# A B-TREE INDEX APPROACH TO STORING AND RETRIEVING RECORDS ON DIRECT ACCESS AUXILIARY STORAGE

By

DAVID DALE CHRISTIAN

Bachelor of University Studies

Oklahoma State University

Stillwater, Oklahoma

1977

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1978

# A B-TREE INDEX APPROACH TO STORING AND RETRIEVING RECORDS ON DIRECT ACCESS AUXILIARY STORAGE

Thesis Approved:

James R. VanDoren
Thesis Adviser

Donald D. Fisher

Donald W. Grace

Norman N. Durham
Dean of Graduate College

## PREFACE

A B-tree indexing scheme is used to access personnel records in a budget and personnel records maintenance system for the Dean's Office in the College of Arts and Sciences at Oklahoma State University. Primary and alternate indices are supported as well as the generic access of the keys in the indices.

This thesis and my graduate education have been aided over the past year and a half by several people and I would like to thank each of them.

I gratefully acknowledge the time and effort Dr. James R. Van Doren, my major adviser, put in to make my graduate education a profitable and memorable experience. Thanks are also extended to my committee members, Dr. Donald W. Grace and Dr. Donald D. Fisher, for their suggestions.

Further thanks are extended to Dr. Grace for the opportunity he gave me to further my knowledge and experience by performing a service for the Dean's Office in the College of Arts and Sciences at Oklahoma State University.

A very special note of appreciation is offered to my wife, Cherry, for the help, patience and love that has helped to make my graduate education the most enjoyable and exciting years of my life. I would also like to thank my

parents, Jerry and Verna Matheny, for their encouragement and assistance both financial and otherwise and for their moral support which ultimately made this thesis possible.

TABLE OF CONTENTS

TABLE

## LIST OF FIGURES

# CHAPTER I

## INTRODUCTION

As computers play a larger role in our society, more and more information is stored in computers. As files or collections of information increase, the efficiency of the techniques used for the storage and retrieval of that information increases in importance. Usually, the information or data in a file is not kept in the main memory of the computer but is stored on auxiliary or secondary storage such as disks or drums. If a file on disk or drum has changes made to it frequently, the file is referred to as a volatile file. Many organizational methods are very inefficient for volatile files and lead to an increase in the time required to access records in the file. A personnel file could be an example of a volatile file since there may be frequent changes made to the file.

For example, a file that is physically stored in a sequential fashion requires the entire file to be rewritten for any deletion or insertion to any part of the file except to the end of the file. The indexed sequential method requires the frequent reorganization of the entire file for volatile files. Unless reorganized, the access of a record

in an indexed sequential file could involve the search of a large overflow area and thus involve a lot of time.

A new approach for the storage and retrieval of records on direct access auxiliary storage was discovered in 1972 by R. Bayer and E. McCreight (2). This approach, using a data structure called a B-tree, requires no complete file reorganizations, makes efficient use of auxiliary storage and has a guaranteed time and space efficiency even in the worst case.

The main topic of this thesis is to discuss the design, implementation and uses for a variation of the general B-tree referred to as a B-tree index. The framework for that discussion follows.

Chapter II will introduce B-trees, B*-trees and the B-tree index. Following that will be a discussion of the uses for a B-tree index in a data base. Chapter II will close with a discussion of the advantages, disadvantages and alternatives to a B-tree index.

The design of a system by the author for the Dean's Office in the College of Arts and Sciences called for an access method based upon B-trees for the storage and retrieval of personnel records. The design and implementation of that B-tree index program is presented in Chapter III. This includes the data structure design, the logic design and implementation factors.

A frequent requirement for application programs is the need to access all the records in a given file that have a particular attribute in common. For example, in the system mentioned above, a program might be required to process all the personnel records in a given department that have a given rank. In this example, the department and the rank are the attributes. These attributes are used as the key field for a B-tree. By using different permutations of these attributes for the key fields of other B-trees, a set of secondary indices could be constructed. Chapter IV discusses a routine that matches a partial key (taken from one of the attributes) to the leading portion of one of the B-tree indices (the one with that attribute listed first) and calls a routine to process each record it finds that matches the partial key. This is commonly referred to as a generic access capability.

The final chapter is a discussion of the utilization of B-tree indices in a system of programs designed and implemented by the author for the Dean's Office in the College of Arts and Sciences at Oklahome State University in Stillwater, Oklahoma. This discussion will be limited to that information necessary to explain the role of the B-tree indices in that system.

Appendices will include Program Design Language (PDL) descriptions of the major programs discussed in this thesis as well as actual program listings. Also included are a few programs showing a sample usage of the above programs.

# CHAPTER II

## B-TREES, B*-TREES AND B-TREE INDICES

### Description and Use of B-trees

A B-tree is a uniform depth search tree with guaranteed efficiency even in the worst case. A B-tree grows from the bottom rather than from the top like a binary tree. The following rules apply to a B-tree of order[1] m (8):

1. Every node has no more than m offspring.
2. Every node except the root node has at least $\lceil m/2 \rceil$ offspring.
3. If the tree is not empty, the root node has at least two offspring.
4. All leaves appear on the same level and carry no information.
5. A nonleaf node with k offspring contains k-1 keys.

The symbol "$\lceil e \rceil$" means "the smallest integer larger than e". Since the leaf nodes (external nodes) carry no information, the pointers to them are null pointers and the leaf nodes are not actually stored in the tree at all.

---

[1]The order of a B-tree is usually the maximum number of branches from a node.

In Figure 1, every node except the root is required to have at least $\lceil m/2 \rceil$ offspring. This means each node has at least three offspring and, therefore, at least two keys.

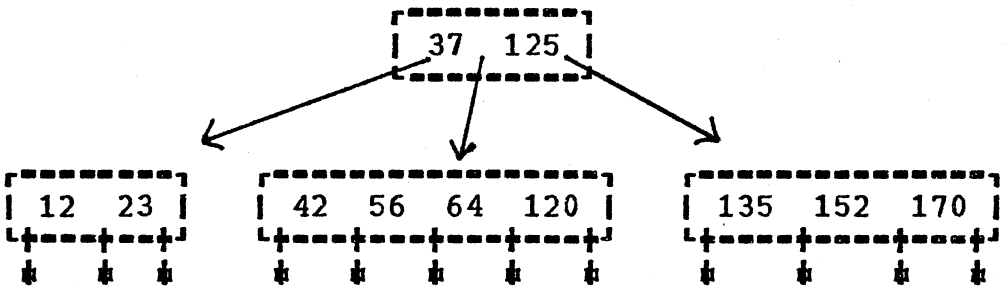



Figure 1. A B-tree of Order 5

A node with j keys and j+1 pointers may be represented as

(j,P(1),K(1),P(2),K(2), . . . , P(j),K(j),P(j+1))

where j is the number of keys in the node. The keys and pointers are situated such that K(1) < K(2) < ... < K(j) and P(i), for 1 < i < j+1, points to a subtree for keys between K(i-1) and K(i). P(1) points to a subtree with keys less than K(1) and P(j+1) points to a subtree with keys greater than K(j). In the root node of Figure 1, K(1) is equal to "37" and P(1) is represented by the left-most arrow descending from the node. This notation is similar to that used by Knuth (8). The difference is that the number of

keys in the node (j) is not included in Knuth's version and Knuth numbers his subscripts for P beginning at zero instead of one.

As can be seen in Figure 2, a B-tree of order 5 with two levels and maximum branching has 25 external nodes and 24 keys. Figure 2 also shows that for minimum branching and three levels in an order 5 B-tree, there are 18 external nodes and 17 keys.

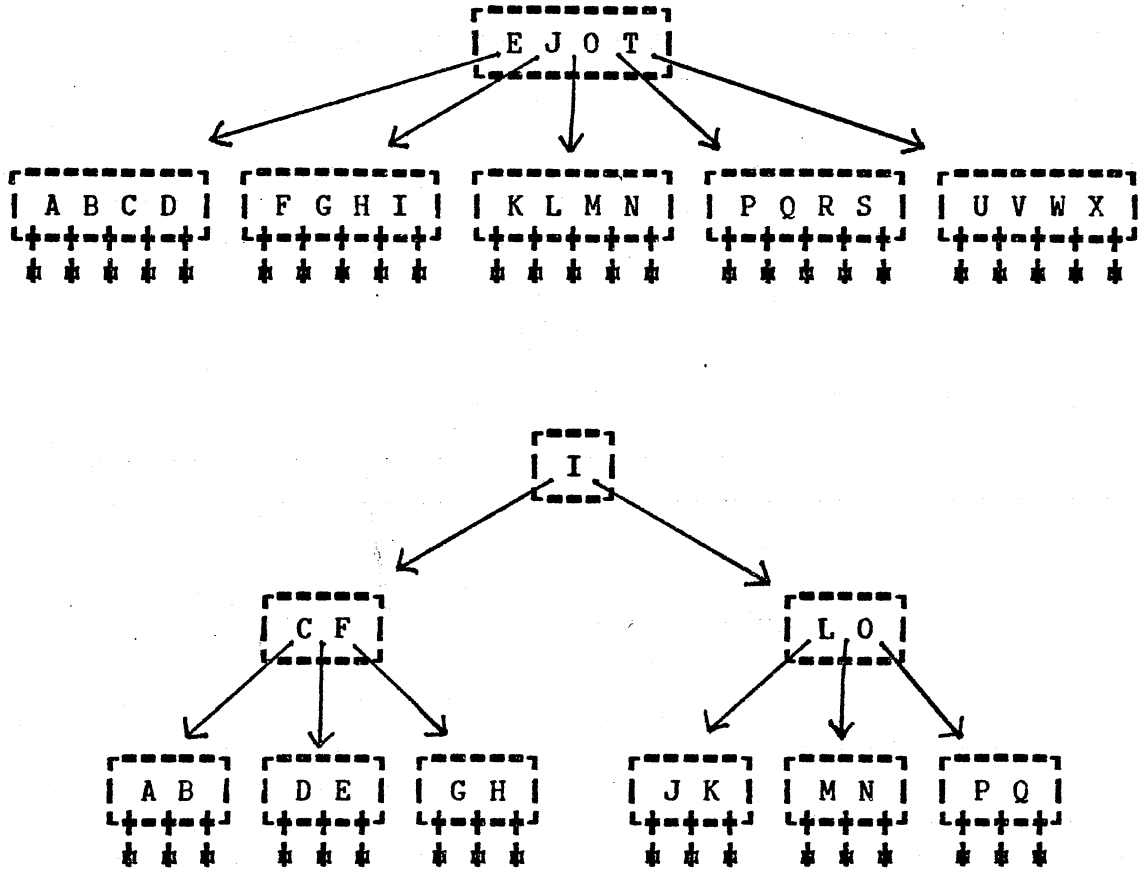



Figure 2. Maximum and Minimum Branching in a B-tree Node of Order 5

## Searching

The basic operations performed on search trees are searching, insertion and deletion. The ordering within nodes makes searching a straightforward procedure:

1. Bring the root node into internal memory.
2. Find the first key "K(i)" in the node greater than or equal to the search key. If the search key is greater than K(j), i=j+1.
3. If the search key is equal to K(i), the search is successful and the search terminates. The search also terminates if the search key is not equal to K(i) and P(i) is null.
4. At this point, the search key is either less than K(1), between K(i-1) and K(i) or greater than K(j). The node indicated by P(i) is brought into internal memory and control returns to step 2.

The search performed in step 2 is a range search and requires that the physical organization of the keys in the node accomodates this type of search. A range search is used here to mean a search for two adjacent keys that "bracket" the search key. Therefore, the keys may be organized as a binary tree, as an ordered sequential list or in any fashion that allows a range search to be performed. Usually, the keys are physically stored in ascending order within a node. A binary search is the recommended method for finding the proper key in step 2 unless the maximum number of keys in a node is small. If there were only about six or eight keys in a node it would probably be as fast (and simpler) to perform a linear search.

## Insertion

The insertion of a key into the tree requires that the key be inserted into a bottom level node such that j is changed to j+1 and K(i-1) < K(i) < K(i+1) where K(i) is the new key and j is the number of keys in the node. If j is less than m (the branching factor) the insertion is finished. Otherwise, j is equal to m and the node is overfull. At that point, the middle key of the overfull node is inserted or promoted into the parent node of the overfull node and the overfull node is split in two. Figure 3 shows the B-tree of Figure 1 after the insertion of the new key "45".

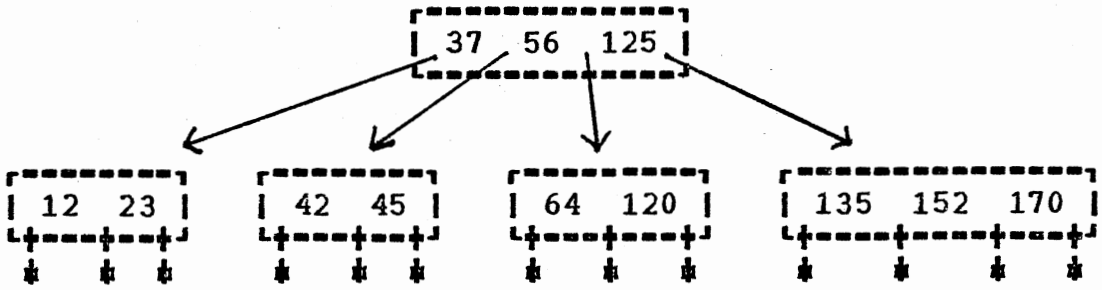



Figure 3. B-tree of Figure 1 After Insertion of the Key "45"

In general, if a node becomes overfull, the node is split putting a middle key into the parent node, lower keys

in one of its successors and higher keys in the other, as follows:

$$(\lceil m/2\rceil-1, P(1), K(1), \ldots, P(\lceil m/2\rceil-1), K(\lceil m/2\rceil-1), P(\lceil m/2\rceil))$$

would be left in the overfull node and

$$(m-\lceil m/2\rceil, P(\lceil m/2\rceil+1), K(\lceil m/2\rceil+1), \ldots, P(m), K(m), P(m+1))$$

would be put into the other node. The key $K(\lceil m/2\rceil)$, the "middle" key, is now inserted into the parent node. For example, if m=5 or m=6, K(3) would be inserted into the parent node. If this process causes the parent node to become overfull, the process of splitting the node and promoting the middle key is followed again. This could continue until the root node itself is split, in which case a new root node is formed with the single key promoted by the split.

## Deletion

Deletion is more complicated. The basic idea is to take whatever reshuffling steps are necessary to maintain a balanced B-tree after deletion, much the same as you would have to do if you removed one element (node) of a "mobile".

Consider a node (called current node) from which the key K(i) is to be deleted. The cases to consider depend on whether the current node is a lowest level node or an upper level node. If the current node is a lowest level node and $j \geq \lceil m/2\rceil$, K(i) and P(i) may be deleted and the deletion proc-

ess is finished. Figure 4 shows the result of deleting the key "64" in the B-tree of Figure 1. On the other hand, if $j=\lceil m/2 \rceil - 1$, the deletion of K(i) would cause the node to become underfull (thereby violating one of the requirements for a B-tree). In that case either a rotation is performed using one of the current node's sibling nodes or, the current node is combined with one of its sibling nodes[2].

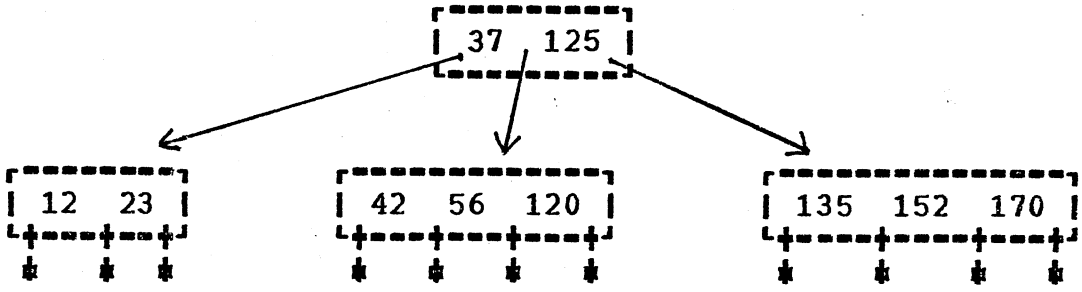



Figure 4. Result of Deleting the Key "64" in the B-tree of Figure 1

A rotation is performed if the current node's right or left sibling has at least $\lceil m/2 \rceil$ keys. This is performed with the right sibling by moving the key K(p) in the parent node into the current node where p is defined such that P(p) is the pointer in the parent node to the current node.

---

[2]The keys within the nodes are rotated, not the nodes themselves.

Then, the keys in the current node and its right sibling are shared equally and K(p) in the parent node is replaced by the rightmost key in the current node. For large order B-trees, more than one key may be moved out of the sibling node. Figure 5 shows the result of deleting the key "12" in the B-tree of Figure 1.

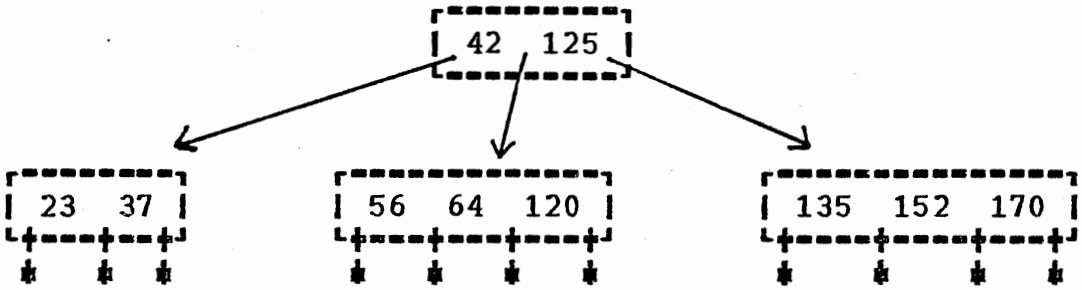



Figure 5. Result of Deleting the Key "12" in the B-tree of Figure 1

A similar procedure is followed if the rotation is to be performed using the current node's left sibling. In this case the key K(p-1) in the parent node is moved into the current node where p is as defined above. Then, the keys are shared and the parent key is replaced as before.

If a rotation cannot be performed because both the left and right siblings contain $\lceil m/2 \rceil - 1$ keys, the current node is combined with its right sibling after the key K(p) in the parent node is moved to the current node. Consider the

result in Figure 6 of deleting "12" in the B-tree of Figure 3. If the current node does not have a right sibling, the current node can be combined with its left sibling after the key K(p-1) in the parent node is moved into the left sibling.

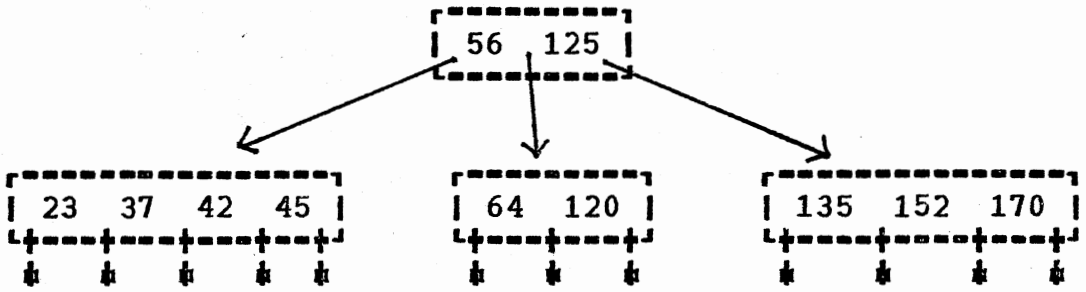



Figure 6. Result of Deleting the Key "12" in the B-tree of Figure 3

Since the above process removes a key from the parent node and does not replace it, it is possible that the parent node can become underfull. If that should occur, the above process is repeated with the parent node becoming the new current node. The root node does not require any reorganization when keys are deleted unless it becomes empty. If the root node should become empty because the two nodes on the next level were combined, it is discarded and the combined nodes become the new root. Figure 7 shows the deletion of the key "N" in the second B-tree of Figure 2.

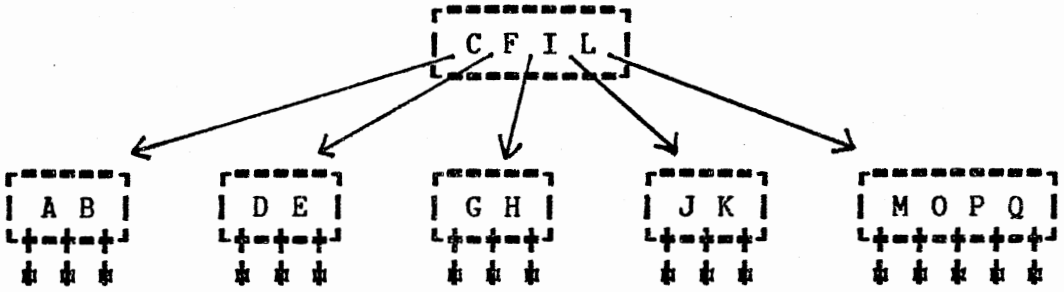
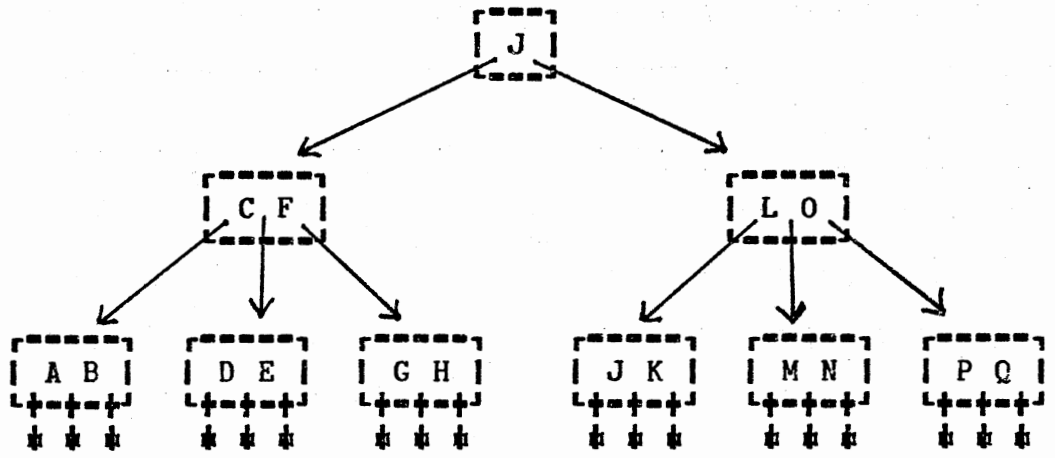



Figure 7. Result of Deleting the Key "N" from the Second B-tree of Figure 2

The deletion discussion above was for the deletion of a key in a lowest level node. If the key to be deleted is in an upper level node, the key K(i) is replaced by the smallest key in its right subtree (P(i+1)) or the largest key in its left subtree (P(i)). Then, the key that was copied to the upper level node is deleted from the lowest level node following the process for deleting a key in a lowest level node. Figure 8 shows the replacement of the key "I" after being deleted from the second B-tree of Figure 2. The deletion is not completed until "J" is deleted from the lowest level which is shown in the second B-tree of Figure 8.

Performance

The B-tree described in the previous paragraphs has a guaranteed space utilization and performance efficiency. The following is a discussion of the upper and lower bounds on that efficiency. For a B-tree of order m stored on disk




The intermediate tree

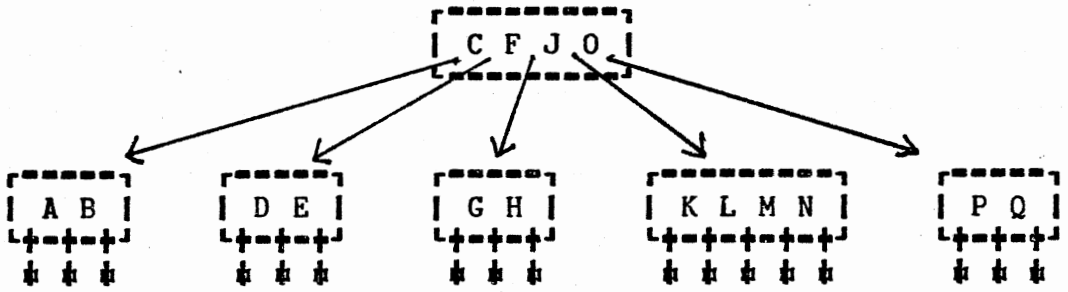



Figure 8. Deletion of the Key "I" in the Second B-tree of Figure 2

or drum, the number of levels in the tree determines the maximum number of accesses to find a given key. In the case that all nodes are filled to the minimum amount, each node has $\lceil m/2 \rceil$ offspring except the root which has only two offspring.

Consider level zero in the B-tree to be the one with the physically non-existent external nodes (leaf nodes).

Level one is the lowest level in the B-tree and level p is the one that contains the root node. The numbering method used here is not the conventional numbering scheme. The conventional method puts the root node at level one and numbers down from there. The method taught in Data and Storage Structures by Dr. James R. Van Doren (11) numbers the levels from the bottom up instead. The reason for this is that once a node is created at any given level, the node stays at that level and its level number never changes. In the more conventional numbering scheme, every time the root node splits, every level is renumbered. One of the advantages of this scheme is that when different branching factors are used at different levels (discussed later in this section), one can always know the branching factor on a given level by its level number (and that level number never changes).

If there are n keys in the B-tree, there are n+1 external nodes. Table I displays the maximum and minimum branching at each level in a B-tree of order m with p levels. The symbol "**" is used to represent exponentiation.

If there are n keys in a B-tree of order m (m>2), there are n+1 external nodes (leaf nodes) and

$$2*\lceil m/2 \rceil **(p-1) \leq n+1 \leq m**p.$$

By solving the above equation for p we get

$$\log_m (n+1) \leq p \leq 1+\log_{\lceil m/2 \rceil} ((n+1)/2).$$

TABLE I

BRANCHING IN A B-TREE OF ORDER M

| Minimum Branching | | Maximum Branching | |
|---|---|---|---|
| level | branches | level | branches |
| p | 2 | p | m |
| p-1 | 2*⌈m/2⌉ | p-1 | m**2 |
| p-2 | 2*⌈m/2⌉**2 | p-2 | m**3 |
| • | • | • | • |
| • | • | • | • |
| • | • | • | • |
| 2 | 2*⌈m/2⌉**(p-2) | 2 | m**(p-1) |
| 1 | 2*⌈m/2⌉**(p-1) | 1 | m**p |

This shows that the number of levels in a B-tree (which also indicates the maximum number of nodes to be searched) is logarithmic in nature. The base of the logarithm is dependent on the branching factor or the order of the B-tree. For example, if 199,999,999 records were to be stored on disk in a B-tree of order 20, $6.38 \leq p \leq 9$. So there are at least seven levels in the B-tree but no more than nine levels. Therefore, any record of the almost 200 million records may be retrieved in nine or less disk accesses.

Uses for B-trees

The primary uses for a B-tree involve the use of auxiliary storage since a B-tree with a sufficiently large branching factor can considerably reduce disk accesses to find needed records. It must be remembered though, that a

B-tree of order m must have room in each node for m-1 keys, m pointers and m-1 records. This is important because there must be sufficient room in internal memory for one or more B-tree nodes and, if the records are not small, there is the potential for a lot of wasted space in nodes that are only about half full. B-trees may also be useful as an internal search tree for programs that execute in a virtual memory system. In such a system, "pages" or sections of a program that are accessed frequently are kept in main memory and the other pages are "swapped out" onto disk until referenced.

If the internal search tree were organized as a binary tree, any branch would reference any page within the tree and all this "hopping" about would probably cause many more "non-resident" or "swapped out" pages to be referenced. If the search tree were organized as a B-tree instead, as much work as possible would be performed in each node, thus reducing the "hopping" about. Also, the root node and perhaps the nodes on the next level would remain in internal memory since these nodes would be referenced frequently.

Summary

This section introduced the B-tree and discussed the search, insertion and deletion processes in such a data structure. Following that was a discussion of the performance of B-trees where it was shown that the search time in a B-tree increases only logarithmically with an increase in the number of keys. Although primarily used for external

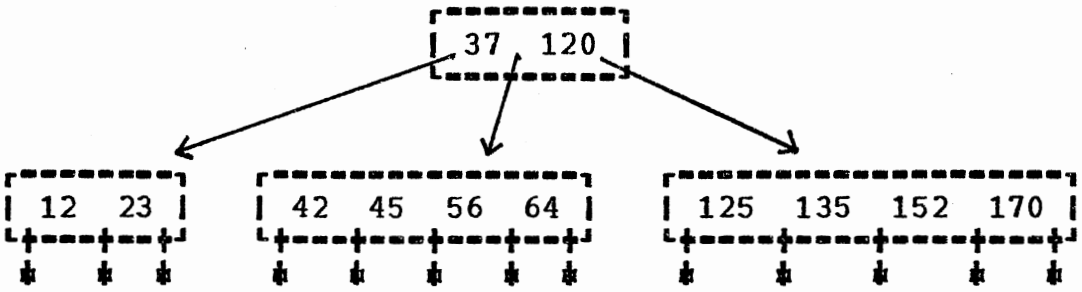
searching on direct access auxiliary storage, B-trees may also be useful for internal searching in cases where paging is a problem.

## The B*-tree Variation

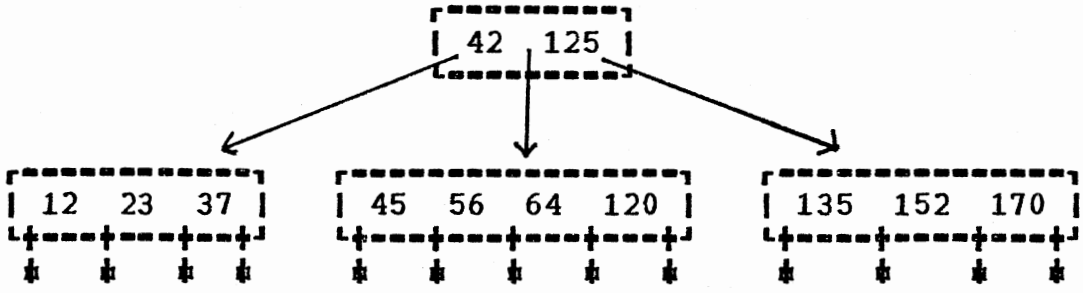
The B-tree of the previous section can be improved upon by not wasting the storage required for the null pointers at level one. Instead, m can be increased for level one by storing only keys in those nodes (8). This does not cause problems because a node created at level one is always a level one node while it exists. If different branching factors were desired for different levels in the tree, a table of these branching factors could be built for use by the B-tree algorithms.

### Overflow Sharing

A significant improvement in storage utilization within a node can be realized by resisting the temptation to split nodes each time they become overfull. The idea is to share overflow with a sibling node. If a node becomes overfull, the proper key in the parent node is put into a sibling node and the keys and pointers in the two nodes are divided up so that they have about the same number of keys and pointers. Then, a key is put back into the parent node to reflect the contents of the two nodes that have been rearranged. Figure 9 shows the result for Figure 3 if overflow sharing is used.




Overflow to the Right




Overflow to the Left

Figure 9. Overflow Sharing During Insertion of the Key "45" into the B-tree of Figure 1

When overflow is not possible because one or both siblings are already full, then a split is necessary. Knuth suggests that the current node and its sibling node could be split into three nodes about 2/3 full (8). This guarantees that utilization of the space in nodes would never be less than 2/3 except perhaps in the root node.

This leads to the definition of B*-trees:

1. Every node has no more than m offspring.
2. Every node except the root node has at least $\lceil (2m-1)/3 \rceil$ offspring.

3. The root node has at least two but not more than $2\lfloor(2m-2)/3\rfloor+1$ offspring.

4. All leaves are on the same level and carry no information.

5. A nonleaf node with k offspring contains k-1 keys.

The symbol "$\lfloor e \rfloor$" means "the largest integer smaller that e". Rule 3 is necessary to insure that a split of the root node produces two nodes that still meet the requirement of rule 2. Rule 2 is the rule that forces a higher minimum usage value for each node of about 2/3.

The report on B-trees written by William Davis (5) shows empirically that the increased utilization gained by doing two-way splits versus three-way splits is not significant for trees built in random key order. A large advantage is gained by sharing overflow, but the addition of three-way splits is probably not worth the increased complexity it adds to the insertion algorithm. Tables VIII, IX and X in Davis' report (5) show that performing three-way splits only rarely produces better results than the simpler two-way splits.

## The B-tree Index Variation

The B-tree is a very versatile data structure and can be used for a variety of purposes. There are three basic variations in the method of storing the data that correspond to keys in the B-tree. One method actually stores the

record with the key and its pointer in the node where the key appears. This method is fine if the records are very small. If records are not small, a significant amount of space can be wasted in nodes that are not nearly full.

Another method of storing the data that corresponds with the keys in a B-tree is to put a pointer to the data record with the key and the B-tree pointer. This method allows the actual data to be stored in a separate location than the B-tree. An "available list" strategy could be used for the data record file. The key and a pointer to the data record would be inserted into the B-tree when a new record is put in the data file.

The third basic variation has the data records stored in level one of the B-tree with the keys at that level. The upper levels in the B-tree do not have data records stored in them. When the insertion of a record causes the split of a level one node, only the key is promoted to the next level. This means that all data records remain in the bottom level of the B-tree and that the keys in the upper levels of the B-tree duplicate the rightmost (or largest or highest) key in each record block at level one except the rightmost one. The Virtual Storage Access Method (VSAM) used by IBM (7) is based upon this variation. The primary difference in VSAM is that all the keys in a given level are duplicates of the rightmost keys of the nodes at the next lower level. Indexed sequential is another storage method based upon using an index. Two of the problems with indexed

sequential are that this method is not good for a volatile file and that the indexing is tied to the physical device (cylinder index, track index, etc.).

The variation used by the author is a hybrid of both the second and the third methods listed above. At level one in the B-tree, all pointers point to the actual records which may or may not be in the same file as the B-tree. In the upper levels, all pointers point to other B-tree nodes. All keys and all pointers to the data records appear in level one. An insertion causes a new key and pointer to appear at level one and, if a split occurs, a key is copied to level two and also remains at level one. In other words, the upper levels of the B-tree contain keys that duplicate the rightmost key in each B-tree node at level one except the rightmost node.

The following rules apply to a B-tree index:

1. Every node has no more than m offspring.
2. Every node except the root node has at least $\lceil m/2 \rceil$ offspring.
3. The root node contains at least one key.
4. All leaves (external nodes) appear on the same level and carry no information.
5. Except for the bottom level nodes, all nodes with k offspring contain k-1 keys.
6. All nodes at the bottom level with k offspring contain k keys.
7. All insertions are made at the bottom level of the B-tree and no keys are moved to the upper levels. A split at the bottom level causes a key to be copied to the next level.

Searching is basically the same for the B-tree index except that the search is never completed until the key is found at level one in the B-tree. The record itself may be retrieved by using the pointer with the key at level one. The search procedure discussed in the first section of this chapter must be changed as follows:

1. Bring the root node into internal memory.
2. Find the first key "K(i)" in the node greater than or equal to the search key. If the search key is greater than K(j), i=j+1.
3. If the current node is a bottom level node, the search is successful if the search key is equal to K(i) and the search is unsuccessful if the search key is not equal to K(i).
4. Otherwise the current node is an upper level node and the node indicated by P(i) is brought into internal memory and control returns to step 2.

Insertion requires all keys to be inserted at level one. If a split occurs, the rightmost key in the node containing the first half of the keys is copied into the next level. Any splits that occur in the upper levels are handled with the same method as that used for a standard B-tree.

Deletion requires that the key (and its pointer) be deleted at level one. If the key is the rightmost key in that node, its occurrence in the upper levels must be changed to reflect the new rightmost key in the node at level one.

A B-tree index of the type just described requires p+1 accesses to retrieve any one record where p is the number of levels in the B-tree. This can be reduced by keeping the root node in internal memory.

The B-tree index described above was designed and implemented by the author. The following chapters in this thesis discuss the design, implementation and uses for this type of a B-tree. Chapter V contains a discussion of a system that used this B-tree index.

## Relational Data Base Uses for a B-tree Index

In a relational data base there are collections of data that are referenced and manipulated by the use of defined relations. A relation, in simplest terms, is the logical structure of a set of related information that may or may not be closely related to the physical representation of the information. The use of a relation is intended to be independent of the way the information is stored or accessed.

An image of a relation is simply a copy of the relation ordered on one or more of the attributes of the relation. Rather than physically copy the relation, an index is created with pointers to the actual data.

In "System R" (1) the developers use a B-tree index to store the pointers to the actual data called Tuple Identifiers (TID). A TID contains the address of the tuple (a piece

of data) that is desired. Therefore, tuples may be accessed directly by traversing one of the B-tree indices defined for a relation. This type of use for B-trees can result in much faster on-line responses to various queries than other organizational methods.

B-trees are a very versatile data structure. Haerder (6) describes a generalized access path structure where images that use the same attribute for the key field are combined into one B-tree index. This also allows the implicit use of binary links[3]. This means binary links can be used without physically storing the links anywhere. The links are an implicit result of the structure and ordering of TID's in a B-tree node. An alternative to this approach is to store the links in the tuples themselves which could cause the following:

1. The system could be too slow.
2. It would increase maintenance problems when tuples are inserted or deleted.
3. It would increase the complexity of access to tuples since there would be at least two separate methods of access (by index and by binary link).

---

[3]A binary link is a direct path from the tuple in one relation to its offspring tuple(s) in another relation (or back to its parent tuple if it is an offspring).

## Advantages, Disadvantages and Alternatives to B-trees

B-trees have several advantages over other contemporary organizational methods. One advantage is that complete file reorganization is never required and all reorganization is performed within a node or with its parent or sibling nodes (called local reorganization). Another advantage is that the lower bound performance and expected performance can be closely approximated and that performance can be quite good (11). In some organizational methods, performance is dependent on the number of overflow records in the file or extensions added onto the file.

One of the disadvantages of B-trees is the amount of work that must be done in a node in internal memory for searching, insertion and deletion. This work is ordinarily insignificant though, when compared to the time required for the physical input and output of records to disk. The B-tree algorithm attempts to reduce access time significantly at the expense of additional work to be done in internal memory.

Another disadvantage with B-trees is the utilization of space if all nodes are only about half full. This problem can be avoided if overflow sharing techniques are used to reduce the number of splits that occur.

An alternative to using an organizational method based upon B-trees (like IBM's Virtual Storage Access Method) is

to use some other organizational method such as indexed sequential or inverted files (4). If an indexed sequential method were used as an alternative to a B-tree based method, a volatile file would need frequent reorganization.[4]

The B-tree index approach described earlier would require an access method that allows direct access of records. The logical access of records in the file could be sequential or direct or sequential from a point arrived at directly. If a group of records were to be retrieved that had a particular attribute in common, the capability to traverse a B-tree sequentially from a point arrived at directly could save a lot of time by traversing only that part of a B-tree that matches the common attribute. Sequential access would be accomplished by performing an in-order traversal of the B-tree. Sequential access from a point arrived at directly to retrieve a subset of records (commonly called "generic access") is discussed in Chapter IV. This capability is offered only by a B-tree approach (such as the author's program or VSAM) and an indexed sequential approach to organizing and accessing records on direct access auxiliary storage.

---

[4]Another alternative is to order the records in a sequential file and use no index. The entire file would be reorganized if records are inserted or deleted. Also, any search for records based upon an attribute that was not used for the ordering requirements would require a scan of the entire file.

Based upon the advantages and disadvantages mentioned, it is the conclusion of the author that organizational methods for storing records on direct access auxiliary storage based upon B-trees would guarantee quick access to a given record, would make reasonably efficient use of that storage[5] and would remove the requirement for rewriting or reorganizing volatile files. An organizational method based upon B-trees would allow sequential access or direct access based upon a key. Also, multiple B-tree indices could be created based upon different permutations of the key fields in an existing B-tree leading to secondary indices that could be used to access the records rather than using the primary index.

---

[5]More efficient use of the storage could be realized by using a strictly sequential file with no index. But, given the properties desired, B-trees have good performance characteristics and still make efficient use of storage (11).

# CHAPTER III

# DESIGN AND IMPLEMENTATION OF A B-TREE INDEX PROGRAM

The data structure design, the design of the logic and the implementation factors of a B-tree index program written by the author are presented in this chapter. Program Design Language (PDL) descriptions and program listings are available in the appendices.

## Data Structure Design

The data structure design described in the previous chapter in the section on the B-tree index variation is the one used by the author. The basic difference between this type of B-tree and the standard B-tree is that pointers at the bottom level of the B-tree are not null or wasted. At the bottom level all pointers actually point to the records that contain the data. Another difference is that all keys in the tree appear at the bottom level. All the keys in the upper levels duplicate the rightmost key in each bottom level node except the rightmost node.

The purpose for using a B-tree this way is to allow the insertion and deletion of records in the record file and the

insertion and deletion of keys in the B-tree index to be independent of each other. If a "simple singly-linked list of available storage" technique is used for creating and deleting records in the record file, the records never need to be reorganized because of the volatility of the file. The only reorganization ever required is the local reorganization of keys in a B-tree index node (and possibly one of its sibling nodes). The index file itself never requires "wholesale" reorganization. The B-tree is always well organized and the pointers at level one locate the actual records requested.

This method or variation of a B-tree leads to interesting differences in the search, insertion and deletion algorithms used for standard B-trees. The next section contains a discussion of the design of the logic required to implement this data structure design.

## Logic Design

The design of the logic for the B-tree algorithms requires the prior determination of the methods to be used in insertion. Overflow can be handled by simply splitting the node or by attempting to share the overflow with a sibling node. It was decided to attempt to share overflow during insertion to first the right sibling and then to the left sibling if the right sibling were full. All splits are simple two-way splits. The basis for this decision came

from the results of a study on B-trees made by Davis (5). The basic design of the program is shown in Figure 10.

```
BTINDX:  PROC;
     set OPCODE to the operation to be performed;
     SELECT(OPCODE);
          WHEN(search) find occurrence of key at level
            one and return the corresponding pointer;
          WHEN(insert) insert the key and the record
            number at level one in the B-tree;
          WHEN(delete) remove the key and its pointer
            at level one in the B-tree;
            Change any occurrence of the key in the
            upper levels of the B-tree;
          OTHERWISE signal an error condition;
     END;
END BTINDX;
```

Figure 10. A High-Level PDL Description of a B-tree Index Procedure

The entire program was designed by writing a PDL (Program Design Language), a pseudo language with structured programming constructs. Its use replaces the flowchart as a design tool. After the transformation of the PDL into PL/I, there were no major design changes required. In the author's opinion this is attributable to the superiority of a PDL over the use of a flowchart method of design. This opinion is supported by a study performed by Dr. Van Doren and others (10).

The PDL for BTINDX, the B-tree index program, is written so that it may now be implemented in any high-level programming language. BTINDX is implemented in PL/I.

```
IF LEFTSIB exists and is not full THEN
   IF CURNODE is not a level one node THEN
      copy LEFTSIB's parent key into LEFTSIB; FI;
   perform overflow sharing to the left;
   set LEFTSIB's parent key = highest key in LEFTSIB;
   RETURN; FI;
IF RIGHTSIB exists and is not full THEN
   IF CURNODE is not a level one node THEN
      copy RIGHTSIB's parent key into RIGHTSIB; FI;
   perform overflow sharing to the right;
   set CURNODE's parent key = highest key in CURNODE;
   RETURN; FI;
CALL SPLIT;
```

Figure 11. Insertion Overflow Procedure in PDL Form

Refer to Figure 11 for a PDL description of the following discussion. When a key is inserted into a lowest level node, overflow is handled a little differently (and will be explained later) than if overflow is in an upper level node. If overflow at the lowest level is handled by sharing, the parent key of the leftmost share partner[1] must be changed to the new rightmost key in that node.

---

[1]Share partners are the current node and either its left or right sibling. The two nodes could be involved in sharing overflow or the two nodes could be the result of a split.

A split at the lowest level requires the rightmost key of the left share partner be inserted into the parent node right before the current node's parent key. The key remains at level one and is inserted into its parent node.

If overflow occurs at other than level one and sharing is possible, the parent key that would come "between" the two nodes must be copied into one of the nodes before sharing is attempted. After the keys are divided, the key in the parent node must be changed to reflect the rightmost key of the leftmost share partner.

If a split is necessary in an upper level node, the rightmost key in the leftmost node of the two created by the split is inserted into the parent node. (A split in a level one node would have left the key in the level one node as well as inserting it into the parent node.)

Deletion always starts at level one. If neither of the following is true, the deletion process is finished.

1. The current node is now underfull.
2. The key deleted was the rightmost key in the node.

If the current node is underfull, its keys are either combined or shared with one of its siblings. As with insertion, any sharing of keys requires the proper key in the parent node be copied into one of the share partners if the current node is not a level one node. Upon completion of the sharing, the proper key in the parent node must be

updated to reflect the new rightmost key in the leftmost node of the two share partners.

When two nodes are combined, the key and pointer to the leftmost node of the two nodes combined must be deleted as well. It is possible that the parent node will become underfull and therefore the process could continue.

When an underfull condition does not exist, but the rightmost key in the most recent node involved was the one deleted, the tree must be traced back towards the root changing the occurrence of the deleted key in the upper part of the tree to the new rightmost key in the node that had its rightmost key deleted. The only time this is not necessary is when the node that had its rightmost key deleted is the rightmost node at its level.

## Implementation Factors

A structure for a single node in a B-tree could look like:

```
DECLARE
1 NODE,
  2 PTRS(0:MAXB+1) FIXED BIN(15),
  2 KEYS(MAXB) CHAR(KLEN);
```

MAXB is the branching factor and KLEN is the length of the key field. For ease of implementation, all nodes are required to have an extra pointer and an extra key so that an overfull node will physically fit in a node and can be handled from that point.

The BTINDX routine was written so that it could manipulate more than one B-tree. The B-trees could be located in different files and have a different branching factor and key size. This generality has a tradeoff in the declaration of the space used by a node. One structure contains all the physical space for up to three nodes. CURNODE, SIBNODE and PARNODE are based upon this physical storage. The key length and dimensions for the number of keys and pointers are variable. PL/I requires a "REFER" option to be applied to based structures with variable-dimensioned substructures. Crotzer (4) did not use this method because it was not available with the PL/I F compiler and therefore he could not use based structures with variable-dimensioned arrays.

Three nodes are contained within memory at any one time. These nodes are referred to as CURNODE, PARNODE and SIBNODE and represent the current, parent and sibling nodes. Although a node may be the current node at one point, it may become the parent node during a traversal. Rather than physically move the values in CURNODE to PARNODE, base variables are switched so that CURNODE just becomes PARNODE. This strategy is used throughout to reduce the amount of work that must be done while maintaining the B-tree.

Refer to Appendices A and E which contain the BTINDX design and program, respectively. The program uses positive pointers in the upper levels to point to other B-tree nodes

but uses negative pointers at level one to point to the actual records. This is transparent to the user of the routine since POS (the location of the record in the record file) contains the absolute value of the negative pointer at level one.

The search routine continues moving down levels in the B-tree, stacking node numbers and the location of the pointer to its offspring node until it arrives at level one where success or failure of the search can be determined. All insertions and deletions are preceded by a search to determine if the key exists and to locate the target for an insertion or deletion if the proper conditions exist.

The stack is used to locate the ancestor nodes if it is necessary to trace back through the tree toward the root. After the physical insertion or deletion of the key and its pointer at level one, a determination is made as to whether the process is finished or more work is to be done.

For insertions, the process is complete after inserting the key and its pointer in a level one node if an overflow condition was not created. If overflow has occurred, an attempt to share the overflow with a sibling node is made. A sibling node is easy to find since the current node's parent node, if it has one, is already in main memory. If a split occurs at level one, the rightmost key in the leftmost share partner is copied to the parent node. Therefore, the number of keys in the parent node increases and it may

become overfull. Splits in upper level nodes are handled in a standard B-tree fashion using overflow sharing if possible. In other words, splits in the upper levels cause a key to get promoted (not copied) to the next level.

If overflow is handled by sharing, the rightmost key in the leftmost share partner will change. This means that a key at the next level must also be changed. This process never requires looking farther than the parent node since the rightmost key in the rightmost offspring node will never change due to overflow sharing.

As mentioned earlier in this chapter, deletion requires that a search be performed to find the key at level one. If the key exists, the key and its pointer are deleted from the node. If an underfull condition exists (there are less than $\lceil maxb/2 \rceil - 1$ keys) or the rightmost key is the one deleted, more work is required.

If a level one node is underfull, the keys in it and one of its sibling nodes are shared as long as one of the siblings is more than minimally full. When the deletion process does not cause an underfull condition but, the rightmost key is the one deleted, the tree must be traced back towards the root to change the occurrence of the deleted key.

If the underfull level one node has siblings that are both minimally full, it is combined with a sibling node and the key that used to be the rightmost key in the leftmost

node of the two combined nodes must be deleted in the upper part of the tree.

Any underfull condition in a node that is not a level one node is handled by the procedure for underfull nodes in a standard B-tree as described in Chapter II. Underfull conditions are handled by rotating if possible and combining nodes otherwise.

# CHAPTER IV

## GENERIC ACCESS OF A B-TREE

A useful capability when using search trees is to access records in the tree whose key matches a partial search key. This is called "generic access" (9). For example, if a person's age and name were recorded in that order as the keys for a search tree, the partial key could specify an age and all records with that age would be accessed. This chapter discusses a generic access routine written for a B-tree index maintained by the program in the previous chapter.

The requirements of the routine are:

1. It should be general enough to permit access to more than one B-tree within the same program.
2. The routine called to process a record when the key condition is satisfied should be variable.
3. It should access records in collating sequence order and should perform a full in-order traversal if the search key is null.

The PDL and program listing are in Appendices B and F.

A use for this routine involves the use of multiple indices for one record file. If n keys are useful for that

record file, n or more permutations or subsets of those keys could be used as the keys for other B-trees. If for example, "department" were one of the key fields, it could be used as the leading portion of the key field for a secondary B-tree index. Thereafter, records could be accessed on the basis of the department key field. This method was used extensively by the author in the system described in the next chapter.

# CHAPTER V

# DISCUSSION OF THE UTILIZATION OF A B-TREE INDEX FOR A PRACTICAL PROBLEM

## Budget and Personnel Records Maintenance System

The file system designed and implemented by the author for the Dean's Office in the College of Arts and Sciences is presented in this chapter. Only enough of the system is presented to show the use of the B-tree index program and the generic access procedure. Basically, the system allows the creation, modification and deletion of personnel records and the analysis of the budget based upon the pay information stored in each personnel record.

During the design phase it had to be determined how the personnel records were to be organized and accessed on disk. Personnel records need to be accessed based upon a name or rank or home department or according to the accounts that contribute to their salaries. Originally, the personnel records were to be stored in an indexed sequential file but there was a need to access the records based upon four different attributes. Indexed sequential organization does not support alternate indices so that method could not be used.

(It would be possible to have several indexed sequential files where one file is the primary file and contains the records themselves and the other files only contain as data the key of the record in the primary file. The primary disadvantage is the amount of reorganization required if the file system is highly volatile.) VSAM with its alternate indices was the natural choice at this point but the University Computer Center does not currently support its use. This led to the design and implementation of the B-tree index program discussed in the previous chapters.

The four B-trees needed are called the NAME, RANK, HOME and ACCTS B-trees. The key field for each B-tree is described below:

1. NAME - catenation of name, home department and rank.
2. RANK - catenation of rank, home department and name.
3. HOME - catenation of home department, rank and name.
4. ACCTS - catenation of an account number and the personnel record number.

The NAME, RANK and HOME B-trees contain different permutations of the same three key fields in a personnel record. All keys in a B-tree index must be unique, which leads to the restriction that no two personnel records may be recorded with the same name, rank and home department. If this were ever necessary, a number could be used for the

middle initial of the name to make each record have unique key fields.

The ACCTS B-tree can be used to access all records that receive money from a given account. It may seem redundant to have the personnel record number in the key as well as in the pointer at level one, but it is necessary to satisfy the requirement of unique keys in the B-tree.

For maintenance purposes, personnel records are accessed by the name field in the record. This leads to a problem if more than one record has the same name. The batch program that allows the maintenance of personnel records treats the problem as an error and produces a message stating that the real-time maintenance routine must be used instead. The real-time routine prompts for the rank and the home to determine exactly which record to modify.

Any new personnel record requires three calls to insert the new keys into the NAME, HOME and RANK B-trees. As many keys as there are unique accounts in the record must be inserted into the ACCTS B-tree.

The method for creating a record in PDL form is:

```
determine name, rank, home and other ID information;
CALL BTINDX(insert,name||home||rank,recno,nameroot);
CALL BTINDX(insert,home||rank||name,recno,homeroot);
CALL BTINDX(insert,rank||home||name,recno,rankroot);
build up pay and account information;
DO for each unique account;
  CALL BTINDX(insert,acct_no||recno,recno,acctroot);
END;
store personnel record in PAFILE at location recno;
```

The deletion of a personnel record requires the reversal of the above procedure. The updating of a record requires no action on the NAME, HOME or RANK B-trees unless one of the key fields in the record is changed. In that case, all three old keys must be deleted and the three new keys must be inserted. Any change to the accounts and pay information causes corresponding deletions and insertions so that when the record is rewritten, the ACCTS B-tree contains an entry for each unique account in the personnel record.

Therefore, personnel records may be accessed based upon a person's name, rank, home department or the accounts that pay them. In fact, it would also be possible to access personnel records based on a combination of attributes as long as those attributes make up the leading portion of the key for the B-tree. This means that the proper record could not be found if only a person's last name and rank were used as the search key. If the last name were used by itself, all the records with that last name could be retrieved. To access all the records in a given class (for example all records with the rank "associate professor") there is a need for a generic access routine. This routine and the logic for it is discussed in Chapter IV.

In the program that prints personnel records, if more than one record has a given name, all occurrences of records with that name are printed. If only a last name is given to the routine, all records with that last name are printed.

In fact, if a single letter is provided as the last name, all records that have a last name that begins with that letter are printed. These actions are the result of the use of the generic access procedure.

Appendices C and D contain PDLs for a batch and a real-time routine that prints a report on full-time equivalents (FTEs[1]) and dollars committed to accounts. The batch routine prints the report for all accounts and the real-time routine prints a report for a single account or subset of accounts. Appendices G and H contain the actual PL/I programs and contain %INCLUDE statements for the inclusion of structure declarations and internal procedures. Therefore, those two programs only show the detail necessary to see how the generic traversal procedure is used.

More information on this file system is available in the Programmer's Guide written by the author and kept in the Dean's Office in the College of Arts and Sciences (3). It contains listings of all the programs in the system and provides more examples of the uses of the B-tree index program and the generic access routine.

Another example of a system where B-trees were used in an information storage and retrieval system is contained in the thesis written by Crotzer (4).

---

[1]The summation of the percentage of full time worked by each employee is called "FTE".

## Summary and Conclusions

A B-tree indexing scheme can be very useful for any applications that store records in a volatile file maintained on direct access auxiliary storage. Such a scheme can greatly increase the speed of programs by reducing the number of records accessed on auxiliary storage. Also, a volatile file does not reduce the performance of the system.

One change in the B-tree index program might be worth looking into. Presently, B-trees with different branching factors or with different key lengths are contained in separate files. For programs that do not know the size of the key that will be used in the index, this is impractical. This requirement could be relaxed by giving the maximum size of a node to the index routine and allowing it to compute what the branching factor must be for a particular B-tree. B-trees with different attributes could still be stored in different files but it would not be necessary if the computed branching factor is of a desirable size.

An area of further research or interest involves the B-tree structure used by Haerder in his General Access Path Structure (6). The general idea allows a list of record identifiers to be stored with a given key. In the system described in this paper, a B-tree could be constructed with RANK as the key attribute. For each rank or key in the B-tree, there could be a list of record identifiers. Such a scheme would reduce the size of the keyfield in the indices

discussed earlier and would provide ready access to all the records that match a certain attribute.

## REFERENCES


(1) Astrahan, M. M. et al. "System R: Relational Approach to Database Management." ACM Transactions on Database Systems, Vol. 1, No. 2 (June, 1976), 97-137.

(2) Bayer, R. and E. McCreight. "Organization and Maintenance of Large Ordered Indexes." Acta Informatica, Vol. 1 (1972), 173-189.

(3) Christian, David D. Programmer's Guide for the Budget and Personnel Records Maintenance System (unpublished). Dean's Office, College of Arts and Sciences, Stillwater, Oklahoma: Oklahoma State University, 1978.

(4) Crotzer, Arthur D. "Efficacy of B-trees in an Information Storage and Retrieval Environment." (unpub. Masters thesis, Oklahoma State University, 1975.)

(5) Davis, William S. "Empirical Behavior of B-trees." (unpub. Masters report, Oklahoma State University, 1974.)

(6) Haerder, Theo. "Implementing a Generalized Access Path Structure for a Relational Database System." ACM Transactions on Database Systems, Vol. 3, No. 3 (September, 1978), 285-298.

(7) Introduction to IBM Direct Access Storage Devices and Organization Methods (GC20-1649-6). New York: International Business Machines Corporation, 1974.

(8) Knuth, Donald E. The Art of Computer Programming, Vol. 3 (1973), 473-480.

(9) OS PL/I Checkout and Optimizing Compilers: Language Reference Manual (GC33-0009-4). New York: International Business Machines Corporation, 1976.

(10) Ramsey, H. Rudy, Michael E. Atwood, and James R. Van Doren. A Comparative Study of Flowcharts and Program Design Languages for the Detailed Procedural Specifications of Computer Programs. Colorado: Scientific Applications, Inc. (portions to be published.)

(11) Van Doren, James R. Data and Storage Structures (unpub. class notes). Stillwater, Oklahoma: Oklahoma State University, 1978.

# APPENDIX A

## PDL DESCRIPTION OF THE B-TREE INDEX PROGRAM

The following section contains a Program Design Language (PDL) description of the B-tree index program.

```
/*
Author:  David Christian
Date:    22 May 1978

Purpose: The purpose of this procedure is to maintain an
   exhaustive index organized into a B-TREE index where the
   lowest level pointers are pointers to the actual records.
   An explanation of this approach is available in:

     Knuth, Donald E.  THE ART OF COMPUTER PROGRAMMING.
        Vol. 3, 473-480.  Reading:  Addison-Wesley, 1973.
     Horowitz, Ellis and Sahni, Sartaj.  FUNDAMENTALS OF
        DATA STRUCTURES.  496-540.  Woodland Hills, Cal.:
        Computer Science Press, 1976.
     Van Doren, James R.  COMSC 5413 Class Notes.  Spring
        1978, Oklahoma State University.

Procedure Descriptions:
   BTINDX - Driver routine.  Chooses the action to be
      performed.
   SEARCH - Searches for a given key at the lowest level and
      stacks pointers to nodes that trace the path to the
      node that contains the given key.
   INSERT - Inserts a new key into the tree at the lowest
      level and then promotes a key if a node splits.
   DELETE - Deletes a key at the lowest level.  Underflow is
      handled by two different methods depending on whether
      the underflow occurs in a lowest level node or not.
      If the rightmost key in a lowest level node is deleted
      the new rightmost key must replace the old one in the
      upper part of the tree.
   GETNODE - Gets a node of the B-TREE and brings it into
      main memory.
   PUTNODE - Puts a node in main memory back into the tree.
```

```
   HDNODE - Creates a new root node with a given key and two
      pointers.
Description of Variables Passed to BTINDX:
   OPCODE - Contents determine whether a search, insertion
      or deletion attempt will be made.
   KEY - Key for search or insertion or deletion.
   POS - The location of the record with key = KEY in the
      record file.
   ROOT - The location of the root node in the B-TREE.
   AVAIL - The location of the first available free node in
      a singly-linked list of such free nodes.
   FLAG - The value of this flag upon return specifies the
      final status of the request specified by OPCODE.  The
      values are:
         1.  operation_completed.
         2.  duplicate_entry.
         3.  key_not_found.
         4.  available_storage_exceeded.
         5.  invalid_opcode.

Description of Internal Variables:
   CURNODE - Current node.
   SIBNODE - Sibling node.
   PARNODE - Parent node.
   CSS - Location in tree of CURNODE.
   SSS - Location in tree of SIBNODE.
   PSS - Location in tree of PARNODE.
   PINCN - Position in CURNODE of first key >= KEY.
   PINPC - Position of pointer in PARNODE to CURNODE.
   PINPS - Position of pointer in PARNODE to SIBNODE.
   PTRS - The pointers in a node of the B-TREE.  PTRS(0) is
      a count of the number of keys in the node.
   KEYS - The keys in a node of the B-TREE.
   STACK - Contains a pointer to a node and a subscript
      value for the location in that node of the first
      key >= KEY.
   NULL - Value considered null for a pointer in the file
      of records.
   NULL_VALUE - Value considered null for a node pointer.
   SAVEIT - Used to contain a node's PTRS(0) value when it
      is to be changed or to hold a sum of PTRS(0) from two
      nodes.
   MAXBRANCHING - The maximum branching allowed in a node.
/*
BTINDX:  PROC(OPCODE, KEY, POS, ROOT, AVAIL, FLAG);
    initialize variables;
    FLAG=operation_completed;
    SELECT(OPCODE);
       WHEN(search) CALL SEARCH;
       WHEN(insert) CALL INSERT;
       WHEN(delete) CALL DELETE;
       OTHERWISE FLAG=invalid_opcode;
    END;
END BTINDX;
```

```
SEARCH: PROC;
    initialize STACK to empty;
    IF ROOT is null THEN POS=0; RETURN; FI;
    CSS=ROOT;
    DO FOREVER;
       CALL GETNODE(CSS,CURNODE);
       PINCN=position in CURNODE of first key >= KEY;
       IF PTRS(PINCN) in CURNODE < 0 THEN
          IF KEY = KEYS(PINCN) in CURNODE THEN
             POS=-PTRS(PINCN) in CURNODE;
          ELSE POS=0; FLAG=key_not_found; FI;
          RETURN;
       FI;
       put CSS and PINCN onto STACK;
       put CURNODE into PARNODE;
       CSS=PTRS(PINPC) in PARNODE;
    END;
END SEARCH;

INSERT: PROC;
    initialize variables;
    LOC=-absolute value of POS;
    IF ROOT=NULL_VALUE THEN
       IF AVAIL=NULL_VALUE
          THEN FLAG=available_storage exceeded;
          RETURN; FI;
       CALL HDNODE(KEY,LOC,NULL);
       RETURN;
    FI;
    CALL SEARCH;
    IF POS¬=0 THEN FLAG=duplicate_entry; RETURN; FI;
  INSRT:
    insert KEY and LOC into CURNODE at position PINCN;
    IF overflow does not exist THEN
       CALL PUTNODE(CSS,CURNODE);
       RETURN;
    FI;
    IF STACK is empty THEN GO TO SPLIT; FI;
    /*
    When sharing keys between siblings care must be taken to
    bring the parent key down into leftmost node before
    shifting if the siblings are not lowest level nodes.
    Then, upon completion, the rightmost key in the leftmost
    node must replace the key that was brought down if the
    siblings are not lowest level nodes.
    */
    IF left sibling exists AND is not full THEN
       perform overflow to the left;
    ELSE
       IF right sibling exists AND is not full THEN
          perform overflow to the right;
       ELSE GO TO SPLIT; FI;
    FI;
    CALL PUTNODE(CSS,CURNODE);
```

```
    CALL PUTNODE(SSS,SIBNODE);
    CALL PUTNODE(PSS,PARNODE);
    RETURN;
  SPLIT:
    IF AVAIL = NULL_VALUE
       FLAG=available_storage_exceeded;  RETURN; FI;
    CALL GETNODE(AVAIL,SIBNODE);
    /*
    CURNODE will be stored in the location for a new node
    and SIBNODE will be stored where CURNODE was stored.
    */
    SSS=CSS;
    CSS=AVAIL;
    AVAIL=PTRS(0) in SIBNODE;
    put upper half of CURNODE into SIBNODE;
    set PTRS(0) in CURNODE and SIBNODE;
    KEY=highest key in CURNODE;
    LOC=CSS;
    IF CURNODE is not a lowest level node THEN
       decrement PTRS(0) in CURNODE; FI;
    CALL PUTNODE(CSS,CURNODE);
    CALL PUTNODE(SSS,SIBNODE);
    IF STACK is empty THEN
       IF AVAIL = NULL_VALUE
          THEN FLAG=available_storage_exceeded;
          RETURN; FI;
       CALL HDNODE(KEY,CSS,SSS);
       RETURN;
    FI;
    CSS=PSS;
    put PARNODE into CURNODE;
    PINCN=PINPC;
    pop STACK;
    IF STACK is not empty THEN
       copy top of STACK into PSS and PINPC;
       CALL GETNODE(PSS,PARNODE);
    FI;
    GO TO INSRT;
END INSERT;

DELETE: PROC;
    initialize variables;
    CALL SEARCH;
    IF POS=0 THEN RETURN; FI;
    delete the key and its pointer in CURNODE;
    IF STACK is empty THEN
       IF CURNODE is empty THEN
          ROOT=NULL_VALUE;
          put CURNODE back on available list;
       FI;
       RETURN;
    FI;
    KEYS(PINPC) in PARNODE=highest key in CURNODE;
    IF CURNODE is not underfull THEN
```

```
      CALL PUTNODE(CSS,CURNODE);
      IF PINCN <= PTRS(0) in CURNODE THEN RETURN;
      GO TO TRACEBACK;
   FI;
UNDERFULL:
   /*
   Share or combine keys with right sibling if it exists.
   */
   IF right sibling exists THEN
      PINPS=PINPC+1;
      SSS=PTRS(PINPS) in PARNODE;
      CALL GETNODE(SSS,SIBNODE);
      SAVEIT=PTRS(0) in CURNODE + PTRS(0) in SIBNODE;
      IF CURNODE is not a lowest level node
         THEN SAVEIT=SAVEIT+1; FI;
      IF SAVEIT > MAXBRANCHING - 1 THEN   /* Share keys */
         IF CURNODE is a lowest level node THEN
            divide keys and pointers between SIBNODE and
               CURNODE;
            KEYS(PINPC) in PARNODE=highest key in CURNODE;
         ELSE
            increment PTRS(0) in CURNODE;
            KEYS(PTRS(0)) in CURNODE =
               KEYS(PINPC) in PARNODE;
            PTRS(PTRS(0)+1) in CURNODE=PTRS(1) in SIBNODE;
            KEYS(PINPC) in PARNODE=KEYS(1) in SIBNODE;
            delete leftmost key and pointer in SIBNODE;
         FI;
         CALL PUTNODE(CSS,CURNODE);
         CALL PUTNODE(SSS,SIBNODE);
         CALL PUTNODE(PSS,PARNODE);
         RETURN;
      ELSE   /* Combine keys */
         IF CURNODE is a lowest level node THEN
            combine CURNODE and SIBNODE into CURNODE;
            KEYS(PINPC) in PARNODE=highest key in CURNODE;
            delete key and pointer to SIBNODE;
         ELSE
            increment PTRS(0) in CURNODE;
            put SIBNODE into CURNODE;
            delete KEYS(PINPC) in PARNODE;
            delete PTRS(PINPS) in PARNODE;
            decrement PTRS(0) in PARNODE;
         FI;
         put SIBNODE back on available list;
         CALL PUTNODE(CSS,CURNODE);
         IF PARNODE is not underfull THEN
            CALL PUTNODE(PSS,PARNODE);
            RETURN;
         FI;
      FI;
   ELSE  /* Share or combine keys with left sibling since
            the right sibling does not exist. CURNODE is
            the rightmost child of PARNODE.  */
```

```
   PINPS=PINPC-1;
   SSS=PTRS(PINPS) in PARNODE;
   CALL GETNODE(SSS,SIBNODE);
   SAVEIT=PTRS(0) in CURNODE + PTRS(0) in SIBNODE;
   IF CURNODE is not a lowest level node
      THEN SAVEIT=SAVEIT+1; FI;
   IF SAVEIT > MAXBRANCHING - 1 THEN   /* Share keys */
      IF CURNODE is a lowest level node THEN
         SAVEIT=PTRS(0) in CURNODE;
         divide keys and pointers between SIBNODE and
            CURNODE;
         KEYS(PINPC) in PARNODE=highest key in CURNODE;
         KEYS(PINPS) in PARNODE=highest key in SIBNODE;
      ELSE
         SAVEIT=1+PTRS(0) in CURNODE;
         shift all keys and pointers in CURNODE one to
            the right;
         increment PTRS(0) in CURNODE;
         KEYS(1) in CURNODE=KEYS(PINPS) in PARNODE;
         PTRS(1) in CURNODE=PTRS(PTRS(0)+1) in SIBNODE;
         KEYS(PINPS) in PARNODE =
            KEYS(PTRS(0)) in SIBNODE;
         decrement PTRS(0) in SIBNODE;
      FI;
      CALL PUTNODE(CSS,CURNODE);
      CALL PUTNODE(SSS,SIBNODE);
      CALL PUTNODE(PSS,PARNODE);
      IF PINCN=SAVEIT THEN GO TO TRACEBACK; FI;
      RETURN;
   ELSE   /* Combine keys */
      IF CURNODE is a lowest level node THEN
         SAVEIT=PTRS(0) in CURNODE;
         combine SIBNODE and CURNODE into SIBNODE;
         KEYS(PINPS) in PARNODE=highest key in SIBNODE;
      ELSE
         SAVEIT=1+PTRS(0) in CURNODE;
         increment PTRS(0) in SIBNODE;
         KEYS(PTRS(0)) in SIBNODE =
            KEYS(PINPS) in PARNODE;
         combine CURNODE into SIBNODE;
      FI;
      decrement PTRS(0) in PARNODE;
      put CURNODE back on available list;
      CALL PUTNODE(SSS,SIBNODE);
      IF PARNODE is not underfull THEN
         CALL PUTNODE(PSS,PARNODE);
         IF PINCN = SAVEIT THEN GO TO TRACEBACK; FI;
         RETURN;
      FI;
      CSS=SSS;
   FI;
FI;
/* PARNODE is possibly underfull */
pop STACK;
```

```
      IF STACK is empty THEN
         IF PARNODE is empty THEN
            ROOT=CSS;
            put PARNODE back onto available list;
         ELSE CALL PUTNODE(PSS,PARNODE); FI;
         RETURN;
      FI;
      put PARNODE into CURNODE;
      copy top of STACK into PSS and PINPC;
      CALL GETNODE(PSS,PARNODE);
      GO TO UNDERFULL;
      /*
      A rightmost key was deleted and its occurrence in the
      rest of the index must be changed to the new rightmost
      key.
      */
    TRACEBACK:
      DO FOREVER;
         pop STACK;
         IF PINPC <= PTRS(0) in PARNODE OR STACK is empty
            THEN RETURN; FI;
         put PARNODE into CURNODE;
         copy top of STACK into PSS and PINPC;
         CALL GETNODE(PSS,PARNODE);
         KEYS(PINPC) in PARNODE=KEYS(PTRS(0)+1) in CURNODE;
         CALL PUTNODE(PSS,PARNODE);
      END;
END DELETE;

GETNODE: PROC(SS,NODE);
      read node number SS into NODE;
END GETNODE;

PUTNODE: PROC(SS,NODE);
      put NODE into node number SS;
END PUTNODE;

HDNODE: PROC(KEY,SS1,SS2);
      CSS=AVAIL;
      CALL GETNODE(CSS,CURNODE);
      AVAIL=PTRS(0) in CURNODE;
      PTRS(0) in CURNODE=1;
      KEYS(1) in CURNODE=KEY;
      PTRS(1) in CURNODE=SS1;
      PTRS(2) in CURNODE=SS2;
      CALL PUTNODE(CSS,CURNODE);
      ROOT=CSS;
END HDNODE;
```

# APPENDIX B

## PDL DESCRIPTION OF THE GENERIC PROGRAM

The following section contains a PDL description of the generic traversal procedure described in Chapter IV.

```
GENTRAV: PROC(PKEY,INPROC,ROOT);
/*
Author - David Christian
Date   - 24 Sept 1978

Purpose - This procedure does an inorder traversal of a
   B-tree starting at the first key that matches generically
   the partial key passed in.  INPROC is called to process
   each record that is found.  A full inorder traversal is
   performed if PKEY = ' ';
*/
    IF PKEY=' ' THEN J=0;
       ELSE J=length of the nonblank portion of PKEY;
    CALL TRAVINGENTRAV(ROOT);

TRAVINGENTRAV: PROC(NODENO);
    read node number NODENO into NODE;
    DO I=1 TO PTRS(0)+1 UNTIL(a key in the node > PKEY);
       IF the PKEY matches the substring of key(I) of
       length J in NODE
       THEN IF it is a lowest level node
          THEN CALL INPROC(key(I),-PTRS(I));
          ELSE CALL TRAVINGENTRAV(PTRS(I)); FI;
       FI;
    END;
END TRAVINGENTRAV;
END GENTRAV;
```

## APPENDIX C

### PDL DESCRIPTION OF A BATCH PROGRAM

The following section contains a PDL description of one of the batch report generating programs of the file system described in Chapter V that uses the generic traversal procedure.

```
$FTE$: PROC;
/*
Author - David Christian
Date   - 24 Sept 1978

Purpose - This procedure prints ftes and dollars by rank for
   all accounts that contribute money.  An example output is
   contained in the system proposal.

/*
    Initialize ACTROOT;
    CALL GENTRAV(' ',BLDFTE$,ACTROOT);
       /* GENTRAV has its own PDL in PDLLIB */
    CALL BLDFTE$(0,' ');

BLDFTE$: PROC(key,RECNO);
    IF the key indicates a new account has been started
       THEN print the dollars and ftes built up for each
          rank for the summer, fall and spring semesters;
          save the value of the new account in CURACCT;
    FI;
    IF key=0 THEN RETURN;
    read record RECNO from PAFILE;
    compute and save the number of dollars and ftes
       committed each month to CURRACT;
    Add these amounts to the amounts already summed for that
    rank if it already appears;  otherwise create an entry
    for that rank and initialize it with the figures just
    obtained;
END BLDFTE$;
END $FTE$;
```

## APPENDIX D

### PDL DESCRIPTION OF A REAL-TIME PROGRAM

The following section contains a PDL description of one of the real-time report generating programs of the file system described in Chapter V that uses the generic traversal procedure. This program is the real-time counterpart to the program described in Appendix C.

```
FTE$: PROC;
/*
Author - David Christian
Date   - 24 Sept 1978

Purpose - This procedure prints ftes and dollars by rank for
   an account. An example output is contained in the
   system proposal.

/*
    Initialize ACTROOT;
    determine account and put value into key;
    CALL GENTRAV(key,BLDFTE$,ACTROOT);
       /* GENTRAV has its own PDL in PDLLIB */
    CALL BLDFTE$(0,' ');

BLDFTE$: PROC(key,RECNO);
    IF the key indicates a new account has been started
       THEN print the dollars and ftes built up for each
          rank for the summer, fall and spring semesters;
          save the value of the new account in CURACCT;
    FI;
    IF key=0 THEN RETURN;
    read record RECNO from PAFILE;
    compute and save the number of dollars and ftes
       committed each month to CURRACT;
    Add these amounts to the amounts already summed for
    that rank if it appears; otherwise, make an entry for
    that rank and initialize it with the values just
    computed above;
END BLDFTE$;
END FTE$;
```

## APPENDIX E

## PL/I B-TREE INDEX PROGRAM

The following section contains the PL/I program that maintains a B-tree index.

```
BTINDX: PROC(OP,KEY,POS,ROOT,AVAIL,AVCNT,MAXB,KLEN,FLAG,
            INFILE,INNODE);
   /*
BTINDX IS AN IMPLEMENTATION OF AN EXHAUSTIVE INDEX ORGANIZED
AS A B-TREE INDEX.  THE INDEX BLOCKS ARE STORED IN ONE
DIRECT ACCESS FILE AND THE ACTUAL RECORDS IN A DIFFERENT
DIRECT ACCESS FILE.  A POINTER IN AN INDEX BLOCK IS A
POINTER TO ANOTHER INDEX BLOCK IF THE VALUE IS >=0.
OTHERWISE, THE ABSOLUTE VALUE OF THE POINTER IS THE RECORD
NUMBER IN THE OTHER FILE.
PARAMETERS:
   OP       - SPECIFIES FUNCTION TO BE PERFORMED.
   KEY      - KEY FOR RETRIEVAL, INSERTION OR DELETION.
   POS      - RELATIVE RECORD NUMBER OF ACTUAL RECORD.
   ROOT     - RELATIVE RECORD NUMBER OF ROOT NODE OF INDEX.
   AVAIL    - RELATIVE RECORD NUMBER OF FIRST AVAILABLE INDEX
              BLOCK IN A SINGLY LINKED LIST OF AVAILABLE
              BLOCKS.
   AVCNT    - NUMBER OF AVAILABLE BLOCKS LEFT.
   MAXB     - MAXIMUM BRANCHING IN AN INDEX BLOCK.  (MUST BE
              LESS THAN WHAT BLOCK WILL PHYSICALLY CONTAIN AS
              AN EXTRA KEY AND POINTER MUST BE PRESENT FOR
              THE MAINTENANCE ROUTINES TO WORK.
   KLEN     - MAXIMUM LENGTH OF KEY.
   FLAG     - STATUS CODE FOR ATTEMPTED FUNCTION.
   INFILE   - THIS FILE CONTAINS THE INDEX BLOCKS.
A DETAILED PDL DESCRIPTION OF THIS PROGRAM IS AVAILABLE AND
SHOULD BE CONSULTED FOR DETAILS OF HOW THIS PROGRAM WORKS.
   */
   DECLARE
   /*
      PARAMETER VARIABLES.
   */
   OP CHAR(*),
   KEY CHAR(*),
   (POS,ROOT,AVAIL,AVCNT,MAXB,KLEN,FLAG) FIXED BIN(15,0),
   INFILE FILE VARIABLE;
```

```
/*************************************************************/
/*                                                           */
/* THE FOLLOWING DECLARATION MAKES IT POSSIBLE TO REFER TO   */
/* AN AREA OF GLOBAL STORAGE THAT CONTAINS THE VARIABLES     */
/* USED BY THE B-TREE INDEX PROGRAM.  MAXB (MAXIMUM          */
/* BRANCHING FOR A NODE) AND KLEN (THE LENGTH OF A KEY IN    */
/* THE NODE) MUST BE KNOWN INTHE BLOCK BEFORE THIS           */
/* DECLARATION IS MADE.                                      */
/*                                                           */
/*************************************************************/
   DECLARE
   1 INNODE,                    /* GLOBAL B-TREE DATA        */
     (2 STKPT,                  /* DEPTH OF STACK            */
      2 NODENO(*),              /* RELATIVE RECORD NUMBER    */
      2 PINN(*),                /* POINTER IN A NODE         */
      2 PINCN,                  /* POINTER IN CURNODE        */
      2 PINPC,                  /* PARENT->CURRENT POINTER   */
      2 PINPS)                  /* PARENT->SIBLING POINTER   */
         FIXED BIN(15),         /*                           */
      2 NODES(3),               /* THE THREE NODES           */
        (3 X, 3 Y, 3 Z)         /* NECESSARY FOR REFER OPTION*/
            FIXED BIN(15),      /* IN BASED STRUCTURES LATER */
         3 SUB,                 /* NODE AS ON DISK           */
            4 PTRS(*)           /* SET OF POINTERS IN A NODE */
               FIXED BIN(15),/*                             */
            4 KEYS(*)           /* SET OF KEYS IN A NODE     */
               CHAR(*),         /*                           */
     (2 P,                      /* POINTERS USED FOR BASED   */
      2 PCUR,                   /* STRUCTURES.               */
      2 PSIB,                   /* P IS USED WHEN TWO        */
      2 PPAR) POINTER;          /* POINTERS ARE SWITCHED.    */
/*************************************************************/
/*                                                           */
/* THESE DECLARATIONS PROVIDE THE LOGICAL ACCESS TO THE      */
/* PHYSICAL DECLARATION OF NODES ABOVE.                      */
/*                                                           */
/*************************************************************/
   /*
      THE FOLLOWING ARE LOCAL INTERNAL VARIABLES.
   */
   DECLARE
   1 CURNODE BASED(PCUR),       /* THESE STRUCTURES OVERLAY  */
     (2 X, 2 Y, 2 Z)            /* THE SUBSTRUCTURE 'NODES'  */
         FIXED BIN(15),         /* IN INNODE.  THIS WAY, IF  */
      2 CURN,                   /* THE CURRENT NODE BECOMES  */
         3 PTRS(0:MAXB+1        /* THE PARENT NODE, ONLY     */
           REFER(CURNODE.X)) /* POINTERS ARE CHANGED.  THE*/
           FIXED BIN(15),       /* REFER OPTION IS REQUIRED  */
         3 KEYS(MAXB            /* FOR BASED STRUCTURES WITH */
           REFER(CURNODE.Y)) /* VARIABLY DIMENSIONED SUB- */
           CHAR(KLEN            /* STRUCTURES.               */
           REFER(CURNODE.Z)),/*                             */
   1 SIBNODE BASED(PSIB),
     (2 X, 2 Y, 2 Z) FIXED BIN(15),
```

```
      2 SIBN,
         3 PTRS(0:MAXB+1 REFER(SIBNODE.X)) FIXED BIN(15),
         3 KEYS(MAXB REFER(SIBNODE.Y))
            CHAR(KLEN REFER(SIBNODE.Z)),
   1 PARNODE BASED(PPAR),
     (2 X, 2 Y, 2 Z) FIXED BIN(15),
      2 PARN,
         3 PTRS(0:MAXB+1 REFER(PARNODE.X)) FIXED BIN(15),
         3 KEYS(MAXB REFER(PARNODE.Y))
            CHAR(KLEN REFER(PARNODE.Z)),
   (I,J,K,LOC,SAVEIT) FIXED BIN(15,0),
   KY CHAR(KLEN),
   NULL FIXED BIN(15,0) INIT(-1),
   (LWSTLVLNODE,LFTSIBXSTS,RITSIBXSTS) BIT(1) ALIGNED,
   TRUE BIT(1) ALIGNED INIT('1'B),
   (FLOOR,CEIL,ABS,ADDR) BUILTIN,
   (CSS,SSS,PSS) FIXED BIN(15,0);
/*  THE FOLLOWING INITIALIZES VARIABLES THAT MAKE
    DECLARATIONS WORK  */
   PCUR=ADDR(NODES.X(1));
   PSIB=ADDR(NODES.X(2));
   PPAR=ADDR(NODES.X(3));
   CURNODE.X,SIBNODE.X,PARNODE.X=MAXB+1;
   CURNODE.Y,SIBNODE.Y,PARNODE.Y=MAXB;
   CURNODE.Z,SIBNODE.Z,PARNODE.Z=KLEN;
   FLAG=1;
   KY=KEY;
   SELECT(OP);
      WHEN('INSERT') GO TO INSERT;
      WHEN('DELETE') GO TO DELETE;
      WHEN('SEARCH') CALL SEARCH;
      OTHERWISE FLAG=5;
   END;
   RETURN;
   /**********************/
   /*                    */
   /* INSERTION ROUTINE. */
   /*                    */
   /**********************/
INSERT:
   LOC=-ABS(POS);
   IF ROOT=-1 THEN DO;
      IF AVAIL=-1 THEN FLAG=4;
      ELSE CALL HDNODE(KY,LOC,NULL);
      RETURN;
   END;
   CALL SEARCH;
   IF POS¬=0 THEN DO;  /* KEY ALREADY EXISTS */
      POS=-LOC;
      FLAG=2;
      RETURN;
   END;
   POS=-LOC;
   FLAG=1;
```

```
INSRT:
  /*
     INSERT KY AND LOC INTO CURNODE AT POSITION PINCN.
  */
  CURNODE.PTRS(CURNODE.PTRS(0)+2) =
     CURNODE.PTRS(CURNODE.PTRS(0)+1);
  DO I=CURNODE.PTRS(0) TO PINCN BY -1;
     CURNODE.KEYS(I+1)=CURNODE.KEYS(I);
     CURNODE.PTRS(I+1)=CURNODE.PTRS(I);
  END;
  CURNODE.KEYS(PINCN)=KY;
  CURNODE.PTRS(PINCN)=LOC;
  CURNODE.PTRS(0)=CURNODE.PTRS(0)+1;
  IF CURNODE.PTRS(0)<MAXB THEN DO;  /* NO OVERFLOW */
     CALL PUTNODE(CSS,1);
     RETURN;
  END;
  /*
     OVERFLOW HAS OCCURRED.  SHARE OVERFLOW WITH  LEFT OR
     RIGHT SIBLING IF POSSIBLE.
  */
  IF STKPT=0 THEN GO TO SPLIT;
  LWSTLVLNODE=CURNODE.PTRS(1)<0;
  LFTSIBXSTS=PINPC>1;
  RITSIBXSTS=PINPC<=PARNODE.PTRS(0);
  IF LFTSIBXSTS THEN SSS=PARNODE.PTRS(PINPC-1);
  ELSE SSS=PARNODE.PTRS(PINPC+1);
  CALL GETNODE(SSS,2);
  IF LFTSIBXSTS & SIBNODE.PTRS(0)<MAXB-1 THEN DO;
     /*
        PERFORM OVERFLOW SHARING TO THE LEFT.
     */
     I=FLOOR((CURNODE.PTRS(0)-SIBNODE.PTRS(0))/2);
     IF ¬LWSTLVLNODE THEN DO;
        SIBNODE.KEYS(SIBNODE.PTRS(0)+1) =
           PARNODE.KEYS(PINPC-1);
        SIBNODE.PTRS(0)=SIBNODE.PTRS(0)+1;
     END;
     DO J=1 TO I;
        SIBNODE.PTRS(0)=SIBNODE.PTRS(0)+1;
        SIBNODE.KEYS(SIBNODE.PTRS(0))=CURNODE.KEYS(J);
        SIBNODE.PTRS(SIBNODE.PTRS(0))=CURNODE.PTRS(J);
     END;
     DO J=I+1 TO MAXB;
        CURNODE.KEYS(J-I)=CURNODE.KEYS(J);
        CURNODE.PTRS(J-I)=CURNODE.PTRS(J);
     END;
     CURNODE.PTRS(0)=CURNODE.PTRS(0)-I;
     CURNODE.PTRS(CURNODE.PTRS(0)+1)=CURNODE.PTRS(MAXB+1);
     J=SIBNODE.PTRS(0);
     IF ¬LWSTLVLNODE THEN SIBNODE.PTRS(0) =
        SIBNODE.PTRS(0)-1;
     PARNODE.KEYS(PINPC-1)=SIBNODE.KEYS(J);
  END;
```

```
   ELSE IF RITSIBXSTS & SIBNODE.PTRS(0)<MAXB-1 THEN DO;
      /*
         PERFORM OVERFLOW SHARING TO THE RIGHT.
      */
      I=FLOOR((CURNODE.PTRS(0)-SIBNODE.PTRS(0))/2);
      K=SIBNODE.PTRS(0);
      SIBNODE.PTRS(K+I+1)=SIBNODE.PTRS(K+1);
      DO J=K TO 1 BY -1;
         SIBNODE.KEYS(J+I)=SIBNODE.KEYS(J);
         SIBNODE.PTRS(J+I)=SIBNODE.PTRS(J);
      END;
      IF ¬LWSTLVLNODE THEN DO;
         SIBNODE.KEYS(I)=PARNODE.KEYS(PINPC);
         SIBNODE.PTRS(I)=CURNODE.PTRS(MAXB+1);
         SIBNODE.PTRS(0)=SIBNODE.PTRS(0)+1;
         I=I-1;
      END;
      SIBNODE.PTRS(0)=SIBNODE.PTRS(0)+I;
      K=MAXB-I;
      DO J=1 TO I;
         SIBNODE.KEYS(J)=CURNODE.KEYS(K+J);
         SIBNODE.PTRS(J)=CURNODE.PTRS(K+J);
      END;
      CURNODE.PTRS(0)=K;
      PARNODE.KEYS(PINPC)=CURNODE.KEYS(CURNODE.PTRS(0));
      IF ¬LWSTLVLNODE THEN CURNODE.PTRS(0)=K-1;
   END;
   ELSE GO TO SPLIT;
   CALL PUTNODE(CSS,1);
   CALL PUTNODE(SSS,2);
   CALL PUTNODE(PSS,3);
   RETURN;
SPLIT: /* OVERFLOW OCCURRED IN AT LEAST THE LOWEST LEVEL */
   IF AVCNT<STKPT+1 THEN DO;/* AVAILABLE STORAGE EXCEEDED */
      FLAG=4;
      RETURN;
   END;
   CALL GETNODE(AVAIL,2);
   SSS=CSS;
   CSS=AVAIL;
   AVAIL=SIBNODE.PTRS(0);
   AVCNT=AVCNT-1;
   I=CEIL(CURNODE.PTRS(0)/2);
   SIBNODE.PTRS(0)=0;
   DO J=I+1 TO MAXB;
      SIBNODE.PTRS(0)=SIBNODE.PTRS(0)+1;
      SIBNODE.KEYS(SIBNODE.PTRS(0))=CURNODE.KEYS(J);
      SIBNODE.PTRS(SIBNODE.PTRS(0))=CURNODE.PTRS(J);
   END;
   SIBNODE.PTRS(SIBNODE.PTRS(0)+1)=CURNODE.PTRS(MAXB+1);
   KY=CURNODE.KEYS(I);
   LOC=CSS;
   IF CURNODE.PTRS(1)<0 THEN CURNODE.PTRS(0)=I;
   ELSE CURNODE.PTRS(0)=I-1;
```

```
   CALL PUTNODE(CSS,1);
   CALL PUTNODE(SSS,2);
   IF STKPT=0 THEN DO;
      CALL HDNODE(KY,CSS,SSS);
      RETURN;
   END;
   /*
      PUT PARNODE INTO CURNODE.
   */
   CSS=PSS;
   PINCN=PINPC;
   P=PCUR;
   PCUR=PPAR;
   PPAR=P;
   STKPT=STKPT-1;
   IF STKPT>0 THEN DO;
      PSS=NODENO(STKPT);
      PINPC=PINN(STKPT);
      CALL GETNODE(PSS,3);
   END;
   GO TO INSRT;
   /********************/
   /*                  */
   /* DELETION ROUTINE */
   /*                  */
   /********************/
DELETE:
   CALL SEARCH;
   IF POS=0 THEN RETURN;
   /*
      DELETE THE KEY AND ITS POINTER IN CURNODE.
   */
   DO I=PINCN TO CURNODE.PTRS(0);
      CURNODE.KEYS(I)=CURNODE.KEYS(I+1);
      CURNODE.PTRS(I)=CURNODE.PTRS(I+1);
   END;
   CURNODE.PTRS(0)=CURNODE.PTRS(0)-1;
   IF STKPT=0 THEN DO;
      IF CURNODE.PTRS(0)=0 THEN DO;
         ROOT=-1;
         CURNODE.PTRS(0)=AVAIL;
         AVAIL=CSS;
         AVCNT=AVCNT+1;
      END;
      CALL PUTNODE(CSS,1);
      RETURN;
   END;
   IF CURNODE.PTRS(0)¬=0 THEN
      PARNODE.KEYS(PINPC)=CURNODE.KEYS(CURNODE.PTRS(0));
   IF CURNODE.PTRS(0) >= FLOOR(MAXB/2) THEN DO;
      CALL PUTNODE(CSS,1);
      IF PINCN > CURNODE.PTRS(0) THEN DO;
         CALL PUTNODE(PSS,3);
         GO TO TRACEBACK;
```

```
      END;
      RETURN;
   END;
UNDERFULL:
   /*
      SHARE OR COMBINE KEYS WITH RIGHT SIBLING IF IT EXISTS.
   */
   IF PINPC <= PARNODE.PTRS(0) THEN DO;
      PINPS=PINPC+1;
      SSS=PARNODE.PTRS(PINPS);
      CALL GETNODE(SSS,2);
      SAVEIT=CURNODE.PTRS(0)+SIBNODE.PTRS(0);
      IF CURNODE.PTRS(1)>=0 THEN SAVEIT=SAVEIT+1;
      IF SAVEIT > MAXB-1 THEN DO;
      /*
        SHARE KEYS.
      */
         IF CURNODE.PTRS(1)<0 THEN DO;
            I=FLOOR((SIBNODE.PTRS(0)-CURNODE.PTRS(0))/2);
            DO J=1 TO I;
               CURNODE.KEYS(CURNODE.PTRS(0)+J) =
                  SIBNODE.KEYS(J);
               CURNODE.PTRS(CURNODE.PTRS(0)+J) =
                  SIBNODE.PTRS(J);
            END;
            DO J=I+1 TO SIBNODE.PTRS(0);
               SIBNODE.PTRS(J-I)=SIBNODE.PTRS(J);
               SIBNODE.KEYS(J-I)=SIBNODE.KEYS(J);
            END;
            CURNODE.PTRS(0)=CURNODE.PTRS(0)+I;
            SIBNODE.PTRS(0)=SIBNODE.PTRS(0)-I;
            PARNODE.KEYS(PINPC) =
               CURNODE.KEYS(CURNODE.PTRS(0));
         END;
         ELSE DO;
            CURNODE.PTRS(0)=CURNODE.PTRS(0)+1;
            CURNODE.KEYS(CURNODE.PTRS(0)) =
               PARNODE.KEYS(PINPC);
            CURNODE.PTRS(CURNODE.PTRS(0)+1)=SIBNODE.PTRS(1);
            PARNODE.KEYS(PINPC)=SIBNODE.KEYS(1);
            DO I=1 TO SIBNODE.PTRS(0);
               SIBNODE.KEYS(I)=SIBNODE.KEYS(I+1);
               SIBNODE.PTRS(I)=SIBNODE.PTRS(I+1);
            END;
            SIBNODE.PTRS(0)=SIBNODE.PTRS(0)-1;
         END;
         CALL PUTNODE(CSS,1);
         CALL PUTNODE(SSS,2);
         CALL PUTNODE(PSS,3);
         RETURN;
      END;
      ELSE DO;
```

```
   /*
      COMBINE KEYS.
   */
      IF CURNODE.PTRS(1)<0 THEN DO;
         DO I=1 TO SIBNODE.PTRS(0);
            CURNODE.PTRS(CURNODE.PTRS(0)+I) =
               SIBNODE.PTRS(I);
            CURNODE.KEYS(CURNODE.PTRS(0)+I) =
               SIBNODE.KEYS(I);
         END;
         CURNODE.PTRS(0)=CURNODE.PTRS(0)+SIBNODE.PTRS(0);
      END;
      ELSE DO;
         DO I=1 TO SIBNODE.PTRS(0)+1;
            CURNODE.KEYS(CURNODE.PTRS(0)+I+1) =
               SIBNODE.KEYS(I);
            CURNODE.PTRS(CURNODE.PTRS(0)+I+1) =
               SIBNODE.PTRS(I);
         END;
         CURNODE.PTRS(0) =
            CURNODE.PTRS(0)+SIBNODE.PTRS(0)+1;
      END;
      DO I=PINPC TO PARNODE.PTRS(0);
         PARNODE.KEYS(I)=PARNODE.KEYS(I+1);
         PARNODE.PTRS(I+1)=PARNODE.PTRS(I+2);
      END;
      PARNODE.PTRS(0)=PARNODE.PTRS(0)-1;
      SIBNODE.PTRS(0)=AVAIL;
      AVAIL=SSS;
      AVCNT=AVCNT+1;
      CALL PUTNODE(SSS,2);
      CALL PUTNODE(CSS,1);
      IF PARNODE.PTRS(0)>=FLOOR(MAXB/2) THEN DO;
         CALL PUTNODE(PSS,3);
         RETURN;
      END;
   END;
END;
/*
   CURNODE IS RIGHTMOST CHILD OF PARNODE.  SHARE OR
   COMBINE CURNODE WITH LEFT SIBLING.
*/
ELSE DO;
   PINPS=PINPC-1;
   SSS=PARNODE.PTRS(PINPS);
   CALL GETNODE(SSS,2);
   SAVEIT=CURNODE.PTRS(0)+SIBNODE.PTRS(0);
   IF CURNODE.PTRS(1)>=0 THEN SAVEIT=SAVEIT+1;
   IF SAVEIT > MAXB-1 THEN DO;
   /*
      SHARE KEYS.
   */
      IF CURNODE.PTRS(1)<0 THEN DO;
         SAVEIT=CURNODE.PTRS(0);
```

```
      I=FLOOR((SIBNODE.PTRS(0)-CURNODE.PTRS(0))/2);
      DO J=CURNODE.PTRS(0) TO 1 BY -1;
         CURNODE.KEYS(J+I)=CURNODE.KEYS(J);
         CURNODE.PTRS(J+I)=CURNODE.PTRS(J);
      END;
      DO J=1 TO I;
         CURNODE.KEYS(J) =
            SIBNODE.KEYS(SIBNODE.PTRS(0)-I+J);
         CURNODE.PTRS(J) =
            SIBNODE.PTRS(SIBNODE.PTRS(0)-I+J);
      END;
      CURNODE.PTRS(0)=CURNODE.PTRS(0)+I;
      SIBNODE.PTRS(0)=SIBNODE.PTRS(0)-I;
      PARNODE.KEYS(PINPC) =
         CURNODE.KEYS(CURNODE.PTRS(0));
      PARNODE.KEYS(PINPS) =
         SIBNODE.KEYS(SIBNODE.PTRS(0));
   END;
   ELSE DO;
      SAVEIT=CURNODE.PTRS(0)+1;
      DO I=CURNODE.PTRS(0)+1 TO 1 BY -1;
         CURNODE.KEYS(I+1)=CURNODE.KEYS(I);
         CURNODE.PTRS(I+1)=CURNODE.PTRS(I);
      END;
      CURNODE.PTRS(0)=CURNODE.PTRS(0)+1;
      CURNODE.KEYS(1)=PARNODE.KEYS(PINPS);
      CURNODE.PTRS(1)=SIBNODE.PTRS(SIBNODE.PTRS(0)+1);
      PARNODE.KEYS(PINPS) =
         SIBNODE.KEYS(SIBNODE.PTRS(0));
      SIBNODE.PTRS(0)=SIBNODE.PTRS(0)-1;
   END;
   CALL PUTNODE(CSS,1);
   CALL PUTNODE(SSS,2);
   CALL PUTNODE(PSS,3);
   IF PINCN=SAVEIT THEN GO TO TRACEBACK;
   RETURN;
END;
ELSE DO;
/*
   COMBINE KEYS.
*/
   IF CURNODE.PTRS(1)<0 THEN DO;
      SAVEIT=CURNODE.PTRS(0);
      DO I=1 TO CURNODE.PTRS(0);
         SIBNODE.KEYS(SIBNODE.PTRS(0)+I) =
            CURNODE.KEYS(I);
         SIBNODE.PTRS(SIBNODE.PTRS(0)+I) =
            CURNODE.PTRS(I);
      END;
      SIBNODE.PTRS(0)=SIBNODE.PTRS(0)+CURNODE.PTRS(0);
      PARNODE.KEYS(PINPS) =
         SIBNODE.KEYS(SIBNODE.PTRS(0));
   END;
   ELSE DO;
```

```
            SAVEIT=CURNODE.PTRS(0)+1;
            SIBNODE.KEYS(SIBNODE.PTRS(0)+1) =
               PARNODE.KEYS(PINPS);
            DO I=1 TO CURNODE.PTRS(0)+1;
               SIBNODE.KEYS(SIBNODE.PTRS(0)+I+1) =
                  CURNODE.KEYS(I);
               SIBNODE.PTRS(SIBNODE.PTRS(0)+I+1) =
                  CURNODE.PTRS(I);
            END;
            SIBNODE.PTRS(0) =
               SIBNODE.PTRS(0)+CURNODE.PTRS(0)+1;
            PARNODE.KEYS(PINPS) =
               SIBNODE.KEYS(SINBODE.PTRS(0)+1);
         END;
         PARNODE.PTRS(0)=PARNODE.PTRS(0)-1;
         CURNODE.PTRS(0)=AVAIL;
         AVAIL=CSS;
         AVCNT=AVCNT+1;
         CALL PUTNODE(CSS,1);
         CALL PUTNODE(SSS,2);
         IF PARNODE.PTRS(0)>=FLOOR(MAXB/2) THEN DO;
            CALL PUTNODE(PSS,3);
            IF PINCN>SAVEIT THEN GO TO TRACEBACK;
            RETURN;
         END;
         CSS=SSS;
      END;
   END;
   /*
      PARNODE IS POSSIBLY UNDERFULL.
   */
   STKPT=STKPT-1;
   IF STKPT=0 THEN DO;
      IF PARNODE.PTRS(0)=0 THEN DO;
         ROOT=CSS;
         PARNODE.PTRS(0)=AVAIL;
         AVAIL=PSS;
         AVCNT=AVCNT+1;
      END;
      CALL PUTNODE(PSS,3);
      RETURN;
   END;
   P=PCUR;
   PCUR=PPAR;
   PPAR=P;
   PINCN=PINPC;
   CSS=PSS;
   PSS=NODENO(STKPT);
   PINPC=PINN(STKPT);
   CALL GETNODE(PSS,3);
   GO TO UNDERFULL;
```

```
   /*
      A RIGHTMOST KY HAS BEEN DELETED AND ITS OCCURRENCE IN
      THE REST OF THE TREE MUST BE CHANGED.
   */
TRACEBACK:
   DO WHILE(TRUE);
      STKPT=STKPT-1;
      IF PINPC<=PARNODE.PTRS(0) | STKPT=0 THEN RETURN;
      P=PCUR;
      PCUR=PPAR;
      PPAR=P;
      PINCN=PINPC;
      CSS=PSS;
      PSS=NODENO(STKPT);
      PINPC=PINN(STKPT);
      CALL GETNODE(PSS,3);
      PARNODE.KEYS(PINPC)=CURNODE.KEYS(CURNODE.PTRS(0)+1);
      CALL PUTNODE(PSS,3);
   END;
RETURN;  /* END OF DELETE ROUTINE. */
   /*******************/
   /*                 */
   /* SEARCH ROUTINE. */
   /*                 */
   /*******************/
SEARCH: PROC;
   DECLARE (LAST,K) FIXED BIN(15,0);
   STKPT=0;
   POS=0;
   IF ROOT=-1 THEN RETURN;
   CSS=ROOT;
   /*
      FIND KY BY SEARCHING DOWN TO LOWEST LEVEL.
   */
   DO WHILE(TRUE);
      CALL GETNODE(CSS,1);
      /*
         BINARY SEARCH TO FIND THE FIRST KY IN CURNODE >= KY
      */
      PINCN=1;
      LAST=CURNODE.PTRS(0);
      DO WHILE(PINCN<=LAST);
         K=FLOOR((PINCN+LAST)/2);
         SELECT;
            WHEN(KY<CURNODE.KEYS(K)) LAST=K-1;
            WHEN(KY>CURNODE.KEYS(K)) PINCN=K+1;
            OTHERWISE DO;
               PINCN=K;
               LAST=K-1;
            END;
         END;
      END;
```

```
      /*
         QUIT IF AT LOWEST LEVEL.
      */
      IF CURNODE.PTRS(1)<0 THEN DO;
         IF KY=CURNODE.KEYS(PINCN) & PINCN<=CURNODE.PTRS(0)
            THEN POS=-CURNODE.PTRS(PINCN);
            ELSE FLAG=3;
         RETURN;
      END;
      /*
         PUT CSS AND PINCN ONTO STACK.
      */
      STKPT=STKPT+1;
      NODENO(STKPT)=CSS;
      PINN(STKPT)=PINCN;
      /*
         CAUSE CURNODE TO BECOME PARNODE.
      */
      P=PPAR;
      PPAR=PCUR;
      PCUR=P;
      PSS=CSS;
      PINPC=PINCN;
      /*
         PREPARE TO GET NEXT NODE.
      */
      CSS=PARNODE.PTRS(PINPC);
   END;
END SEARCH;
   /************************/
   /*                      */
   /* READ NODE ROUTINE.   */
   /*                      */
   /************************/
GETNODE: PROC(SS,NODE);
   DECLARE (SS,NODE) FIXED BIN(15,0);
   SELECT(NODE);
      WHEN(1) READ FILE(INFILE) INTO(CURN) KEY(SS);
      WHEN(2) READ FILE(INFILE) INTO(SIBN) KEY(SS);
      WHEN(3) READ FILE(INFILE) INTO(PARN) KEY(SS);
      OTHERWISE STOP;
   END;
END GETNODE;
   /************************/
   /*                      */
   /* STORE NODE ROUTINE.  */
   /*                      */
   /************************/
PUTNODE: PROC(SS,NODE);
   DECLARE (SS,NODE) FIXED BIN(15,0);
   SELECT(NODE);
      WHEN(1) WRITE FILE(INFILE) FROM(CURN) KEYFROM(SS);
      WHEN(2) WRITE FILE(INFILE) FROM(SIBN) KEYFROM(SS);
      WHEN(3) WRITE FILE(INFILE) FROM(PARN) KEYFROM(SS);
```

```
      OTHERWISE STOP;
   END;
END PUTNODE;
   /**********************/
   /*                    */
   /* HEAD NODE ROUTINE. */
   /*                    */
   /**********************/
HDNODE: PROC(KY,SS1,SS2);
   DECLARE
   KY CHAR(*),
   (SS1,SS2,I) FIXED BIN(15,0);
   I=AVAIL;
   CALL GETNODE(I,1);
   AVAIL=CURNODE.PTRS(0);
   AVCNT=AVCNT-1;
   CURNODE.PTRS(0)=1;
   CURNODE.KEYS(1)=KY;
   CURNODE.PTRS(1)=SS1;
   CURNODE.PTRS(2)=SS2;
   CALL PUTNODE(I,1);
   ROOT=I;
END HDNODE;
END BTINDX;
```

# APPENDIX F

## PL/I PROGRAM SHOWING USE OF GENERIC PROGRAM

The following section contains the PL/I program that performs a generic traversal of a B-tree index.

```
/********************/
/*                  */
/* GENTRAV ROUTINE */
/*                  */
/********************/
GENTRAV:  PROC(ROOT,PARKEY,CNT,MAXB,KLEN,MAXNODES,INFILE,
               INPROC,FLAG);
   DECLARE  /**  FOR PARAMETERS  **/
      (ROOT,CNT,MAXB,KLEN,MAXNODES) FIXED BIN(15,0),
      PARKEY CHAR(*),
      FLAG BIT(*) ALIGNED,
      INFILE FILE,
      INPROC ENTRY (FIXED BIN(15,0),CHAR(*));
   DECLARE  /**  INTERNAL VARIABLES  **/
      (I,J) FIXED BIN(15,0),
      (LENGTH,SUBSTR,VERIFY) BUILTIN;
   IF PARKEY=' ' THEN J=0;
      ELSE J=LENGTH(PARKEY);
   DO I=1 TO J;
      IF VERIFY(SUBSTR(PARKEY,I),' ')=0 THEN LEAVE;
   END;
   IF I<=J THEN J=I-1;
   IF J>KLEN THEN DO;
      FLAG='1'B;
      RETURN;
   END;
   FLAG='0'B;
   CNT=0;
   CALL TRAVINGENTRAV(ROOT);
```

```
/*************************/
/*                       */
/* TRAVINGENTRAV ROUTINE */
/*                       */
/*************************/
TRAVINGENTRAV: PROC(NODENO) RECURSIVE;
   DECLARE
      (NODENO,I) FIXED BIN(15,0),
      1 NODE,
         2 PTRS(0:MAXB+1) FIXED BIN(15,0),
         2 KEYS(MAXB) CHAR(KLEN);
   IF NODENO<0 | NODENO>MAXNODES THEN RETURN;
   READ FILE(INFILE) INTO(NODE) KEY(NODENO);
   IF PTRS(0)<1 | PTRS(0)>=MAXB THEN RETURN;
   DO I=1 TO PTRS(0)+1 UNTIL(SUBSTR(KEYS(I),1,J)>PARKEY);
      IF I=PTRS(0)+1 & PTRS(1)<0 THEN RETURN;
      IF SUBSTR(KEYS(I),1,J)>=PARKEY | I=PTRS(0)+1 | J=0
      THEN DO;
         IF PTRS(I)<0 THEN DO;
            IF SUBSTR(KEYS(I),1,J)>PARKEY THEN RETURN;
            CALL INPROC(-PTRS(I),KEYS(I));
            CNT=CNT+1;
         END;
         ELSE CALL TRAVINGENTRAV(PTRS(I));
      END;
   END;
END TRAVINGENTRAV;
END GENTRAV;
```

## APPENDIX G

## PL/I BATCH PROGRAM

The following section contains the batch routine that uses GENTRAV to do a full in-order traversal of the B-tree in ACTFILE. The procedures and structures that are included can be found in the programmer's guide written by the author located in the Deans's Office in the College of Arts and Sciences.

```
$FTE$: PROC OPTIONS(MAIN);
   DECLARE (CNT,MAXB,KLEN) FIXED BIN(15,0),
      ACTFILE DIRECT INPUT KEYED ENV(REGIONAL(1)),
      PAFILE  DIRECT INPUT KEYED ENV(REGIONAL(1)),
      FLAG BIT(1) ALIGNED,
      (SUBSTR,LOW,INDEX,FLOAT,FLOOR,MOD,MULTIPLY,ROUND)
      BUILTIN;
   %INCLUDE REC0;
   %INCLUDE PA;
   MAXB=120;
   KLEN=10;
   READ FILE(PAFILE) INTO(REC0) KEY('0');
   CALL GENTRAV(ACTROOT,' ',CNT,MAXB,KLEN,MAXNDE,ACTFILE,
      BLDFTE$,FLAG);
   CALL BLDFTE$(0,'          ');
   %INCLUDE BLDFTE$;
   %INCLUDE GENTRAV;
   %INCLUDE SERDAYNO;
   %INCLUDE AMTCOMM;
   %INCLUDE READPA;
END $FTE$;
```

## APPENDIX H

## PL/I REAL-TIME PROGRAM

The following contains the real-time counterpart to the program in Appendix G.

```
FTE$: PROC OPTIONS(MAIN);
   DECLARE (CNT,MAXB,KLEN) FIXED BIN(15,0),
      ACTFILE DIRECT INPUT KEYED ENV(REGIONAL(1)),
      PAFILE  DIRECT INPUT KEYED ENV(REGIONAL(1)),
      INPUTKEY CHAR(6),
      NOKEY CHAR(1),
      FLAG BIT(1) ALIGNED,
      (SUBSTR,LOW,INDEX,FLOAT,FLOOR,MOD,MULTIPLY,ROUND)
      BUILTIN;
   %INCLUDE REC0;
   %INCLUDE PA;
   ON ATTN STOP;
   MAXB=120;
   KLEN=10;
   READ FILE(PAFILE) INTO(REC0) KEY('0');
   PUT EDIT
      ('ENTER THE ACCOUNT NUMBER (NO EMBEDDED DASHES):')
      (COL(1),A);
   GET EDIT(INPUTKEY) (COL(1),A(6));
   PUT EDIT('POSITION THE PAPER AND HIT ''RETURN'':')
      (COL(1),A);
   GET EDIT(NOKEY)(COL(1),A(1));
   CALL GENTRAV(ACTROOT,INPUTKEY,CNT,MAXB,KLEN,MAXNDE,
      ACTFILE,BLDFTE$,FLAG);
   IF CNT>0 THEN CALL BLDFTE$(0,'          ');
   ELSE PUT EDIT('ACCOUNT ',INPUTKEY,' DOES NOT EXIST.',
      'CHECK ACCOUNT NUMBER AND ENTER IT AS A ',
      '6 DIGIT NUMBER (SUCH AS ''102201'')')
      ((2)(COL(1),(3)A));
   %INCLUDE BLDFTE$;
   %INCLUDE GENTRAV;
   %INCLUDE SERDAYNO;
   %INCLUDE AMTCOMM;
   %INCLUDE READPA;
END FTE$;
```

# APPENDIX I

## PL/I DESCRIPTION OF A PERSONNEL RECORD

A PL/I description of a personnel record is included in this appendix for the purpose of giving a little reference to what the file system has to work with. The key fields described in Chapter V can be seen in the substructure ID and in the substructure ACCOUNTS.

```
DECLARE
1 PA_RECORD,
   2 LINK_EXT              FIXED BIN(15,0),
   2 ID,
      3 NAME,
         4 LAST            CHAR(20),
         4 FIRST           CHAR(15),
         4 MI              CHAR(1),
      3 SSN                CHAR(9),
      3 RANK               CHAR(4),
      3 HOME               CHAR(6),
      3 PPN                CHAR(6),
      3 HIRED              CHAR(6),
      3 PROMOTED           CHAR(6),
      3 EXW@H              CHAR(1),
   2 NO_DISTR              FIXED BIN(15,0),
   2 NEXT_ACCT             FIXED BIN(15,0),
   2 DISTR(6),
      3 FROM               CHAR(5),
      3 THRU               CHAR(5),
      3 SALARY             FIXED DEC(7,2),
      3 FTE                FIXED DEC(7,4),
      3 BEGIN              FIXED BIN(15,0),
   2 ACCOUNTS(20),
      3 ACCT               CHAR(6),
      3 BLN                CHAR(6),
      3 PAY                FIXED DEC(7,2),
      3 COMM               FIXED DEC(7,2),
      3 LINK               FIXED BIN(15,0),
   2 FILLER                CHAR(137),
1 PA,
   2 LINK_EXT              FIXED BIN(15,0),
```

```
2 ID LIKE PA_RECORD.ID,
2 NO_DISTR                FIXED BIN(15,0),
2 NEXT_ACCT               FIXED BIN(15,0),
2 DISTR(12) LIKE PA_RECORD.DISTR,
2 ACCOUNTS(40) LIKE PA_RECORD.ACCOUNTS;
```

# VITA[2]

David Dale Christian

Candidate for the Degree of

Master of Science

Thesis: A B-TREE INDEX APPROACH TO STORING AND RETRIEVING RECORDS ON DIRECT ACCESS AUXILIARY STORAGE

Major Field: Computing and Information Sciences

Biographical:

Personal data: Born in Houston, Texas, on July 11, 1955. Moved to Wewoka, Oklahoma, in 1963 and married Cherry Choate in Seminole, Oklahoma, on May 14, 1976.

Education: Graduated from Wewoka High School, Wewoka, Oklahoma, in May, 1973; received Bachelor of University Studies from Oklahoma State University, Stillwater, Oklahoma, in May, 1977; completed requirements for Master of Science degree at Oklahoma State University, Stillwater, Oklahoma, in December, 1978.

Professional Experience: Graduate research assistant working as a programmer analyst for the Dean's Office in the College of Arts and Sciences, Summer of 1977 and January 1978-December 1978; graduate teaching assistant at Oklahoma State University, Computing and Information Sciences Department, Fall 1977; tutor in computer science for the Veteran's Administration, Summer 1977-Summer 1978.