tree structure of nodes which contained the obsticale of fixed length keys. In this project the basic B-tree algorithms are followed, but with new modifications made to allow

variable length keys inside the B-tree node. This modification in some instances has been found to lead to a better disk space allocation.

TABLE OF CONTENTS

CHAPTER	PAGE
INTRODUCTION.	1
Project Environment	2
Constraints Placed on MUMPS Global Arrays Unique to This Implementation.	3
I. MUMPS HISTORY AND DEVELOPMENT	4
Early History	4
MUMPS Development	5
Standardization Effort.	6
II. INTRODUCTION TO THE B-TREE DATA STRUCTURE	7
B-tree Definition	7
B-tree Maintenance.	8
III. MUMPS GLOBAL ARRAYS	9
General Overview.	9
Global References	9
The Naked Reference	11
Global Array Directives	12
The SET and READ commands	13
The KILL Command.	15
The LOCK Command.	15
Tracing and Existence Functions	15
The \$NEXT Function.	17
The \$DATA Function.	17
IV. THE GLOBAL ARRAY FILES.	19
The Global Array String File.	19
The B-tree Global Array Node File	21
B-tree Global File Header	21
The Index Nodes	24
The B-tree Global Array Node Blocks	26
The B-tree Node Block Header.	27
The B-tree Node Subkeys	27
Subkey Field Referencing Within the B-tree Node Block	30
B-tree Global Array Node File Space Allocation.	31
V. THE GLOBAL ARRAY PROCESSOR.	35
General Overview.	35
Global Array File Initialization.	36
The Global Array Processor Routines	39
Retrieval of a Global Array Reference	42
Store/Replace a Global Array Reference.	44
B-tree Global Array Node File Insertion	44
B-tree Global Array Node Value Replacement.	48

CHAPTER

PAGE

V. (Continued)

The \$NEXT Function of the Global Array Reference. .	49
Deletion of a Global Array Reference.	50
The \$DATA Function of the Global Array Reference. .	52
The B-tree Global Array Node File	
Dynamic Storage Management	54
Global Array String File Maintenance.	56
VI. PERFORMANCE ATTAINED.	60
B-tree Search Time.	60
B-tree Insertion Time	61
B-tree Deletion Time.	62
Variable Length Keys Vs. Fixed Length Keys.	65
Conclusion and Evaluation	66
LIST OF REFERENCES.	69
APPENDICES.	71
APPENDIX A. B-TREE SEARCH ALGORITHM.	72
APPENDIX B. B-TREE INSERTION ALGORITHM	76
APPENDIX C. B-TREE DELETION ALGORITHM.	82
APPENDIX D. SUBROUTINE CROSS REFERENCE	90
APPENDIX E. EXPLANATION OF THE COMMON	
COMMUNICATION VECTORS.	93
APPENDIX F. STRING FILE COMPRESSION ALGORITHM. . .	98
VITA.	101

LIST OF TABLES

TABLE	PAGE
I. Subkey Field Addressing.	32
II. B-tree Node Block Space Allocation	34
III. Global Array Directives.	40
IV. Subroutine Cross Reference	90
V. Main Common Vectors.	94
VI. Global Array Processor Communication Vectors	97

LIST OF FIGURES

FIGURE	PAGE
3-1. Conceptual View of a Global Array.	10
3-2. The Head of the Global Array	14
3-3. The Effect of a Global Array Node Deletion	16
4-1. Global Array String File	20
4-2. Global Array String File Header.	22
4-3. B-tree Global Array Node File.	23
4-4. B-tree Global Array Node File Header	25
4-5. B-tree Node Block Header	28
4-6. B-tree Global Array Subkey	29
5-1. The Global Array File Initialization Process . . .	38
5-2. The Global Array Processor Process	41
5-3. The Global Array Retrieval Process	45
5-4. The Global Array Insertion Process	47
5-5. The Global Array \$NEXT Function Process.	51
5-6. The Global Array Deletion Process.	53
5-7. The Global Array \$DATA Function Process.	55
5-8. The Global Array String File Compression Process .	59
A-1. The Nodes of an M-way Search Tree.	73
A-2. A Stable B-tree Before a Key Insertion	75
B-1. A Stable B-tree After the Insertion of Key 5 . . .	77
B-2. An Unstable B-tree After the Insertion of Key 4. .	78

FIGURE	PAGE
B-3. An Unstable B-tree After a Node Split.	80
B-4. A Three Level Stable B-tree.	81
C-1. A Stable B-tree of Order 5	85
C-2. An Unstable B-tree After the Removal of Key 9. . .	86
C-3. An Unstable B-tree After the Catenation Process. .	87
C-4. A Stable B-tree with the Catenation Process Creating a New Root.	89
E-1. The G Communication Vector	95

INTRODUCTION

This thesis implements a B-tree data structure to handle variable length keys in processing MUMPS global arrays. The project will replace the traditional binary mapped N-way tree implementation of MUMPS global arrays currently incorporated in the host interpreter. The project has the responsibility to handle the full set of MUMPS global array directives; to find global references, insert global references, delete global references, and process the \$NEXT and \$DATA global array functions.

Along with the management of the global array structure, this project also has the responsibility to perform MUMPS global file management. Routines were written to handle allocation/deallocation of B-tree nodes and to perform string file compression on fragmented string file space. The goals of this project were:

- 1) to program the global array file processor,
- 2) to optimize global array file space utilization, and
- 3) to reduce response time in accessing information contained in the global array file structure.

Project Environment

The project in which this work is incorporated is the single user version of the MUMPS interpreter developed at the University of Tennessee, Knoxville by Dr. Kevin C. O'Kane. This implementation of MUMPS is designed primarily for use on the Digital LSI-11/2 processor with the KEV-11 floating point option, 60KB main memory (extended for Q-Bus), 2 RX02 diskette drives, the RT-11 SJ or FB/XM V3B operating system with timer support, and a CRT terminal. The interpreter accepts the full standard MUMPS as defined by the U.S. Department of Commerce/National Bureau of Standards [NBS,1976], with the exception of a few directives that are associated with a multi-terminal supported system.

The MUMPS interpreter, including the global array processor, is written entirely in RT-11 FORTRAN, with a few system dependent library calls. This feature allows transportability to other systems with limited software changes. Other specialized modules may be readily incorporated. They need not necessarily be written in RT-11 FORTRAN, though every module must conform to RT-11 linking conventions.

Constraints Placed on MUMPS Global Arrays

Unique to This Implementation

This implementation of the MUMPS global array handler varies slightly from the general definition. First, it does not handle the LOCK global array directive. This directive would be meaningless in the single user version of MUMPS in which this project is incorporated. If necessary, this directive would be simulated by the main interpreter by keeping a table of "locked" nodes.

The second difference places a greater restriction on MUMPS global arrays. This implementation does not allow the user to express global array references with unlimited upper and lower bounds, or to express global array references in any particular data type. It will handle indices varying in length from 1 to a maximum of 13. The user, though, may only reference to a maximum of 12 indices, with the first index being reserved for global array identification. The global array identification index, unseen to the user, allows simulation of multiple global arrays inside one global array file. The second global array reference constraint is that all indices are treated as REAL*4 values. This allows the user to specify indices with 7 digits of accuracy, and up to a maximum length of 20 digits. The processor does not allow alphabetic or other data type indices.

CHAPTER I

MUMPS HISTORY AND DEVELOPMENT

Early History

During the early 1960's most clinical management systems were developed on small computers [Greenes,1969]. These machines were too small to program in higher level languages, such as FORTRAN or ALGOL, so most of the application programs were written in assembler or machine language. The cost to develop and maintain such programs was too restrictive for most physicians [Greenes,1969]. In 1964 the RAND corporation released an interpretive language called JOSS to run on their JOHNNIAC machine. The purpose behind JOSS was to give users a direct way of communicating small numerical problems to the machine without a large investment in time learning the operating system, its assembler, or its debugging tools [Sammet,1969].

During the same time, Dartmouth College was developing its own interpreter called BASIC for its G.E. 225 computer. The premise behind its development was to educate people on how to program [Sammet,1969]. It was relatively simple to use, but did not have the string handling capabilities needed in the clinical environment.

MUMPS Development

MUMPS, Massachusetts General Hospital Utility Multi-programming System, was developed between 1965 and 1969, in the Laboratory of Computer Science at Massachusetts General Hospital. MUMPS was designed to handle the medical profession's textual and statistical information in a simple, rapid, and an efficient manner on an interactive basis, thus filling the void between the numeric capabilities of JOSS, and the simplicity of BASIC.

MUMPS is an interpretive language. With an interpretive language, programs are run directly in their source code. A typical program written in an interpretive language takes up about 10%-20% of the space taken up by the object code generated for a similar program written in a compiler language [Greenes,1969]. Other advantages include rapid program development, facilitated by immediate program correction and re-execution. System application changes and additions can rapidly be made without hindering everyday service activities [Sammet,1969].

Much of the information processing done in medical applications involves the storing, retrieving, and editing of text-string information. MUMPS provides extensive string processing facilities. In addition to arithmetic and Boolean capabilities, it handles string data in a symbolic format. MUMPS commands can perform a pattern matching

operation to determine if the string contains a predefined sequence of numerics, alphabets, punctuation, or any combination of these. Strings can be assembled, disassembled, modified, or searched with the same simplicity that most higher-level languages give to simple arithmetic statements.

Standardization Effort

On September 17, 1975 MUMPS was standardized by the National Bureau of Standards [NBS,1976]. This action was done for the benefit of MUMPS application programmers and those wishing to implement MUMPS. Special consideration was given to the portability of source code between various MUMPS implementations. Guidelines were established for the implementors and application programmers. To be standard, new implementors of MUMPS must accept at least all the syntax rules as outlined by the 1975 standard. Application programmers, on the other hand, to be compatible with other implementations of MUMPS, may meet but must not exceed the limits as described by the 1975 standard [NBS,1976].

CHAPTER II

INTRODUCTION TO THE B-TREE DATA STRUCTURE

B-tree Definition

The B-tree, developed by Dr. R. Bayer and Dr. E. M. McCreight, [Bayer,1972], was presented as a way to organize and to maintain indices for a dynamically changing random access file; indices are keys with associated information. A B-tree of order M is defined as an M-way search tree that is either empty, or is of height greater than or equal to 1 and satisfies the following properties [Bayer,1972]:

- 1) Every node in the B-tree, has at most M sons.
- 2) The root has at least two sons (unless it is a leaf).
- 3) All nodes other than the root node and leaves have at least greater than or equal to $\lceil M/2 \rceil - 1$ sons.
- 4) All leaves are on the same level.
- 5) A nonleaf node with K sons will contain K-1 keys.

The B-tree offers several advantages [Bayer,1972].

Among these are:

- 1) The storage utilization is at least 50 percent at any time, and usually should be better on the average.
- 2) There is no congestion problems or degradation of performance as the storage occupancy of the B-tree file becomes full, since storage is requested and released dynamically

and independently as the file grows and constricts.

- 3) The B-tree maintains its indices in a constant balanced M-way tree order, no matter what the order of insertions and deletions.

B-tree Maintenance

A B-tree is organized in fixed sized pages. Pages are blocks of information transferred between main storage and the system's auxiliary storage. Each page in the B-tree constitutes a node in the tree structure. Each page may hold a maximum of M indices, though a page may be partially filled. In this data structure the tree grows and contracts in only one way: nodes split in half and become brothers, or two brothers are combined or "catenated" into a single node. The splitting and catenation processes are initiated at the leaves only and propagate upward toward the root. If the root splits, a new root node will be introduced; this is the only way the height of the tree can grow. The opposite process occurs if the tree contracts. See Appendices A, B, and C for the B-tree search, insert, and delete algorithms.

CHAPTER III

MUMPS GLOBAL ARRAYS

General Overview

The data base facility offered by MUMPS is done through global arrays. The data structure may be thought of as a directed acyclic tree, with a head node that has all its children descending downward away from it. Figure 3-1 graphically depicts the MUMPS global array. The global array is hierarchical in construction, consisting of linked tiers or levels. Each level in the global array structure consists of a number of data points or nodes. These nodes may contain:

- 1) An integer value
- 2) A real value
- 3) A string value
- 4) An undefined value
- 5) Any of (1) to (4) with a pointer to descendant nodes

Global References

Syntactically, nodes are specified by a sequence of subscripts appended to the name of the global array; for example `^GBL(3,4,5)`. The circumflex "^" is used to signify the use of a global array; The name of the global array is

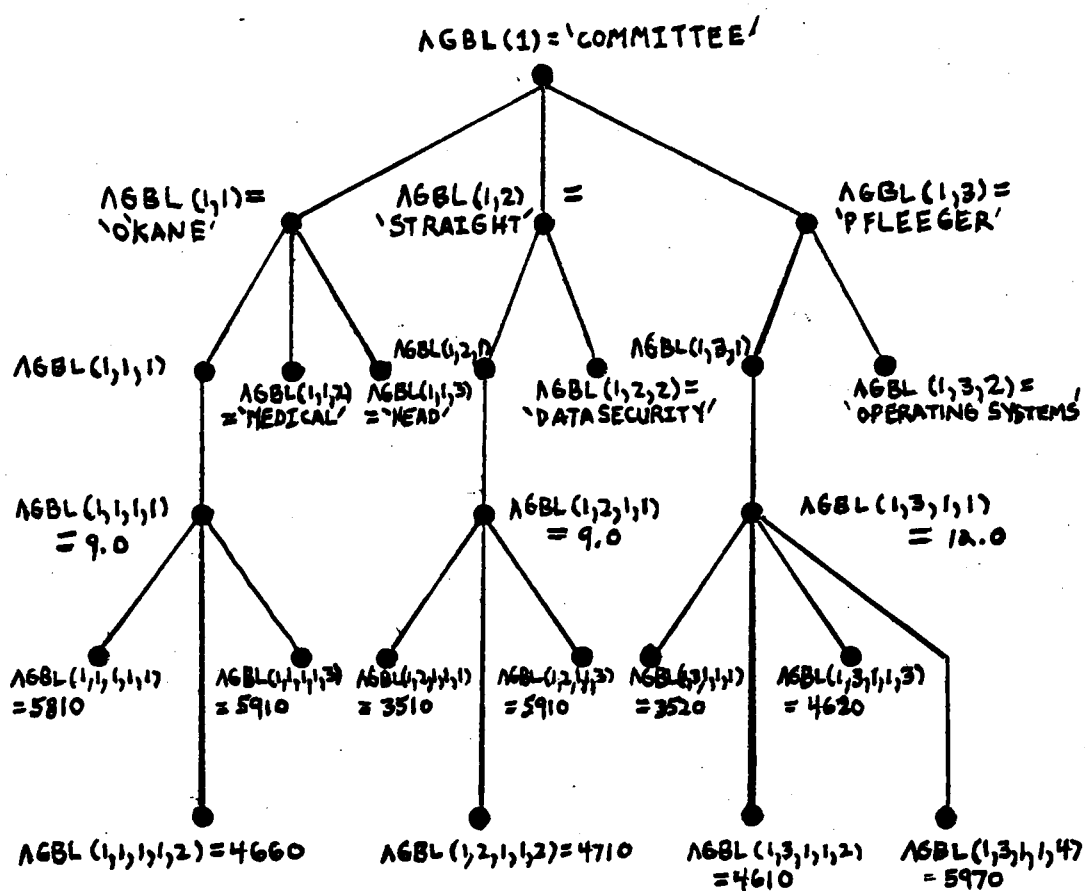


Figure 3-1. Conceptual View of a Global Array

GBL, and (3,4,5) is the sequence of subscripts specifying the node desired. Each position in the subscript list represents a different level of the tree, with the specified subscript list representing a trace (or path) from the head to the particular node.

The language does not restrict the values of the subscripts at the various levels, with the exception that indices may not be negative. Thus MUMPS makes no assumptions about the bounds of its global array subscripts; i.e. global arrays are not predimensioned. The global array is treated as a sparse matrix. Storage is allocated for a node only if it is created to contain a value, or to provide a path to a particular node, which has been explicitly assigned a value. Global arrays are dimensionless with storage being dynamically allocated and deallocated.

The Naked Reference

In general, a global array node may be accessed through an arbitrary number of subscripts. The only exception allowed in referencing a global array node is through the MUMPS special feature called the "naked reference". The naked reference is a syntactic structure which allows transversals of descendants from a given global array node.

The naked reference may be utilized only after an

initial global array reference has been processed. After the initial reference, the global array name and the invariant part of the global array subscript list may be omitted. Thus the programmer need only specify those subscripts which differ from the last global array reference.

For example, consider the global array as shown in Figure 3-1, with an initial reference of $\text{^GBL}(1,1)$, which references the head of the subtree containing information for "O'KANE". To reference $\text{^GBL}(1,2)$, or the information for the "STRAIGHT" subtree, it would be possible to abbreviate the global array reference $\text{^GBL}(1,2)$ to $\text{^}(2)$. The naked reference may be extended to lower levels as well. For example, given the initial reference as $\text{^GBL}(1,1)$, referencing $\text{^}(1,3)$ would have the same effect as referencing $\text{^GBL}(1,1,3)$. It is important to note that the naked references are one directional, away from the root node. In other words, after $\text{^}(1,3)$ had been processed, a reference to $\text{^GBL}(1,1)$ would require a full subscript list.

Global Array Directives

MUMPS contains six directives to manipulate global arrays: SET, READ, KILL, LOCK, \$NEXT, and \$DATA. The first four are actual global array commands that affect the global tree structure and access rights, the latter two are MUMPS

functions that indicate global array characteristics.

The SET and READ Commands

The SET command either places a value into the global array, or extracts a value from it. The READ command assigns a value that is defined outside the MUMPS program to a global array reference. For example consider Figure 3-2 and the commands below:

```
SET ^GBL(1) = 'COMMITTEE'
```

or

```
READ ^GBL(1)
```

Either of these commands would create the initial, or root, global array node and assign the string value "COMMITTEE" to it. Subsequent SET or READ commands might create the global array depicted in Figure 3-1. The creation of ^GBL(1,1,1,1) implies that ^GBL(1), ^GBL(1,1), and ^GBL(1,1,1) exist. If they were not, then the command:

```
SET ^GBL(1,1,1,1) = 9.0
```

or

```
READ ^GBL(1,1,1,1)
```

would create those nodes in order to form a complete path from the root to the new global array entry. Notice that the node ^GBL(1,1,1) does not contain a value, its purpose is to point to ^GBL(1,1,1,1).

AGBL(1) = 'COMMITTEE'



Figure 3-2. The Head of the Global Array

The KILL Command

The KILL command instructs the global array processor to delete a particular node and all its descendants from the global array structure. For example consider Figure 3-1 on page 10 and the command:

```
KILL ^GBL(1,1)
```

This command would cause the removal of "O'KANE" and all related information. The resulting global structure is illustrated in Figure 3-3. MUMPS allows the killing of all information contained in a global array by issuing the command:

```
KILL ^GBL
```

This global reference, without a subscript list, would cause the removal of every global array node.

The LOCK Command

The LOCK command allows the MUMPS user to restrict access to particular portions of the global array structure. Its purpose is meaningless in a single user environment, and thus will not be discussed.

Tracing and Existence Functions

Due to the dynamic nature of MUMPS global arrays, it is rarely possible to know beforehand the structure of a

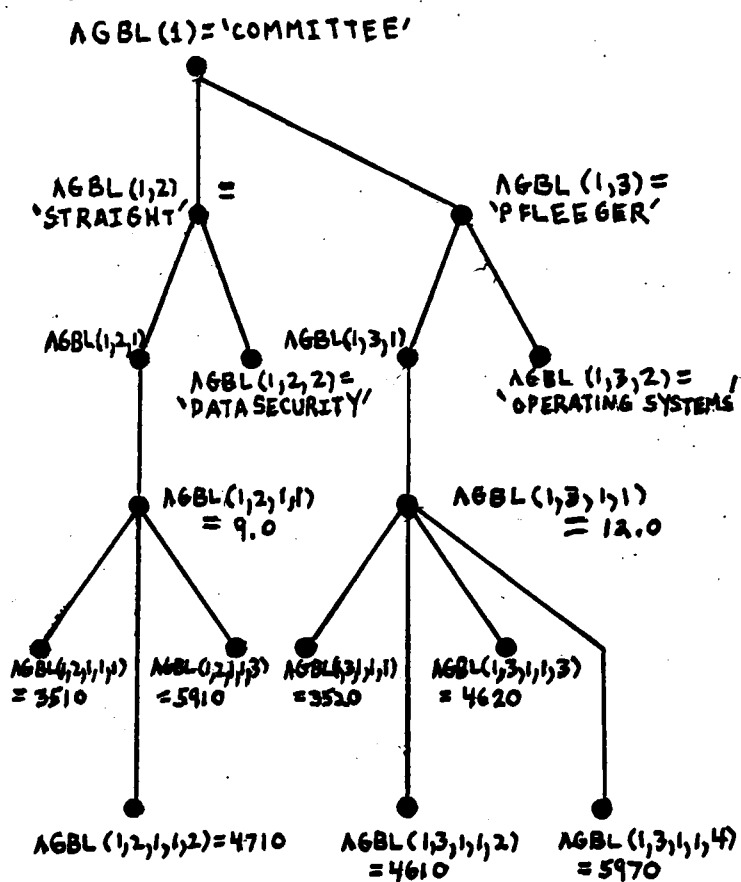


Figure-3-3. The Effect of a Global Array Node Deletion

particular global array; i.e. whether a particular node is a terminal node, a trace node, or a node with or without an associated value. The MUMPS language provides two functions for inquiry about the global structure: \$NEXT and \$DATA.

The \$NEXT Function

The \$NEXT function provides the ability to trace all siblings at a given level below the root node in ascending subscript order. In other words, the \$NEXT function returns the value of the next higher subscript at the lowest level reference in a global array node trace. A -1 is allowed as the lowest level subscript in \$NEXT in order that the smallest subscript on that level can be determined. If \$NEXT returns a -1, no higher subscripts exist at the level referenced. For example in Figure 3-1 on page 10, the function call

```
$NEXT(^GBL(1,1))
```

returns the value 2, or the index heading the subtree containing "STRAIGHT"'s related information.

The \$DATA Function

The \$DATA function provides a way of determining whether a particular global array node exists, whether it has descendant nodes, and/or whether a value is associated with it. This function returns a two digit binary code.

The units digit is 1 if the node referenced has a value associated with it, and 0 if no value exists. The tens digit is a 1 if the node has descendant nodes. A \$DATA reference to a non-existent node, or a terminal node with no associated value, would return a code of 00. For example, consider Figure 3-1 on page 10 with the function call:

\$DATA(^GBL(1,1,1))

This function call would return a code of 10 since the node exists with a descendant node, but does not contain a value.

CHAPTER IV

THE GLOBAL ARRAY FILES

The Global Array String File

The global array string file is the resource by which character string values associated with MUMPS global array references are stored. (A character string is a continuous group of characters, to a maximum of 255, terminated by a zero byte.) This file consists of an arbitrary number of 512 byte blocks, which may be thought of as a contiguous block of storage. Figure 4-1 graphically depicts the structure of the global array string file.

The first nine bytes of this file are reserved as the header of the file. The first four bytes contain a pointer to the next available byte in the file in which a character string may be stored. This address, as well as any string address, is divided into two INTEGER*2 values. The first contains the block number in the file where the start of the character string may be found. The second value contains the offset into that block where the string begins. The next four bytes contain the total number of bytes available to store a character value.

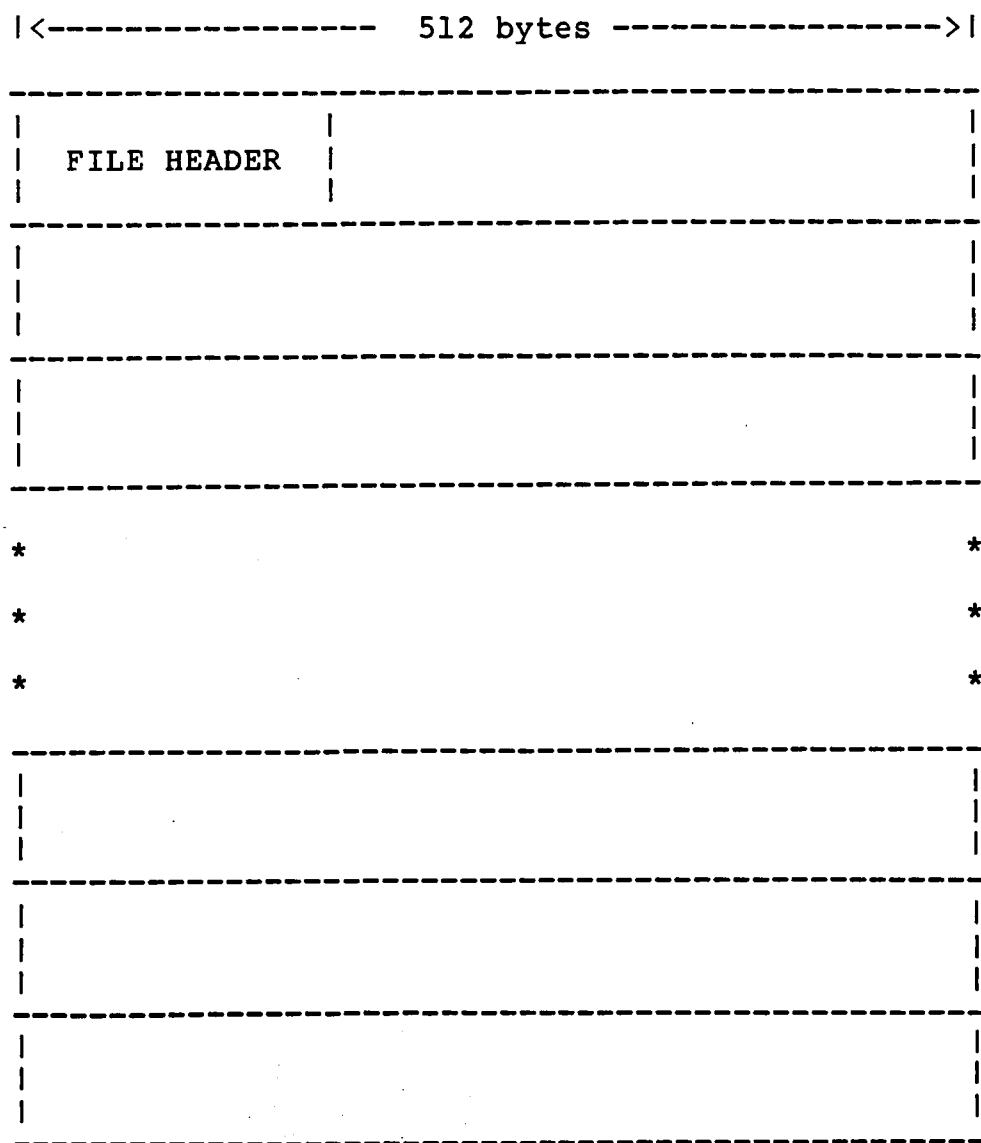


Figure 4-1. Global Array String File

The final byte is used for a file initialization code byte. After the file is initially opened, subroutine GBLOPN checks this byte for a value of 250. If this byte does not contain 250, then the file initialization routine, SINIT, is called. Figure 4-2 shows the layout of the string file header.

The B-tree Global Array Node File

This file is the dynamic storage resource for the contents of MUMPS global arrays. The B-tree global array node file consists of an arbitrary number of 512 byte blocks (the optimal I/O block size). There are two types of information blocks contained in this file, Index and B-tree global array node blocks (B-tree node blocks). The number of each block type is determined by the user-defined file size parameter, G5RECS, passed in common from the host interpreter. The maximum file size MUMPS can handle is 32,767. The maximum file size corresponds to the largest RT-11 FORTRAN INTEGER*2 value. See Figure 4-3 for the graphic design of the B-tree global array node file.

B-tree Global File Header

The first 11 bytes of this file are reserved for important history. A maximum of 4,008 nodes may be mapped

	--	END-OF-FILE MARK	--
		FILE FREE SPACE	
		INITIALIZATION BYTE	

2 INTEGER*2

REAL*4

LOGICAL*1

Figure 4-2. Global Array String File Header

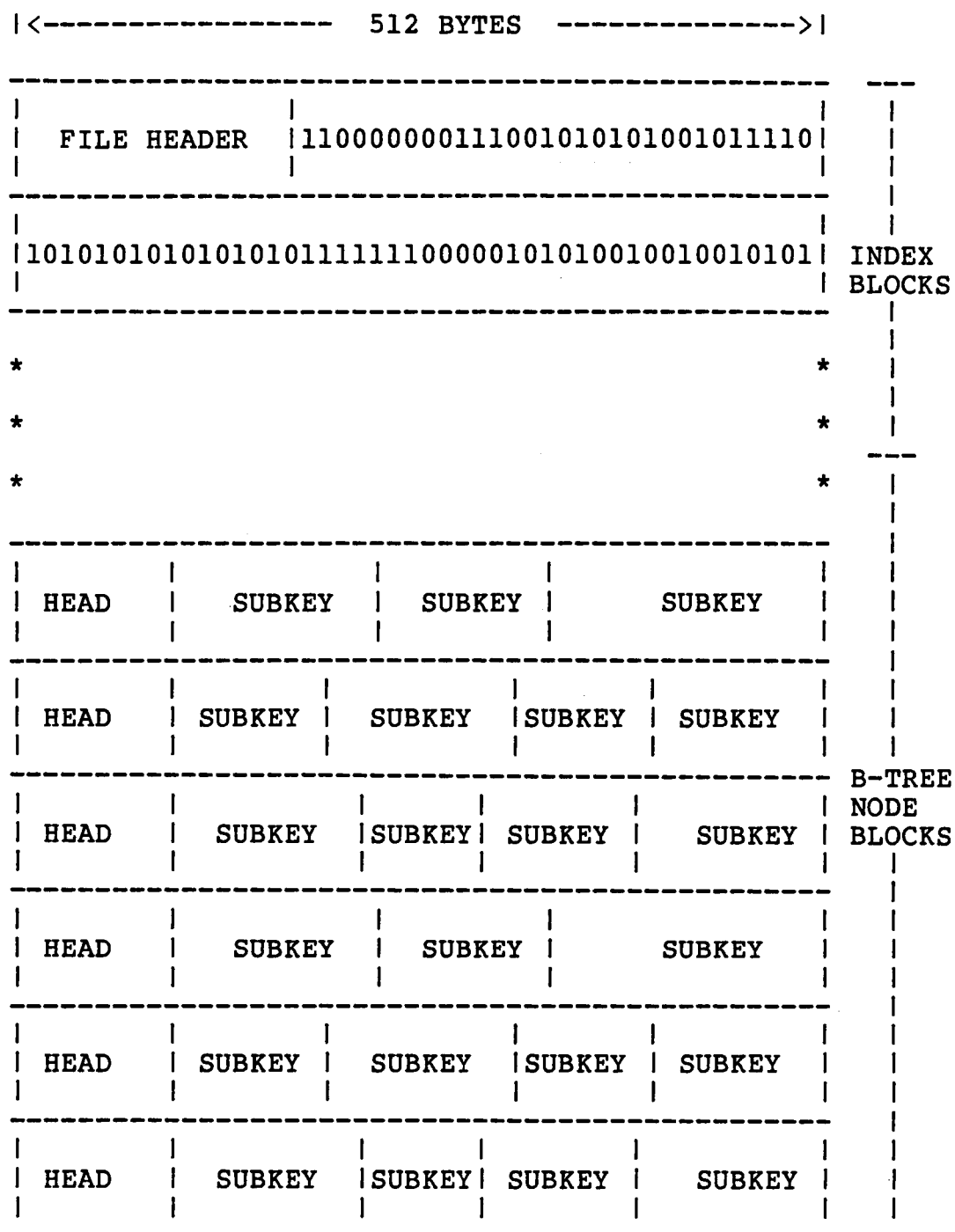


Figure 4-3. B-tree Global Array Node File

in the first index node. The first four bytes of this header contain the block number of the node which serves as the root node of the B-tree. The second four bytes reflect the number of unallocated blocks. The next two bytes indicate the number of necessary index nodes to map this file. The 11th byte is an initialization indicator byte. This last header byte is checked immediately after this file is opened; if it contains a code of 250 then the file initialization routines are not executed. The information in the file header is updated when global array processes change the information reflected in it; i.e. when a new root node is assigned, or a B-tree node block is allocated/deallocated. The added I/O to update the file header protects from destroying the file in the case of program abnormal termination. See Figure 4-4 for the graphic representation of the B-tree global array file header.

The Index Nodes

The index nodes are the media by which dynamic allocation/deallocation of B-tree node blocks are managed. Each bit in these index blocks maps to a block inside the B-tree global array file. They are automatically reserved during the B-tree global array node file initialization

B-TREE ROOT NODE BLOCK NUMBER	INTEGER*4
NUMBER OF INDEX NODES	INTEGER*4
NUMBER OF UNALLOCATED BLOCKS	INTEGER*4
INITIALIZATION BYTE	LOGICAL*1

Figure 4-4. B-tree Global Array Node File Header

routine. Except for the first block, which has the first eleven bytes reserved for file header information, each index block can map 4,096 nodes, or 512 bytes by 8 bits/byte. If the bit associated with a certain file block is 1, then the block is allocated, else if it is 0, then the block is unallocated. The following equation will yield the number index nodes reserved for a given file size:

$$\text{INXNOD} = (\text{G5RECS} + 87) / 4097 + 1$$

For example, a file with 5000 blocks would have the first 2 blocks reserved as index blocks.

The B-tree Global Array Node Blocks

The second file block type contains the actual nodes which reflect the information processed through MUMPS global arrays. Each B-tree node block consists of a node header and from 1 to 30 global array nodes, or what are referred to as "subkeys". The number of B-tree node blocks contained in this file may be given by the following equation:

$$\text{B-tree node blocks} = \text{G5RECS} - \text{INXNOD}$$

For example, given a file size of 5000 block, the first 2 blocks would be reserved as index nodes, which would leave 4,998 B-tree node blocks.

The B-tree Node Block Header

The first 8 bytes of each B-tree node block contain information pertinent to the maintenance, and its position within the B-tree file structure. The first two bytes contain an INTEGER*2 value representing the total number of bytes utilized by the individual subkeys in the particular B-tree node block. The second two bytes are an INTEGER*2 value representing the starting byte of the subkey with the smallest global array reference key in the B-tree node block. The remaining four bytes contain an INTEGER*4 value whose low order 2 bytes point to the B-tree node block's left descendant. The high order 2 bytes of this value are reserved for future expansion. See Figure 4-5 for a graphic explanation of the B-tree node block header.

The B-tree Node Subkeys

The subkey is the smallest complete unit of information that manages global array information. In the traditional way of viewing a global array, a subkey would be the global array node. Each subkey is between 16 and 64 bytes in size. The size varies with the number of global array node reference subscripts that are catenated together to form the subkey's key. See Figure 4-6 for a graphic explanation of the subkey elements.

NUMBER OF SUBKEY INFORMATION BYTES	INTEGER*2
HEAD OF THE SUBKEY LINK LIST	INTEGER*2
LEFT DESCENDANT NODE POINTER	INTEGER*4

Figure 4-5. B-tree Node Block Header

THE GLOBAL ARRAY REFERENCE DATA VALUE OR THE BLOCK # and OFFSET STRING POINTER	REAL*4 or 2 INTEGER*2
DESCENDANT POINTER	INTEGER*4
SUBKEY LINK LIST POINTER	INTEGER*2
DATA TYPE CODE	LOGICAL*1
SUBKEY KEY LENGTH	LOGICAL*1
SUBSCRIPT NUMBER 1	REAL*4
SUBSCRIPT NUMBER 2	REAL*4
SUBSCRIPT NUMBER 3	REAL*4
*	*
*	*
*	*
SUBSCRIPT NUMBER 12	REAL*4
SUBSCRIPT NUMBER 13	REAL*4

Figure 4-6. B-tree Global Array Subkey

The first four bytes, depending on the 11th subkey byte, contain either a REAL*4 value or a pointer into the global array string file where the string associated with the subkey is stored. The next two bytes, or the low order bytes of an INTEGER*4 value, point to the B-tree node block which is the right descendant of the subkey. The next two bytes, or the high order of an INTEGER*4 value, are reserved for future subkey expansion. The next field is a two byte, INTEGER*2 field which contains the starting byte number of the next subkey in the B-tree node block; a zero in this field indicates the largest subkey's key has been scanned. The 11th byte, relative to the start of the individual subkey, is a LOGICAL*1 field which reflects the data type of the value associated with this subkey; a code of 1 = REAL*4, 2 = character string, and 4 = NULL or undefined. The 12th byte is a LOGICAL*1 value which reflects the number of global array subscripts, or the length of the key, that were catenated to form the subkey's key. The remaining 4 to 52 bytes are the REAL*4 global array subscripts.

Subkey Field Referencing Within the B-tree Node Block

The individual subkeys within a B-tree node block contain data fields that must be referenced by four different data types: LOGICAL*1, INTEGER*2, INTEGER*4, and REAL*4. In order to properly attain the information stored

in a particular subkey, there must be four arrays, one of each data type, equivalenced together. Given LNODE(512), TNODE(256), PNODE(128), and RNODE(128) as the LOGICAL*1, INTEGER*2, INTEGER*4, and REAL*4 arrays respectively, and BYTE1 as the first byte in the B-tree node block where a subkey is located, Table I illustrates the different references to each field in the selected subkey.

B-tree Global Array Node File Space Allocation

In any data base system, there is concern for the efficient use of the peripheral storage capacity. From its definition, a B-tree of order M will contain B-tree node blocks consisting of at least $\lceil M/2 \rceil - 1$ subkeys to a maximum of M - 1 subkeys. The root is the only exception, it may consist of between 1 and M - 1 subkeys.

In order to find the minimum number of subkeys that can fit into the B-tree global array node file of size P, the assumption must be made that every B-tree node block in the file contains the minimum number of subkeys; i.e., the root node will contain 1 subkey, and all the other B-tree node blocks will contain $\lceil M/2 \rceil - 1$ subkeys.

TABLE I
SUBKEY FIELD ADDRESSING

INFORMATION	REFERENCE

# of SUBKEY BYTES	= TNODE(1)
SUBKEY LINK LIST HEAD	= TNODE(2)
LEFT DESCENDANT PTR	= PNODE(2)
DATA FIELD	= RNODE((BYTE1/4) + 1)
DESCENDANT PTR	= PNODE((BYTE1/4) + 2)
SUBKEY LINK LIST PTR	= TNODE((BYTE1/2) + 5)
DATA TYPE	= LNODE(BYTE1 + 10)
LENGTH OF KEY	= LNODE(BYTE1 + 11)
KEY ELEMENT	= RNODE((BYTE1/4) + 3 + I)

where I is between 1 and 13

To find the maximum number of subkeys that will fit in the same file, one must assume that each block contains $M - 1$ subkeys, the maximum number of subkeys allowed in a block. The equations below may be used to determine the limits on the number of subkeys a B-tree global array file will contain:

$$\text{Minimum number of subkeys} = 1 + ([M/2] - 1) * (P - 1)$$

$$\text{Maximum number of subkeys} = (M - 1) * (P)$$

For example, let the file size be equal to near the capacity of a double density floppy disk, or 960 blocks. From Table II, a B-tree consisting of global reference with 5 levels of indices yields a tree of order 14. The minimum number of subkeys that will be able to fit is 5,755 with possible expansion to 12,480 subkeys.

TABLE II
B-TREE NODE BLOCK SPACE ALLOCATION

NUMBER OF INDICES	ORDER M	MINIMUM NUMBER SUBKEYS/ BLOCK	MAXIMUM NUMBER SUBKEYS/ BLOCK	MINIMUM NUMBER SUBKEYS/ FILE FILE = 960	MAXIMUM NUMBER SUBKEYS/ FILE BLOCKS
1	28	12	27	12468	25920
2	23	10	22	9591	21120
3	19	8	18	7673	17280
4	16	6	15	6714	14400
5	14	5	13	5755	12480
6	13	5	12	4796	11520
7	12	4	11	4796	10560
8	11	4	10	3437	9600
9	10	3	9	3437	8640
10	9	3	8	2878	7680
11	8	2	7	2878	6720
12	8	2	7	2878	6720
13	7	2	6	1919	5760

CHAPTER V

THE GLOBAL ARRAY PROCESSOR

General Overview

The global array processor is a network of 17 routines to manage MUMPS global arrays with the B-tree data structure to organize the global array files. (For a cross reference of the different routines, see Appendix D.) The global array processor routines may be divided into 4 categories:

- 1) The global array file initialization routines are called to open and clear the global array files. The routines in this category are GBLOPN, SINIT, and BINIT.
- 2) The global array processor routines are responsible for the execution of the global array directives. The routines in this category are GLOBAL, FIND, KEYCOM, INSERT, DELETE, SKCOMP, INSSTR, FETSTR, DELDAT, and BLKADD.
- 3) The B-tree global array node file dynamic storage management routines manage the storage of the B-tree global array node file. The routines in this category are

GETNOD and PUTNOD.

- 4) The global array string file maintenance routines will compress the fragmented global array string file. The routines in this category are SQUASH and READJ.

Global Array File Initialization

The global array processor has two disk files through which it handles the storage of MUMPS global arrays. One file is the B-tree global array node file. This file contains the B-tree structure which manages the individual global array references. The other file is a secondary file to store the character string values associated with global array references.

In order for the global array processor to work properly, it must have its two files opened and initialized to their proper format. The subroutine GBLOPN is responsible for opening the two files, and making sure they have been properly formatted. If the B-tree global array node file has not been initialized, GBLOPN calls subroutine BINIT to format the file. During the B-tree global array node file initialization, the number of index node blocks are calculated, and every bit in them is set to zero. The B-tree node block bytes, with the exception of the B-tree node block header which is set to zero, are initialized to a

code of 255. The header information is gathered and stored, and then control is returned to GBLOPN.

If the string file has not been formatted, GBLOPN calls subroutine SINIT to initialize the file. During the global array string file initialization, every byte with the exception of the global array string file header which is set to zero, is set to a code of 250. The end-of-file mark is set to block 1/byte 10, or the first available byte that may contain a character string value. Next, the amount of global array string file free space is calculated from the following equation:

$$\text{FREESP} = (\text{G2RECS} * 512 \text{ BYTES}) - 9 \text{ HEADER BYTES}$$

The file header information is stored, and control returns to GBLOPN.

GBLOPN may be called either by MUMPS or by subroutine GLOBAL. MUMPS would make the initial call since it has the responsibility of obtaining the interpreter resources. MUMPS would also call this routine in order to clear the global files during normal system execution. GLOBAL would only call GBLOPN if the global array processor was initiated after MUMPS had issued a directive to close the global files. Figure 5-1 shows the structure of the initialization process.

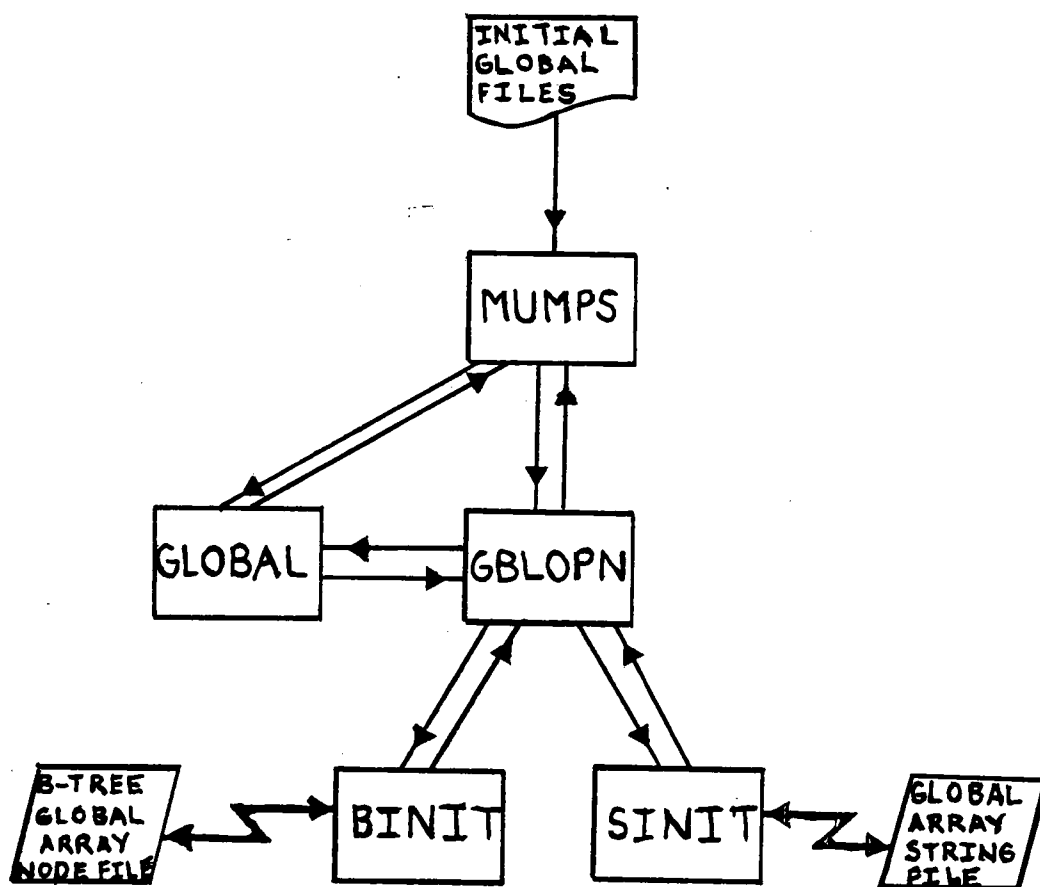


Figure 5-1. The Global Array File Initialization Process

The Global Array Processor Routines

Subroutine GLOBAL is the main module called by the MUMPS interpreter to process global array references. When invoked, GLOBAL first performs a series of initial steps. The first step is to make sure that the two global array files have been opened. If either has not been opened, a call to subroutine GBLOPN is made. The root node of the B-tree global array file is then identified and stored. The next step is to extract the global array reference subscripts passed from MUMPS. The subscripts of the global array reference are then catenated to form a key that may be compared to other keys stored in the B-tree node blocks; for uniqueness, each key is right padded by a subscript value of zero when the global array reference has fewer than 13 subscripts.

The next process selects which global array directive the global array processor has been requested to perform. The possible directives are listed in Table III. See Figure 5-2 for the structure of the global array processor routines.

TABLE III
GLOBAL ARRAY DIRECTIVES

DIRECTIVE	RESPONSE
0	Retrieve the value associated with the specified global array reference.
1	Store/Replace the value associated with the specified global array reference.
2	Request the \$NEXT function be performed.
3	Delete all subkeys in the B-tree global array node file whose key value is equal to, or is the prefix of, the passed global array reference.
4	Request the \$DATA function be performed.

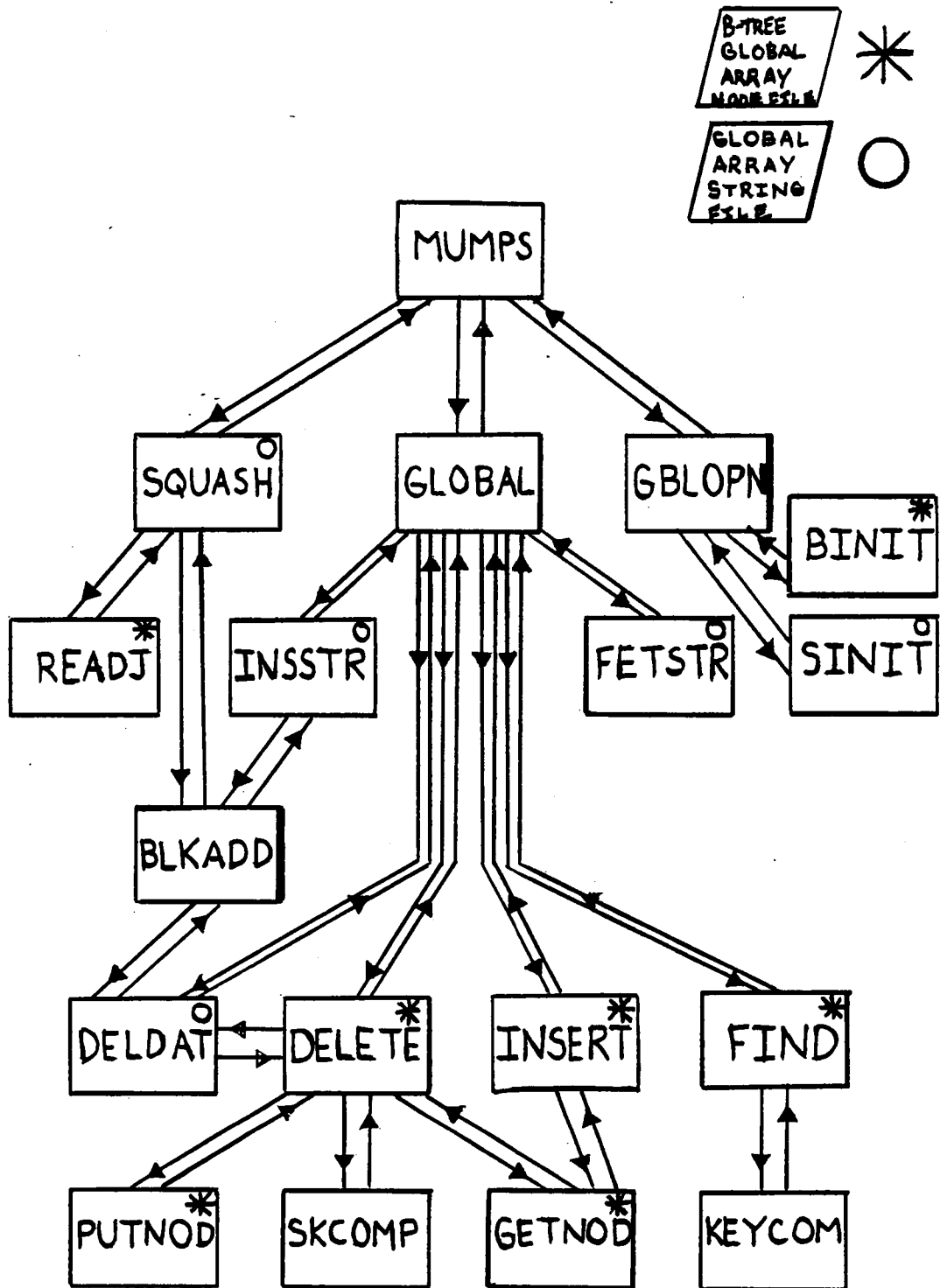


Figure 5-2. The Global Array Processor Process

Retrieval of a Global Array Reference

The retrieval process begins with a call to the function FIND, passing the global reference key length and a command of one. Function FIND utilizes an M-way tree search in the attempt to locate the specified global array reference. FIND is supported by another function named KEYCOM. This function compares the passed global array reference key with that of a subkey inside a B-tree node block. If an exact match is not found, an attempt is made to locate a subkey whose key is prefixed by the passed key. If a match still is not discovered, FIND is set to zero, and control is passed back to GLOBAL. If a match is realized, then the B-tree node block and the offset of the located subkey in the B-tree node block are passed back to GLOBAL with FIND set to one.

If FIND equals zero upon return, the global array reference was not in the B-tree global array node file. GLOBAL posts an unsuccessful lookup flag, and control is returned to MUMPS.

If FIND equals one upon return, the global array reference was found. The next step is to check if the global array reference refers to a subkey, or whether the global array reference refers to a node that exists as a path node; a path node is a node that would have been conceptually created to facilitate a path between the root

subkey and the inserted subkey. This is determined by comparing the length of the passed global array reference with the length of the located subkey's key. If the lengths are equal, then an actual subkey was found and further processing may be done. If the passed global array reference length is shorter than the located subkey's key, then a path node was found. If a path node was found, an undefined value is associated with the global array reference.

If a subkey is located, the data type byte of the subkey is checked to determine what data type this subkey has associated with it. If the subkey indicates a numeric value, the first four bytes of the subkey are extracted as a REAL*4 number. If the subkey's data type byte indicates a character string value is associated with the global array reference, a call to subroutine FETSTR, using the first four bytes of the subkey as a string pointer, is made to retrieve the associated character string. If the data type byte indicates a NULL value is associated with the global array reference, no information is extracted from the first four bytes of the located subkey. GLOBAL then returns control to MUMPS with the data value and data type loaded into their appropriate communication vectors. See Appendix E for an explanation of the global array processor communication vectors. For the structure of the retrieval process, see

Figure 5-3.

Store/Replace a Global Array Reference

In order to determine whether the global array reference passed from MUMPS will replace an existing data value, or will create a new subkey, a call to function FIND is made with the passed global array reference key length and a command of two. If FIND returns zero, no subkey's key matched exactly with the passed global array reference key. The global array reference will have a subkey created and inserted into the B-tree global array node file. If the function returns a value of one, the old data associated with the global array reference will be replaced by the new value associated with the current global array reference.

B-tree Global Array Node File Insertion

If a new global array reference is to be inserted into the B-tree global array node file, a subkey must be created to "house" the new global array reference. If the global array reference has a numeric value associated with it, the passed REAL*4 numeric is transferred to the first four bytes of the new subkey, and a code of one is transferred into the data type byte of the subkey. If the global array reference has a character string value associated with it, a call is made to subroutine INSSTR to insert the passed character

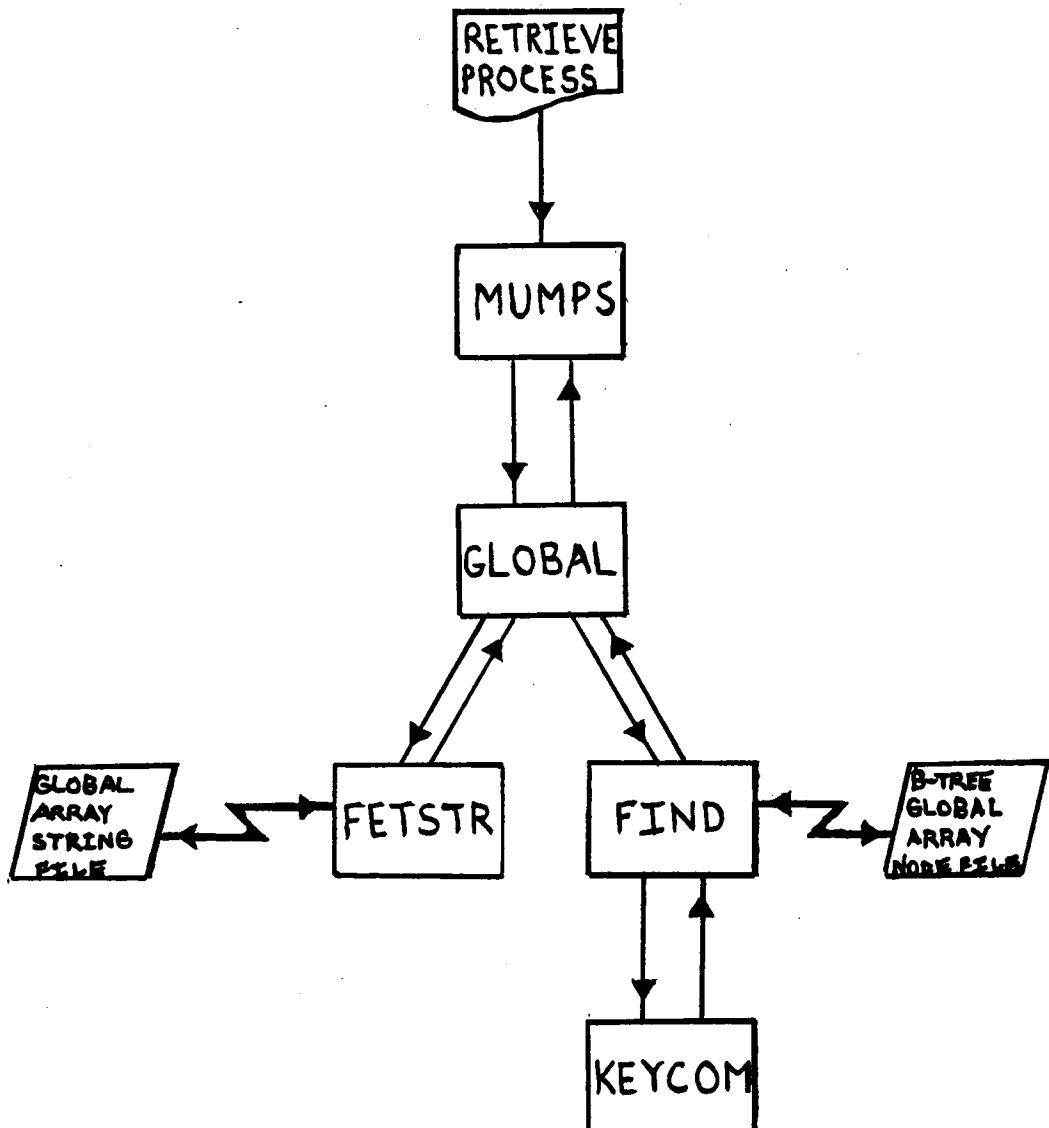


Figure 5-3. The Global Array Retrieval Process

string into the global array string file. The returned pointer to the character string is placed into the first four bytes of the subkey, and a code of two is placed into the data type byte of the new subkey. If no value is to be associated with the global array reference, the first four bytes of the subkey are irrelevant; a code of four transferred into the data type byte is the only data value information carried in the new subkey. The global array reference subscripts are then placed into the subkey. The key length byte is set to the number of global array reference subscripts which were combined together to form the new subkey's key. A call to subroutine INSERT is then made to insert the subkey into the B-tree global array node file. If the insertion of the new subkey would cause a B-tree node to split, a call is made to subroutine GETNOD. This subroutine dynamically allocates B-tree node blocks.

Upon return from INSERT, a check is performed to see if a new root B-tree node block was created by the insertion of the new subkey. If it did, the new B-tree root is recorded. Subroutine GLOBAL then returns to MUMPS. Figure 5-4 shows the structure of the insertion process.

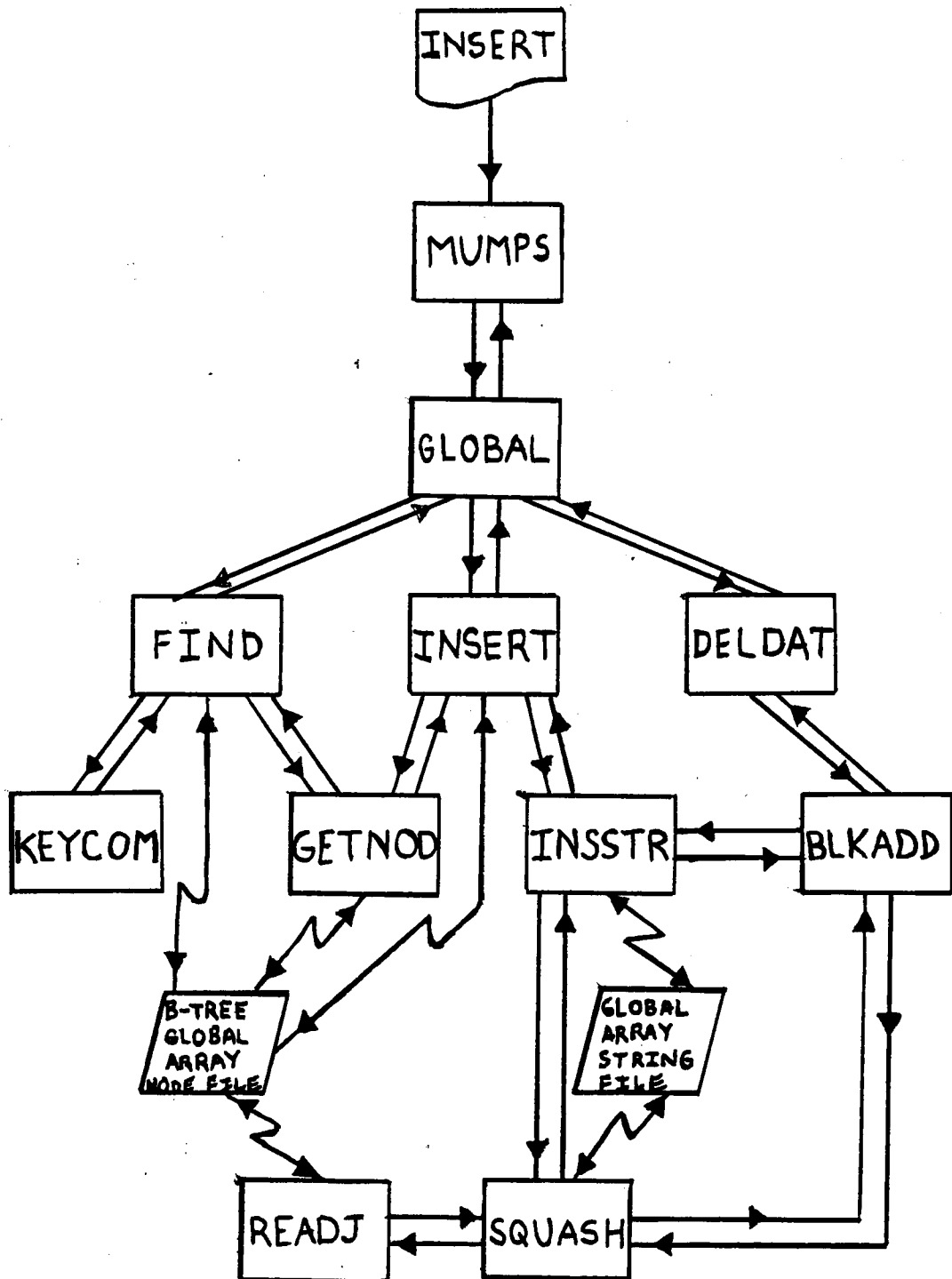


Figure 5-4. The Global Array Insertion Process

B-tree Global Array Node Value Replacement

If the global array reference is found, then a new value will be associated with that reference. The data type byte of the located subkey is checked for a code of 2, meaning the located global array reference has a character string value associated with it. If this reference currently has a character string value associated with it, a call to subroutine DELDAT is performed with the first four bytes of the located subkey as the pointer to the character string. If the new value to be associated with this global array reference is a number, the passed REAL*4 value is moved into the first four bytes of the located subkey, and a code of one is moved into the data type subkey byte. If a character string value is to be associated with the global array reference, a call to INSSTR is made to insert the passed character string into the global array string file. The pointer returned from this call is inserted into the first four bytes of the located subkey with a code of two moved into the data type byte of the subkey. If a NULL or undefined value is to replace the current global array reference value, a code of four is transferred into the data type byte of the subkey. After the subkey has been adjusted to contain its new value, the B-tree node block containing the located subkey is rewritten, and process is returned from GLOBAL to MUMPS.

(NOTE: If a character string value that is to be associated with a global array reference will not fit into the global array string file, regardless of string file compression, the character string value will be substituted with a NULL value.)

The \$NEXT Function of the Global Array Reference

This directive requires the special use of the function FIND. The function FIND is called with the global array reference key length and a command of three. Function FIND first attempts to locate either a subkey that matches the passed global array reference key, or a subkey whose key is prefixed by the passed global array reference key. IF FIND cannot locate a subkey which is not at least prefixed with the passed key, FIND is set to zero indicating \$NEXT failure. If a match is found, the B-tree global array file is again searched until the passed key is no longer a prefix to a stored subkey key. If the two keys differ in the last subscript, FIND is set to that subscript's key position and the search is again continued to locate the smallest key differing in the last compared subscript. If the keys differ in any other subscript, control is returned to global.

If FIND returns a value of zero, then there is no \$NEXT for the given global array reference, a failure indication

is posted, and GLOBAL returns to MUMPS. If FIND returns a value greater than zero, then the global reference has a \$NEXT value. The contents of the subkey's key at that subscript position is the \$NEXT value. Global posts a successful indication flag, then returns to MUMPS. Figure 5-5 shows the structure of this process.

Deletion of a Global Array Reference

This directive will cause the removal from the B-tree global array node file, every subkey whose key value matches in total or is prefixed by, the global array reference key passed from MUMPS. This process is accomplished by multiple iterations of the following steps. First a call is made to function FIND with the passed global array reference key length and a command of four. If FIND is equal to one, subroutine DELETE is called to delete the located subkey. If the subkey to be deleted has a character string associated with it, a call to subroutine DELDAT is made to delete the string referenced by the first four bytes of the located subkey. Anytime a subkey is removed from, or is redistributed to another B-tree node block, the subroutine SKCOMP is invoked to compress the space utilized by the removed subkey. As B-tree node blocks become empty, subroutine PUTNOD is called to dynamically deallocate those blocks. When the last subkey is removed from the B-tree

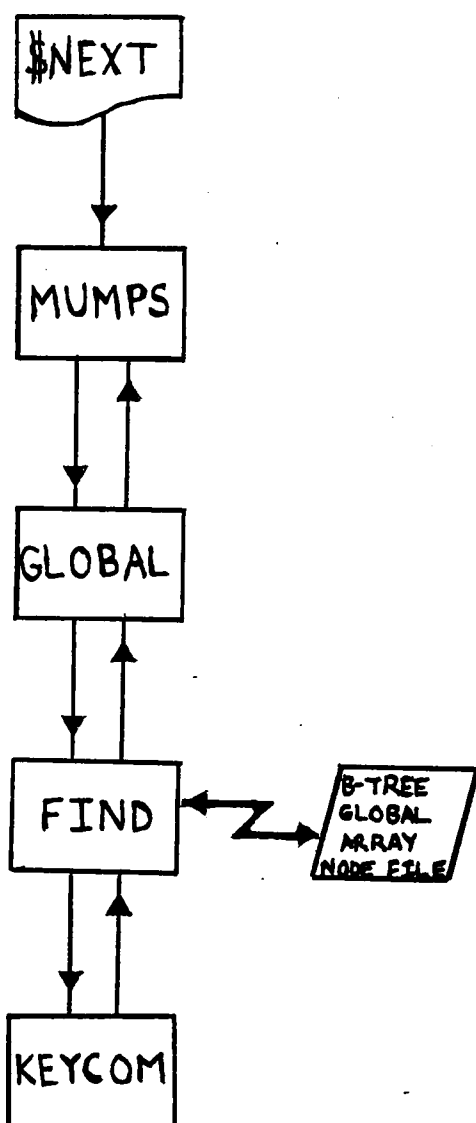


Figure 5-5. The Global Array \$NEXT Function Process

global array node file, a special call to subroutine GETNOD is made to reestablish the existence of a B-tree root node block.

Upon return from DELETE, a check is made to determine whether the subkey's deletion caused a new B-tree root node to be formed. If there is a new root node, the pointer to the head of the B-tree global array is updated. The deletion process continues until all the subkeys that match exactly or are prefixed by the passed global array reference key, are removed from the B-tree global array node file. Figure 5-6 shows the structure of the deletion process.

The \$DATA Function of the Global Array Reference

This function is performed by special use of the function FIND. A call is made to function FIND, passing the global array reference key length and a command of five. FIND first attempts to locate the subkey which exactly matches the global array reference. If unsuccessful, the search is reinitiated, this time searching for a subkey whose key is prefixed by the passed global array reference key. If a key match can not be attained, there is no \$DATA for global array reference. FIND is set to zero, and control is returned to GLOBAL. If a match is realized, and the global array reference has a value associated with it, the "ones" digit of the \$DATA code is set to one. The

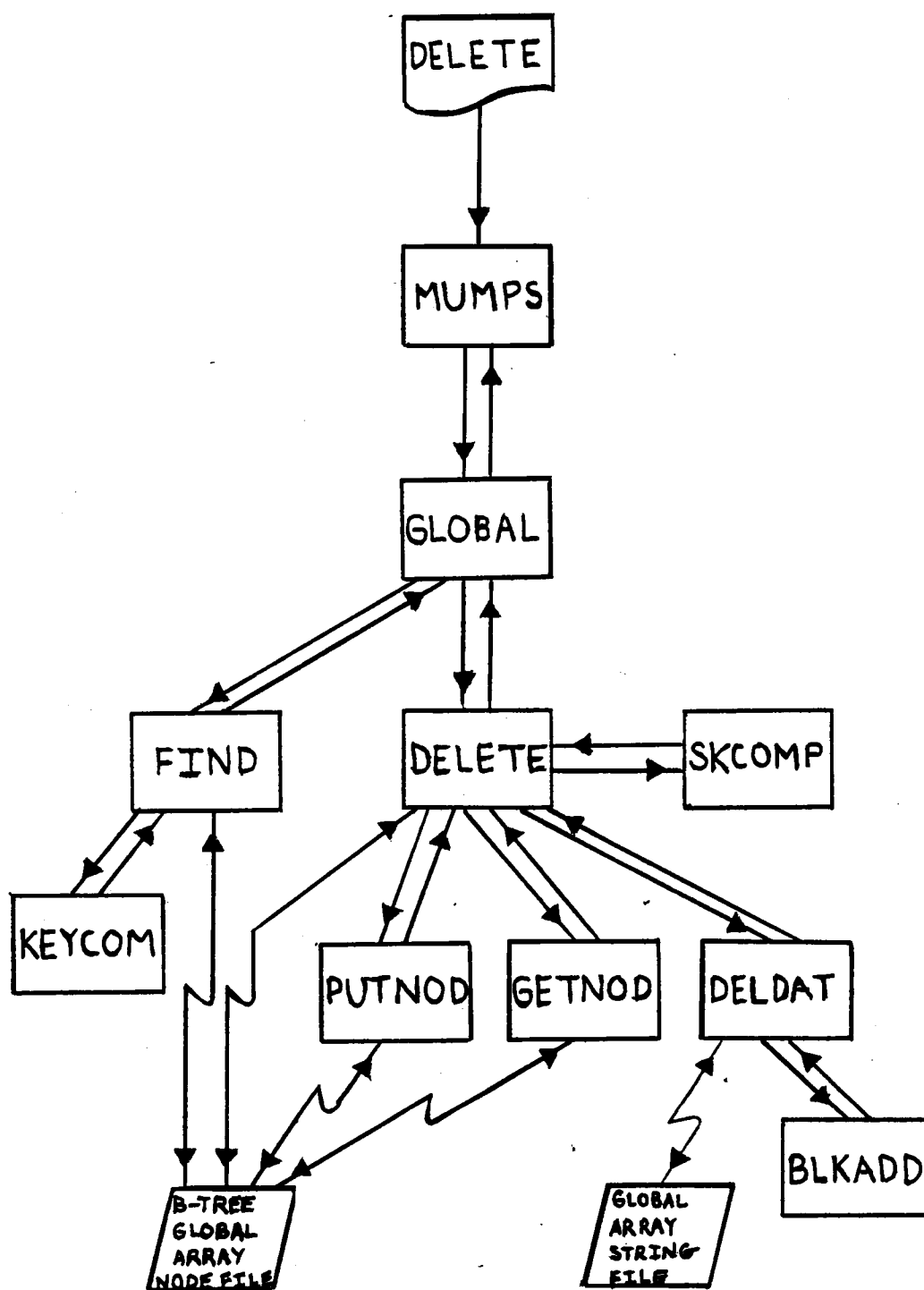


Figure 5-6. The Global Array Deletion Process

search is again initiated, this time for a subkey whose key is prefixed by the passed global array reference key. If such a subkey is found, the "tens" digit of the \$DATA code is set to one. The final step is to set FIND to one and return to GLOBAL. Figure 5-7 shows the structure of this process.

The B-tree Global Array Node File Dynamic Storage Management

In a dynamic environment where the storage requirements are constantly changing, there should be an efficient mechanism to manage file storage. This project utilizes a bit map to determine the availability of a block of storage. Advantages to this technique are:

- 1) The source code complexity involved with storage management is reduced to a series of simple comparisons.
- 2) Many blocks of storage may be handled with little file space overhead.
- 3) The simplicity of code combined with the small file space overhead, leads to rapid storage management.

The first "N" 512 byte blocks of the B-tree global array node file are used as index blocks. See Figure 4-3 on page 23, for the format of the B-tree global array node

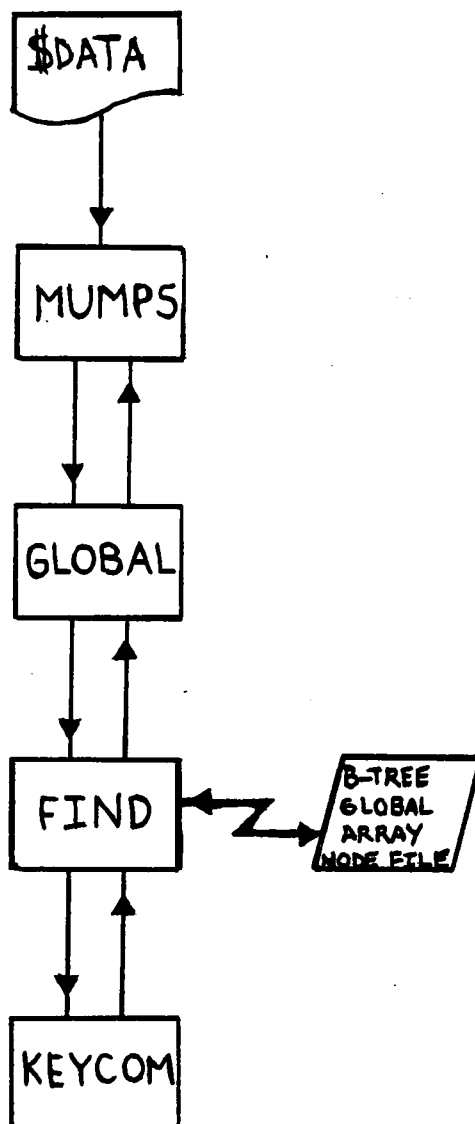


Figure 5-7. The Global Array \$DATA Function Process

file. Subroutines GETNOD and PUTNOD handle the dynamic storage. As a B-tree node block of storage is requested, a call is made to subroutine GETNOD. GETNOD determines the availability of a file block by testing the individual bits of the index block bytes. The first zero bit encountered has its corresponding file block allocated and the index bit set to one.

The reverse of this process is done to deallocate a file block. When a block is to be deallocated, subroutine PUTNOD is called. PUTNOD, from the passed file block number, determines the index bit that maps to the file block in question. That bit is then reset to zero, and the reinitialized block is returned to the pool of unallocated file blocks.

Global Array String File Maintenance

During MUMPS execution, many global array references will have character string values assigned to them. The end-of-file mark points to the next available byte in which data may be stored. Each time a character string is inserted into the global array string file, the end-of-file mark moves toward the bottom of the global array string file. Regardless of the number of character strings deleted, this marker will not be moved backward toward the top of the global array string file. The only exception to

this rule is if the deleted character string is physically the last character string in the global array string file. In this case the end-of-file mark is moved to the starting byte of the old string.

In any dynamic environment, after a number of insertions, the end-of-file mark will reach the bottom of the string file. When this happens, no more strings will be able to be placed in the string file, regardless of the number and size of the "holes" left behind as character strings are deleted. This phenomenon is called "fragmentation". The only cure to this problem is to periodically compress the global array string file, reclaiming the "holes" in order to process more strings.

The global array processor has two routines that perform string compression, SQUASH and READJ. Subroutine SQUASH is responsible for compressing the individual global array string file blocks. The other subroutine, READJ, is responsible for relinking the old string file addresses to the new string file addresses in those B-tree global array subkeys that have character string values associated with them.

These compression routines would be called in two instances:

- 1) The MUMPS user issues an interpreter directive to compress the string files.

- 2) Subroutine `INSSTR`, attempts to insert a character string into the string file that has the free space to accommodate the string, but the end-of-file mark is positioned such that there is no room between it and the bottom of the file.

Figure 5-8 shows the structure of the global array string file maintenance process. Appendix F explains the string file compression algorithm.

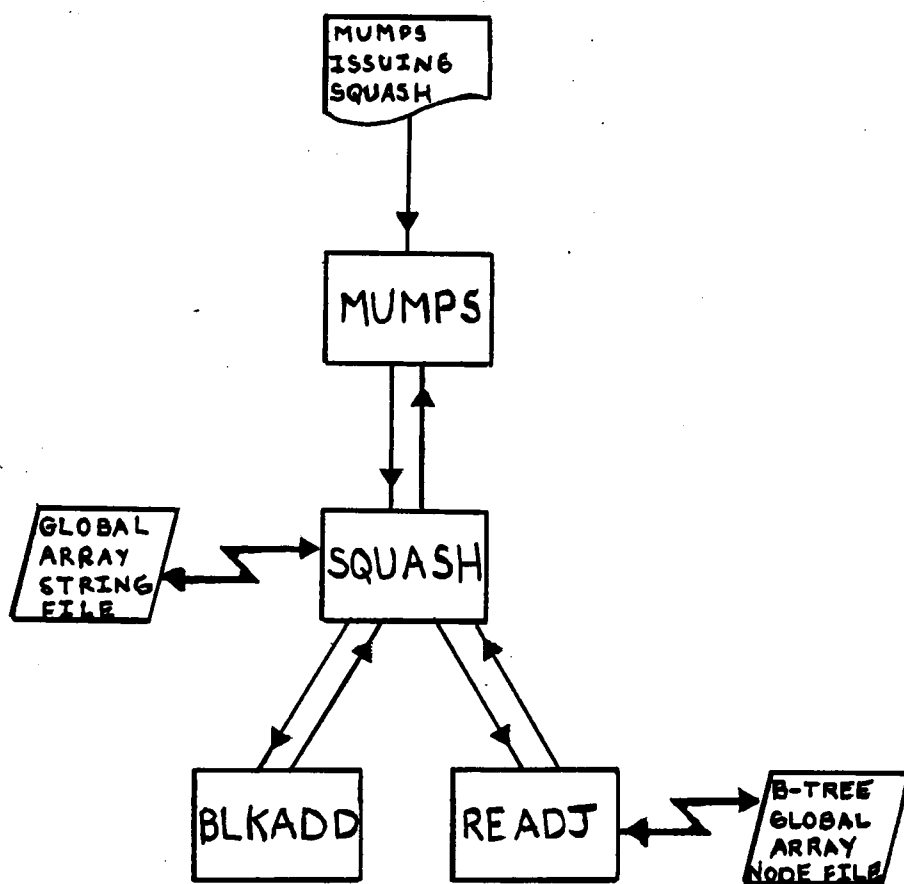


Figure 5-8. The Global Array String File Compression Process

CHAPTER VI

PERFORMANANCE ATTAINED

B-tree Search Time

The search time to access the global array references is affected by two factors; the number of subkeys per B-tree node block and the height of the B-tree. The first is the number of subkeys contained in a B-tree node block. The amount of overhead to process an individual B-tree node block is affected by the number of key comparisions, and the length of the subkey link list chain. For a node with a small number of subkeys, the overhead in processing the node is not a major timing factor.

The height of the B-tree is the second, more important timing factor. The height of the B-tree is defined as the maximum level or the length of the path from the root node to a leaf node. The height is important since the maximum number of disk accesses needed to determine the absence of a key is equal to the height of the tree.

A tree is said to be height balanced if, for any node in the tree, the height of the left subtree does not differ from the height of the right subtree by more than a value of 1 [Knuth,1976]. G.M. Adel'son-Vel'skii and E.M. Landis [Adel'son-Vel'skii,1962], have shown that in height balanced trees searches, inserts, and deletes can be processed in

time $O(\log N)$, where N is the number of nodes in the tree. Given a B-tree of order M with N nodes, there are $N+1$ possible failure nodes; failure nodes are conceptually present to indicate an absence of a global array reference. It follows that the number of nodes on levels 1, 2, 3,.... is at least 2, $2*[M/2]$, $2[M/2]^2, \dots$; this means the number of failure nodes at level L is greater than or is equal:

$$2 * [M/2] ^ L - 1$$

By reversing the equation, the height of the B-tree, or the maximum number of disk accesses to determine the presents of a global array reference, is less than or equal to

$$1 + \log[M/2] ((N + 1) / 2)$$

The disk access time, multiplied by the result of the equation above, is added to the B-tree node block subkey overhead to determine the time needed to locate a global array reference in the B-tree global array node file.

B-tree Insertion Time

Regardless of any other process, all subkey insertions require the overhead of the B-tree search routines to locate the place in the B-tree global array in which the new subkey is to be inserted. The additional overhead is of two types; the motion between subkeys and the disk I/O involved with node division and file header maintenance.

The best time the insertion routine can produce is the case when a subkey may be accommodated into a B-tree node block without node splitting. In this case, the overhead involved merely includes the transferal of the passed subkey into a B-tree node block and an I/O to update that B-tree node block back into the file.

The worst case involves the insertion of a subkey that causes multiple node divisions, with the node division propagating to the root and causing a new root to be formed. This process involves disk I/O to request dynamic space, to refresh updated B-tree node blocks, and to update the B-tree global array node file header. This process also involves the necessary time to transfer subkeys from one B-tree node block to the newly allocated brother node block. When the number of subkeys transferred is small, the time to transfer keys is negligible.

B-tree Deletion Time

The deletion process requires the most time to execute because of the amount of disk I/O which needs to be performed. Regardless of the amount of time required in deleting a subkey, the deletion process has the amount of time accumulated in locating the specified subkey as part of its total processing time. The deletion algorithm, explained in Appendix C, has four phases in which different

times can be accrued; there is the time to delete a subkey and to recompress the node which contained the subkey, the time to exchange the deleted subkey with another subkey, the time to perform the catenation process, and the time to perform the underflow process.

The first phase accumulates time as (a) subkeys are removed from a node, and (b) nodes are subsequently recompressed. The time required to delete a key is small since it requires only the subkey link list to be modified. The recompression process requires the time to copy all the allocated subkeys from one node into a temporary node and then to copy the compressed node back into the permanent node. This phase requires very little time because it is CPU bound, with no external I/O involved.

The subkey exchange phase requires multiple disk I/O. It will read a path of B-tree node blocks from the current level in the tree to the level in the tree which contains the leaves. After locating the subkey to exchange, this phase then accumulates the time to delete the selected subkey, to replace it with the smallest subkey of the leaf node, to remove the exchanged subkey from the leaf node, and to recompress both nodes. Finally, this phase requires one additional I/O to update the B-tree global array node file with the revised node that originally contained the deleted subkey.

The catenation phase requires 4 disk I/O operations in addition to the I/O required to update the file header and to dynamically free unused nodes. The first I/O is required to retrieve the parent node of the node which contains the subkey to be deleted. The second I/O is required to retrieve the adjacent brother node of the node which contains the subkey to be deleted. The third and fourth I/O operations update the B-tree global array node file with the processed parent and catenated nodes. In addition to the disk I/O operations, time is accumulated from the recompression of those nodes which had subkeys removed and the time required to transfer subkeys from one node to another. This process can occur multiple times from the deletion of one subkey. The number of times a B-tree will have to catenate from the deletion of a subkey is bounded by the height of the tree.

The underflow phase requires 5 I/O operations. The first is required to retrieve the parent node of the node which contains the subkey to be deleted. The second I/O operation is required to retrieve the adjacent brother node. The third, fourth, and fifth I/O operations are required to update the processed nodes in the B-tree global array node file. In addition to the required I/O, the underflow process accumulates time from the removal of subkeys and the required node recompression. Also time is accumulated as

subkeys are transferred from one node to another. This process does not occur multiple times from the deletion of one subkey.

Variable Length Keys Vs. Fixed Length Keys

This implementation of the MUMPS global array processor handles global array references with between 1 and 13 indices. The global array processor attempts to make the best utilization of the global array file space. To allocate 64 bytes for a global array reference that only needs 32 bytes to store the information, wastes 32 bytes per global array reference. From Table II on page 34, handling a B-tree of the smaller sized subkeys allows 3,834 more subkeys to be stored, in the extreme case, than the same B-tree if it consisted of the larger subkeys.

As long as the global array references have homogeneous subkey lengths, the resulting tree will be a true B-tree. When global array references are allowed to vary in key length, the resulting tree which handles the global array references no longer is a B-tree with respect to the number of subkeys held in a particular node. But the resulting tree is still height balanced, and the same global array maintenance algorithms still apply.

In terms of tree node space, the resulting tree may be considered a B-tree. Each block, except for the root block,

contains between $(\lceil M_{\max}/2 \rceil - K_{\max})$ and $(M_{\max} - K_{\max})$ bytes of subkey information, where $M_{\max}$ is the order of a B-tree which consists entirely of the largest subkey key length currently contained in the tree structure, and $K_{\max}$ is the length of the largest subkey currently in the tree structure. The only additional overhead involved with varying length global array references is the time it takes to decide the absence of a global array reference in the worst case. The overhead time mentioned above refers to the internal tree node subkey comparison time. Overall search time may be better, since each tree node block might contain more subkeys than a regular B-tree node. This could lead to a shorter balanced tree requiring fewer disk accesses.

Conclusion and Evaluation

The final project is the culmination of three project designs. The first design incorporated multiple B-tree structures. With this design, the number of subscripts describing a global array reference were "unlimited". Each level of subscripting would have had a separate B-tree structure. The flaw with this design was that it would have fragmented disk space, required extensive source code overhead, and increased data acquisition time.

The second design eliminated the multiple B-tree concept. This version incorporated a single B-tree

structure to manage global array references. This design concatenated the passed global array reference subscripts to form the key to the global array reference information. The limitation imposed in this version restricted the user to 13 subscripts in referencing global array information. A bigger limitation required the storage of a fixed length key, regardless of the actual number of subscripts. This limitation would have caused an underutilization of disk space.

The current implementation is an extension to the second design. The implemented version better utilizes disk space by only storing the specified subscripts without the wasteful padding involved with fixed length keys. The added source code involved to handle the varying length keys is minimal and causes no degradation in data acquisition time. Other improvements include a bit map dynamic storage management technique. This technique improves processing speed, as well as, reduces source code and disk space overhead.

As with most research projects, there is always the possibility for improvements. This project, for example, does not allow alphabetic global array references. The addition of this feature would make the global arrays more flexible. Also, this project only permits single precision real values to be associated with a global array reference.

This restriction limits the user to numbers of only 7 digits of accuracy. An improvement would be to modify the project to handle double precision real values. Even with these restrictions, the author is confident the stated goals of the project were met.

LIST OF REFERENCES

LIST OF REFERENCES

- (1) Adel'son-Vel'skii, G. M., and Landis, E.M.
"An Information Organization Algorithm". DANSSSR,
146, (1962), 263-266.
- (2) Bayer, R., and McCreight, E. M. "Organization
and Maintenance of Large Order Indexes". Acta
Informatica., Volume 1., (1972), 173-189.
- (3) Greenes, R. A., Pappalardo, A. N., Marble, C. W., and
Barnett, G. O., "A System for Clinical Data
Management"., Fall Joint Computer Conference, 1969.
(1969), 297-305.
- (4) Horowitz, E., and Sahni, S., Fundamentals of Data
Structures., Fifth Edition. Potamic:
Computer Science Press, Inc., 1976.
- (5) Knuth, D. E., The Art of Computer Programming: Sorting
and Searching. Reading: Addison-Wesley., 1976.
- (6) NBS Handbook 118: MUMPS Language Standard.
Washington D.C.: U.S Department of Commerce,
National Bureau of Standards., 1976.
- (7) Sammet, J. E., Programming Languages: History and
Fundamentals., Englewood Cliffs:
Prentice-Hall, Inc., 1969.

APPENDICES

APPENDIX A

B-TREE SEARCH ALGORITHM

The B-tree is an extended version of the M-way tree. Each node in the B-tree structure has the following format:

NODE P: M, L0, A0, (K1, A1, L1), ... , (Km, Am, Lm)

M is an INTEGER*2 value which contains the number of allocated subkey bytes (number of subkeys) in the B-tree node block. The LI pointers contain INTEGER*2 values which point to the next subkeys in the node; a pointer value of 0 ends the link list; L0 points to the first subkey in the B-tree node block. The AI pointers contain INTEGER*4 values that point to the current node's descendant nodes; A0 points to the current B-tree node block's left descendant. The KI values contain the REAL*4 values that make up the global array key reference. See Figure A-1 for a graphic explanation of the B-tree node blocks.

A search for a given key, X, begins with the B-tree head node read into memory. Index links, LI, are followed in ascending order until (a) X is found, (b) the search key meets a key less than X, or (c) the search key is found greater than the largest key in a particular node. If the search key is equal to X, then the routine returns a successful indication, the B-tree node block, and a pointer to X in the B-tree node block.

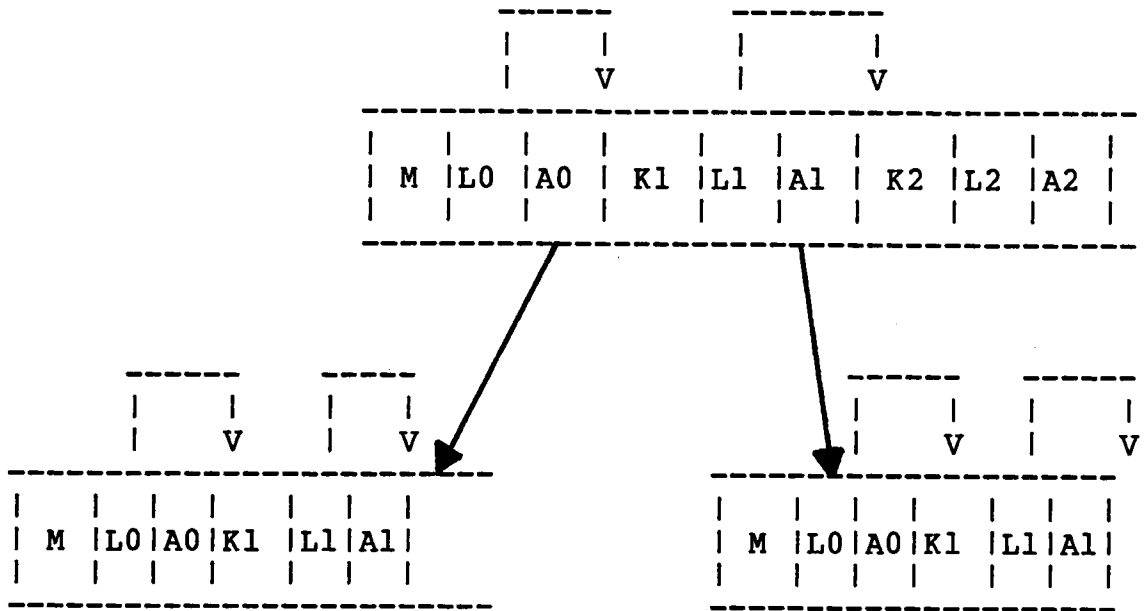


Figure A-1. The Nodes of An M-way Search Tree

If the search key is less than a key in a particular node, the previous key's descendant pointer, AI-1, is followed to a lower node where the same process again occurs; else if the search key is less than a key in a particular node and there is no descendant pointer, a failure indication is returned along with the last node read and the position in that node where the new key could be inserted. If the search key is greater than the largest key in a particular node, then the right descendant pointer is followed to the next level; else, if no descendant pointer exists, a failure indication is return along with the last B-tree node read and the position to insert X.

For example, if one wished to locate key 3 in Figure A-2, node A is first read into memory. The routine would check to see if 3 was less than 2. It is not, so it will link to the next key in node A or key 7. This time key 3 is less than key 7 which causes the node pointed to by key 7's descendant pointer (node C) to be read into memory where processing will begin again. The search is again applied to node C where after one comparison, Key 3 is found.

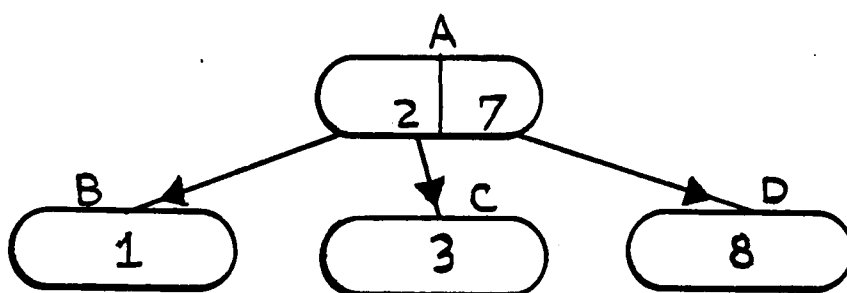


Figure A-2. A Stable B-tree Before a Key Insertion

APPENDIX B

B-TREE INSERTION ALGORITHM

The problem with inserting a key into a B-tree is that the algorithm must maintain the B-tree structure. New keys may be inserted only into the leaf nodes of the B-tree. All internal key repositioning will start at the leaf in which the new key is inserted, and propagate upward toward the root.

For example, let the order M of the tree be 3, so every node in the B-tree, except the root, will contain at least 1 key and no more than 2. Given the stable B-tree in Figure A-2, key 5 is to be inserted into the tree. The first task is to search the B-tree and locate the leaf node and the key after which the new key may be inserted. In the tree, node C is located and the insertion place is noted after key 3. The next step is to insert the key into node C to form the stable B-tree shown in Figure B-1. The tree is stable since all of the nodes conform within the B-tree specified rules.

Now given the stable B-tree in Figure B-1, key 4 is to be inserted. Again node C is located with the place to insert marked between keys 3 and 5. The new key 4 is then inserted into node C. From observation, the resulting tree in Figure B-2 is unstable because node C contains more than 2 keys. This condition will force the node to split into

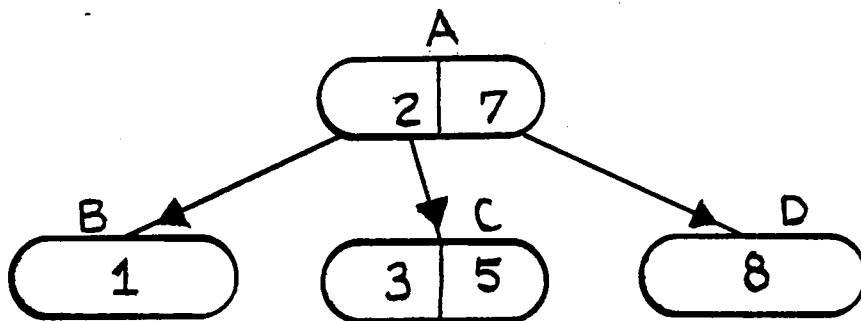


Figure B-1. A stable B-tree After the Insertion of Key 5

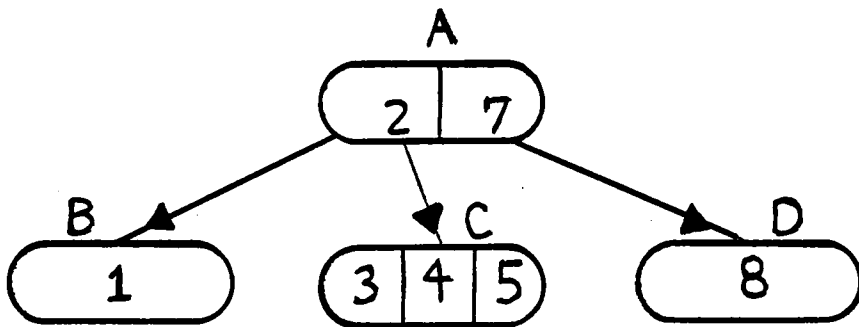


Figure B-2. An Unstable B-tree After the Insertion of Key 4

halves. This split will cause a new node to be dynamically allocated to the B-tree structure. The split forms two nodes with the following formats:

NODE P: $M/2-1, L_0, A_0, (K_1, A_1, L_1), \dots, (K_{M/2-1}, A_{M/2-1}, L_{M/2-1})$

NODE P': $M-M/2, L_{M/2}, A_{M/2}, (K_{M/2+1}, L_{M/2+1}, A_{M/2+1}), \dots, (K_M, L_M, A_M)$

The remaining "middle" key value, $K_{M/2}$, and the new allocated node address forms a new index tuple, $(K_{M/2}, P')$, which will be inserted into the parent node.

In the example, key 3 remains in node C, key 5 is transferred into the newly allocated node E, with key 4 and the address of node E extracted to be placed in the parent node A. A search of the parent, node A, is made to determine where the extracted "middle" key from the node split will be placed. The "middle" key information is then inserted into the parent. Figure B-3 shows the unstable tree after the extracted subkey is inserted into the parent node.

Again the B-tree is unstable because node A has more than 2 keys. The splitting process reoccurs with the "middle" key 4, propagating above the head node. When this special case arises, a second node is dynamically allocated, the "middle" key inserted, and the new node becomes the new head of the B-tree. Figure B-4 shows the resultant tree with a new root node F.

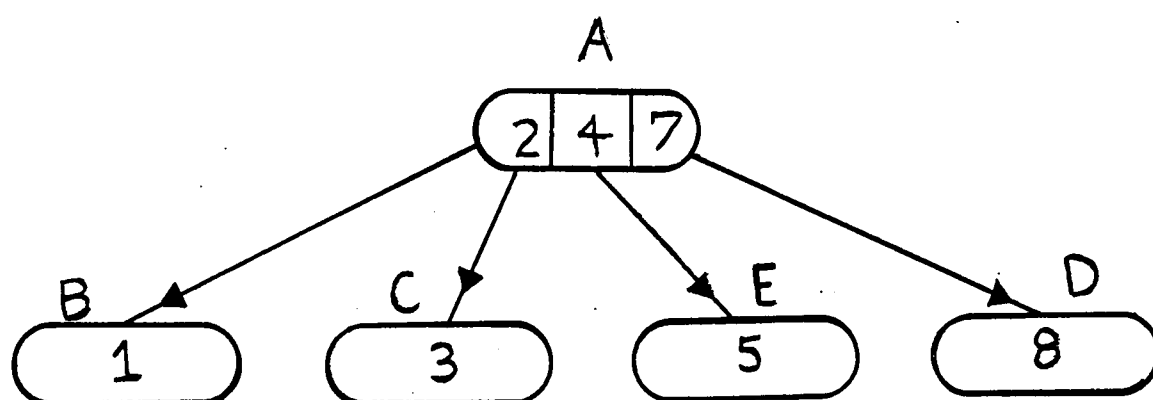


Figure B-3. An Unstable B-tree After a Node Split

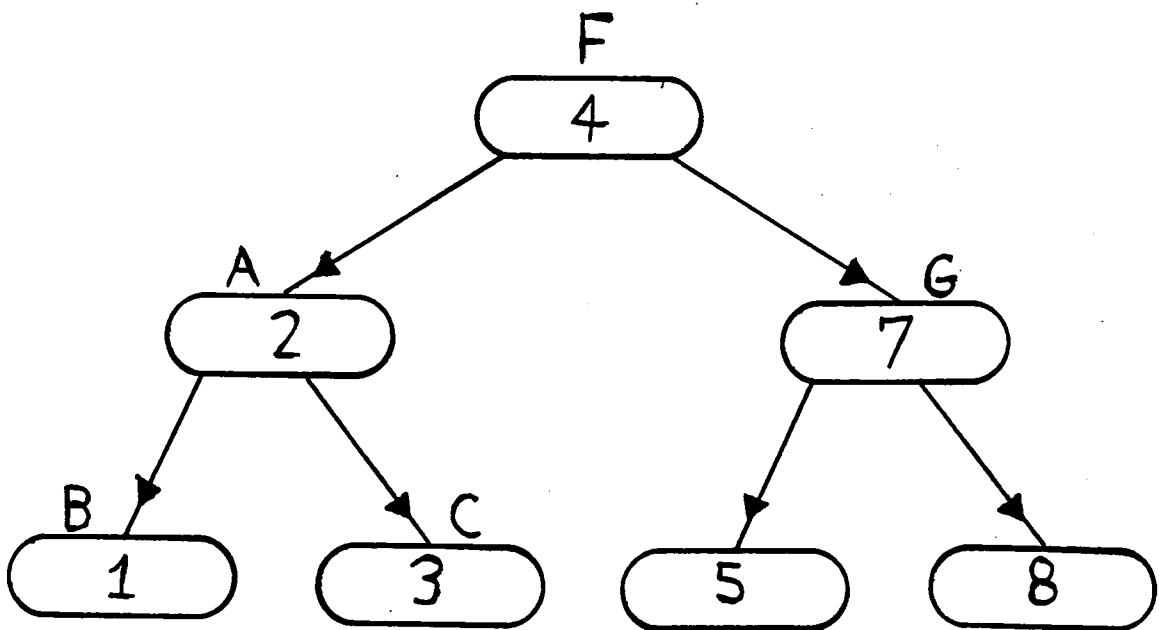


Figure B-4. A Three Level Stable B-tree

APPENDIX C

B-TREE DELETION ALGORITHM

In any dynamic environment, it must be possible to remove previously stored information. The task of deleting a key from the B-tree must not only remove the key, but also maintain the B-tree structure.

The first step is to locate the key to be deleted, say X. If X is currently resident on a leaf page, denoted by the absence of a descendant pointer, then X is simply removed from that page. Unlike the insertion routine, keys may be deleted from places in the B-tree other than the leaf nodes. If this is the case, the first step is to descend the subtree pointed to by AX. The next step is to follow all the A0 descendant pointers until a leaf node is reached, say node L. The next step replaces X with the smallest key in L, and removes the smallest key from L. As a consequence, L may contain fewer than $\lfloor M/2 \rfloor - 1$ keys and appropriate action must be taken to regain a stablized B-tree structure. The two possible corrective actions are the catenation and the underflow (key redistribution) processes. These processes occur between L and an adjacent brother; node Y being an adjacent brother of node L if the parent node has a key, called the parent key (pk), such that the descendant pointers on either side of the parent key

point to nodes L and Y.

The Catenation Process

The catenation process, (the combination of nodes L and Y), will occur if together L and Y have fewer than $(M - 2)$ keys. Nodes are catenated by combining the right adjacent brother, the parent key (with its descendant pointer replaced by the A0 pointer of the left adjacent brother), followed by the key values of the left adjacent brother. The parent key is then deleted from the parent node with the previous parent key's descendant pointer set to the newly catenated node. Finally, the useless left adjacent node is dynamically set free into the pool of available B-tree nodes.

As a consequence of deleting the parent key from the parent node, it is possible that the parent node will contain fewer than $\lfloor M/2 \rfloor$ keys such that special action must be taken for the parent. This process, like the insertion process, may propagate upward toward the root node.

The Underflow Process

The underflow process performs a simple redistribution of keys among two adjacent brothers and the parent key. Underflow occurs when node Y has greater than or equal to $\lfloor M/2 \rfloor$ keys. If node Y is the left adjacent brother, the

parent key (with A0 of Y replacing Apk) is inserted at the head of node L. The largest key in node Y replaces the old parent key (replacing A0 of Y with A1 of Y, then moving the address of Y into the new Apk), and the largest key in node Y is removed. If node Y is the right adjacent brother, the parent key (with Apk replaced by the last AI in node Y) is moved at the end of node L. Next, the parent key is replaced by the smallest key in node Y (Apk replaced with a pointer to node L), and finally the smallest key in node Y is removed. This process will not cause any further readjustments to propagate up the tree because the net change of keys in the parent node is unaffected.

For example, let Figure C-1 be a stable B-tree of order M, where M is set to 5. In other words, each node in the B-tree, except the root, will contain a minimum of 2 keys ($\lceil M/2 \rceil - 1$) and a maximum of 4 keys ($M - 1$). If one were to delete key 9 from leaf node E, the tree shown in Figure C-2 would result. This tree is unstable, since node E has only 1 key. Node E's adjacent brother is node D. Nodes D and E are catenated since node D contains $\lceil M/2 \rceil - 1$ keys. Figure C-3 shows the result after catenation has taken place; node D was released, and a key from node A was deleted causing further maintenance propagation. Catenation again occurs since node B contains only 2 keys. This time the catenation creates a new root formed by the merger of nodes A, B, and

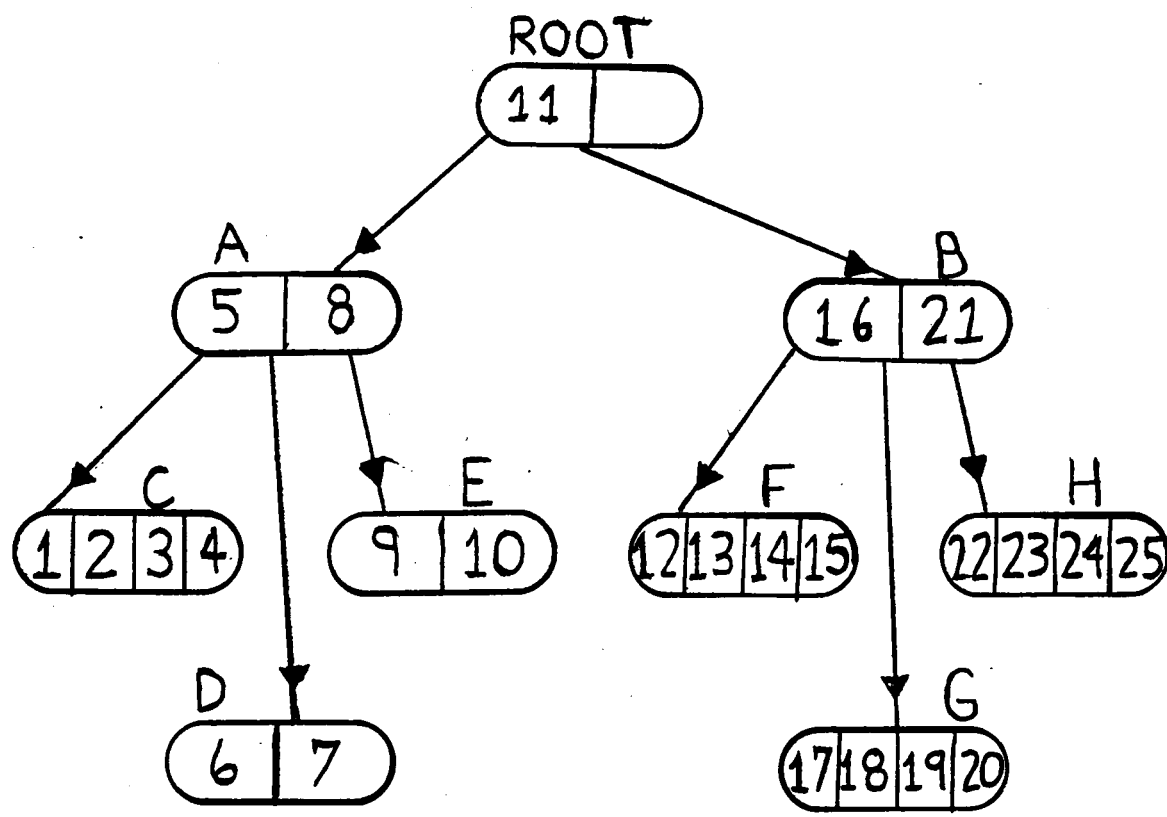


Figure C-1. A stable B-tree of Order 5

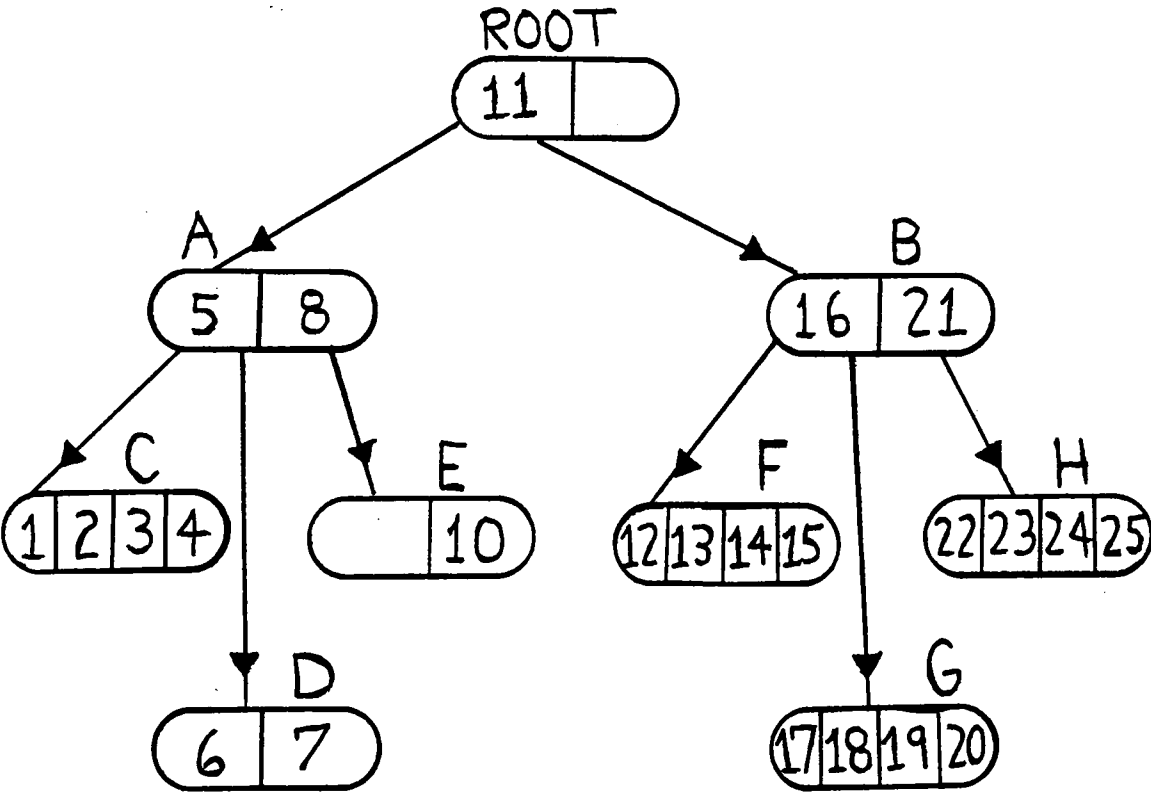


Figure C-2. An Unstable B-tree After the Removal of Key 9

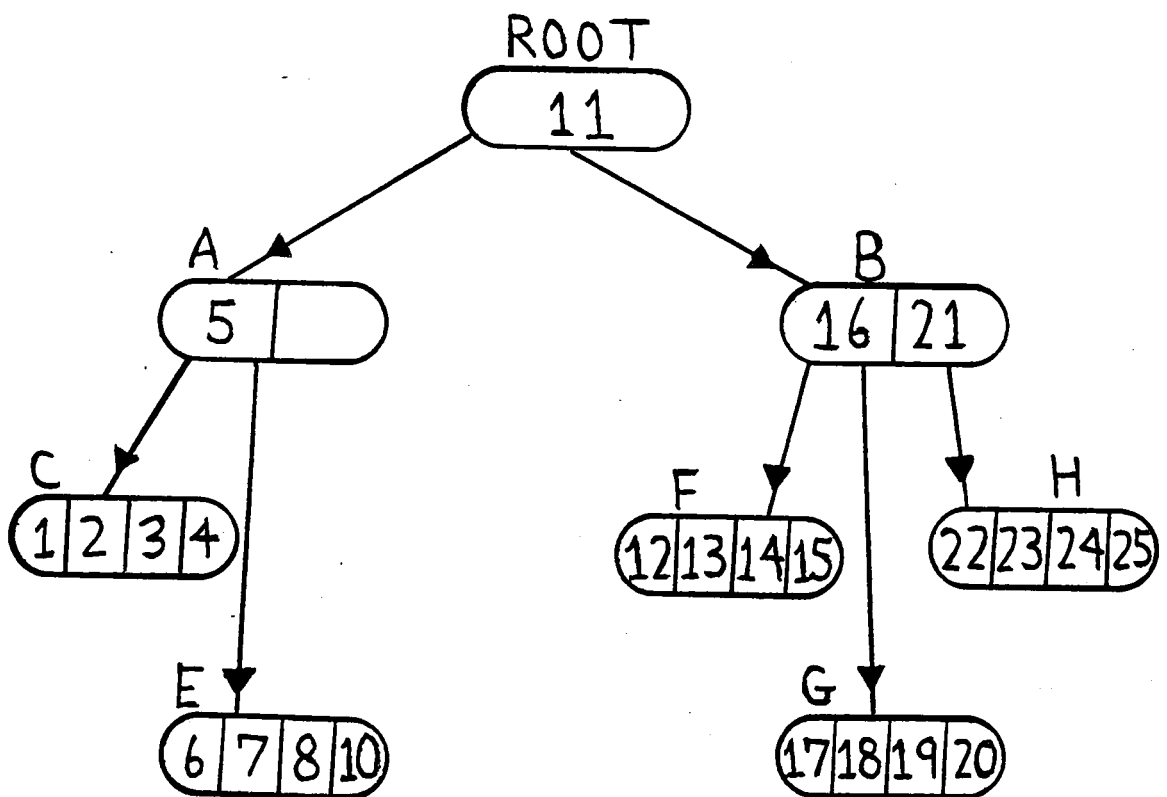


Figure C-3. An Unstable B-tree After the Catenation Process

the old root. Figure C-4 illustrates the stable balanced B-tree.

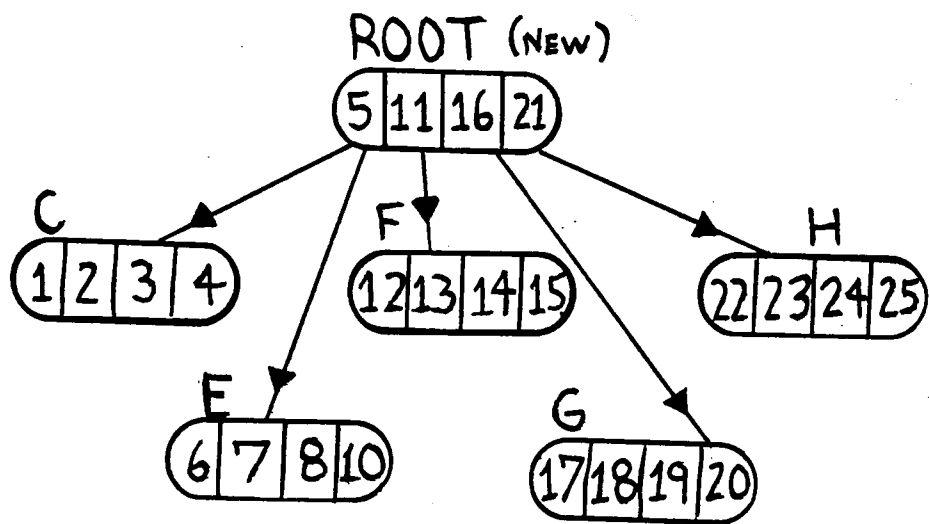


Figure C-4. A stable B-tree with the Catenation Process Creating a New Root

APPENDIX D

TABLE IV
SUBROUTINE CROSS REFERENCE

ROUTINE	ARGUMENTS	BLOCK COMMON	CALLS MADE	CALLED BY
BINIT	NONE	IO NODE1 BTREE FILES	NONE	GBLOPN
BLKADD	ADDRESS CONSTANT RESULT	NONE	NONE	DELDAT INSSTR SQUASH
DELDAT	PTR	IO STRING	BLKADD	DELETE GLOBAL
DELETE	NONE	IO NODE2 STACK SEARCH NODE1	DELDAT GETNOD PUTNOD SKCOMP	GLOBAL
FETSTR	PTR	IO	NONE	GLOBAL
FIND	LENGTH COMMAND	IO NODE1 STACK SEARCH	KEYCOM	GLOBAL
GBLOPN	NONE	IO FILES BTREE STRING	BINIT PUTS SINIT	GLOBAL MUMPS

TABLE IV (Continued)

ROUTINE	ARGUMENTS	BLOCK COMMON	CALLS MADE	CALLED BY
GETNOD	NONE	IO NODE2 BTREE FILES	NONE	DELETE INSERT
GLOBAL	NONE	IO BTREE STRING STACK NODE1 NODE2 SEARCH SUBKEY FILES	DELDAT DELETE FETSTR FIND GBLOPN INSERT INSSTR	MUMPS
INSERT	NONE	IO NODE1 NODE2 STACK SUBKEY	GETNOD	GLOBAL
INSSTR	PTR	IO STRING FILES	BLKADD FLOAT LEN PUTSTR SQUASH	GLOBAL
KEYCOM	LENGTH	SEARCH	NONE	FIND
PUTNOD	BLOCK NUM	IO	NONE	DELETE
READJ	NONE	MAP BTREE IO FILES	NONE	SQUASH
SINIT	NONE	IO STRING FILES	NONE	GBLOPN

TABLE IV (Continued)

ROUTINE	ARGUMENTS	BLOCK COMMON	CALLS MADE	CALLED BY
SKCOMP	PTR BLOCK PREV PTR LAST PTR AVAILABLE	NONE	NONE	DELETE
SQUASH	NONE	IO MAP STRING FILES	BLKADD READJ	INSSTR MUMPS

APPENDIX E

EXPLANATION OF THE COMMON COMMUNICATION VECTORS

The MUMPS global array processor communicates between MUMPS and the processor's subordinate routines mainly through the use of COMMON. COMMON is used because passing arguments involves costly overhead. The global array processor communicates with MUMPS through five vectors held in main COMMON. Table V contains the main common vectors which interface with the global array processor. Figure E-1 graphically explains the B-tree global reference vector G.

TABLE V
MAIN COMMON VECTORS

VECTOR	EXPLANATION
G	This is a 30 element INTEGER*4 array which contains the pertinent information concerning the global array reference passed from MUMPS.
P	This is a REAL*4 variable which transports a numeric value to/from the global array processor.
PD	This is a 257 element LOGICAL*1 array which transports a character string to/from the global array processor.
F	This is a LOGICAL*1 flag that indicates the success or failure of a global array directive.
DATRIB	This is a LOGICAL*1 variable that communicates the \$DATA code.

Internal Processor Communication Vectors

In addition to the main COMMON, which communicates with MUMPS, there are 10 named COMMON areas that are used for communication between the different global array processor routines. Table VI lists the names of the common areas and the vectors they contain.

TABLE VI
GLOBAL ARRAY PROCESSOR COMMUNICATION VECTORS

COMMON AREA	VECTOR NAMES
BTREE	HEAD NUMNOD INXNOD
FILES	U8 U9 CLRGBL G5NME G2NME TERM G5RECS G2RECS
IO	ERR
MAP	MAP MPTR
NODE1	PNODE PPNTR
NODE2	YPNTR
SEARCH	SKEY1 SKEY2
STACK	LEVEL LEVPTR
STRING	ENDSF FREESP
SUBKEY	SNODE

APPENDIX F

STRING FILE COMPRESSION ALGORITHM

The Compression Process

The algorithm for file compression utilizes two 512-element LOGICAL*1 arrays. Each array maps to a node in the global array string file. The first array is used to return compressed string file blocks back to disk. The other array is used to hold the fragmented string file block, extracted from the disk. There are two sets of pointers; one set for each array. The first pointer points to the current file block to which the particular array maps. The second pointer is an address relative to the top of the file that references a particular byte's position in the global array string file. The third pointer is an array offset relative to the first array element.

The process begins with all related pointers set to the first byte of the first block in the global array string file. Next the first file block is read into the fragmented array. Every element in the fragmented array that contains either a character string value or a character string delimiter, is transferred into the compressed array. As character string elements are transferred, the compressed file address pointer and the compressed file array offset pointer are incremented by 1. When the compressed array

becomes full it is written into the global array string file at the block pointed to by the compressed file block pointer. The array offset pointer is reset to the first element of the compressed array, and the compressed file block pointer is increased by 1.

Whether or not a character string value is transferred, as elements in the fragmented array are processed, the fragmented file address pointer and the fragmented array offset pointer are incremented by 1. When the fragmented offset pointer points past the last element of its associated array, the fragmented file block pointer is incremented by 1, the array offset pointer is reset to the first element of the fragmented array, and the block referenced by the new fragmented file block pointer is read into the fragmented array for processing.

This process continues until the fragmented file address pointer becomes greater than the current end-of-file mark address. At this point the end-of-file mark address is set to the compressed file address pointer. The rest of the global array string file is reinitialized from the new end-of-file mark to the bottom of the file. The global array string file header is updated to reflect the new end-of-file mark address. Control is then returned to the calling routine.

String Pointer Readjustment Process

As each new character string is transferred from the fragmented array to the compressed array, the corresponding file addresses are recorded into a 64 by 2 REAL*4 map matrix. MAP(I,1) holds the old fragmented file address pointer and MAP(I,2) holds the new compressed file address pointer. When all the entries in this matrix have been filled, or the last character string value has been passed between the fragmented array and the compressed array, a call is made to readjust the subkey character string pointers, inside the B-tree, to point to the new compressed string file addresses.

The readjustment routine sequentially reads each allocated B-tree node block. The subkeys in the selected node block are compared to see if they reference character string values. If a subkey is found to reference a character string value, the elements in MAP(I,1) are checked. If a match between an old string file address is found, the new compressed string file address contained in MAP(I,2) is transferred to that subkey. This process is repeated until all passed character string addresses have been readjusted in the B-tree global array node file.

VITA

Harry S. Cook, III was born in Camden, New Jersey, on November 2, 1956. His home town was Haddonfield, New Jersey until leaving for college in September of 1975. He attended Elizabeth Haddon Elementary School, Central Junior High School, and graduated from Haddonfield Memorial High School in June of 1975. The following year he began course study at the University of Tennessee, Knoxville in the college of Business Administration. He received a Bachelor of Science degree, with Honors, in Business Administration, majoring in Marketing, in June of 1979. Immediately after he finished his Bachelor's degree, he began study to attain a Master of Science degree in Computer Science, at the University of Tennessee, Knoxville. Finishing his degree requirements in December of 1980, he was granted his Master of Science degree in March of 1981.

Harry will be married in December of 1980 to Miss Elizabeth R. Whorton. They plan to live in the Washington D.C. area after their marriage.