B+-TREES

By

ROBERT EDWARD WEBSTER

Bachelor of Science

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
May, 1980

Name: Robert Edward Webster Date of Degree: May, 1980

Institution: Oklahoma State University

Location: Stillwater, Oklahoma

Title of Study: B+-TREES

Pages in Study: 129 Candidate for Degree of Master of Science

Major Field: Computing and Information Sciences

Scope and Method of Study: This paper describes a data structure called the B+-tree, developed by D. Comer and D. Knuth. The B+-tree is a modification of the B-tree. The storage characteristics of the structure are discussed, and empirical data is given from actual test cases generated from an implementation of the B+-tree designed for test purposes. Buffering of the B+-tree nodes is discussed, along with empirical results from two buffering methods. The design of an application of B+-trees in a relational database is presented.

Findings and Conclusions: The upper and lower bounds for storage utilization in a B+-tree were obtained analytically. An estimation of the average storage utilization was found empirically. Information was provided empirically and analytically on the effectiveness of the buffering of index nodes. Program listings and test results are included.

ADVISER'S APPROVAL_________________________________

B⁺-TREES

Report Approved:

Report Adviser

Dean of Graduate College

Preface

This report contains the description of B^+-trees and a partial analysis of their storage characteristics. Two buffering methods for B^+-tree nodes are presented. Empirical results given from algorithms tested on the computer. Programs were written in PL/I, compiled on the optimizing compiler, and run on the IBM 370/168.

I would like to thank my advisor, Dr. Phillips, and the other members of my committee, Dr. Chandler and Dr. Fisher, for their help on this report.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure | Page

CHAPTER I

STORAGE STRUCTURES

There are many techniques and structures used for the storage and access of data. Many of these allow "keyed" access to data records. Keyed access is the ability to access a data record by specifying a key that is associated with that record. The key may be unrelated to the physical storage location of the data record. There are two major classes of keyed access storage techniques: key to address transformations and search tree structures. Within each of these classes, there are techniques for accessing data on external as well as internal storage.

Key to address transformation (also called "hashing") is the mathematical transformation of the key into a physical storage address. Although the key may not be totally unrelated to the physical location of the data record, the selection of an appropriate function may make it seem that way. Key to address transformation has been used for the storage of data on both internal and external storage.

This method is typically faster than search tree structures, because much of the time the data record may be accessed with no intermediate accesses. However, it is necessary to obtain a key to address function which is appropriate for a given set of key values. The range and distribution of the values of keys may affect the efficiency of a key to address transformation system to a great degree. This makes it difficult to use such a system in a general database application in which the properties of keys are not known in advance.

A possible solution to this problem has been introduced by Fagin, (10), which consists of combining a radix tree, or trie (16), with key to address transformation. This technique, called extendible hashing, transforms the key into a pointer to a page with several keys. The range of key values within a page is dynamic. Because of this, the hashing function is not as tightly bound to the characteristics of the key values as it is in conventional key to address transformation systems.

A search tree is a tree structure in which the keys are arranged in such a manner that they can be accessed by key value. There are four major operations performed on search trees: searching, insertion, deletion, and traversal. Searching is the process of searching for a given key's position in the tree. Insertion is the process of inserting a key into the tree, and deletion the process of deleting a key from the tree. The traversal of a search tree involves traversing the tree, normally accessing keys in collating sequence.

Binary Search Trees

A binary search tree is a search tree in which each node contains one key and two pointers. The left pointer of each node points to a (possibly empty) subtree in which all keys are less than the key in the parent of the subtree. Similarly, all keys in the right subtree are greater than the key in the parent. The pointers in leaf nodes are null.

The search of a binary tree begins at the root node and proceeds to its descendants, visiting one node at each level. When a node is visited, if the desired key is less than the node's key, then the left pointer is followed. If the desired key is greater than the node's key, the right pointer is followed. The search terminates when the desired key is found or when a null pointer is encountered. When a key is inserted into a binary tree, a search is performed for the key. If the search is successful, the key cannot be inserted. If a null pointer is encountered, it is set to point to a new node that contains the new key and two null pointers.

Keys are always deleted from leaf nodes or semi-leaf nodes in a binary tree. A semi-leaf node is a node with only one descendant. If a key to be deleted has two descendants, then that key is exchanged with the next larger or next smaller key in the tree, which has at most one descendant. Then, the key and its node are deleted. If the deleted node had a descendant, the descendant is moved up into the space left by the deletion.

There are several ways to traverse binary search trees, the most common of which is the inorder traversal. This type of traversal accesses all keys in order. Other methods of traversal include preorder, postorder, and level order (16).

Binary search trees may or may not be well-balanced. A well-balanced binary search tree is one in which each node's two subtrees have approximately the same height. When a binary search tree is built by insertion of random keys, it is likely to be well-balanced. On the other hand, consider the case where the keys are inserted in ascending order. A degenerate tree is then formed in which every left pointer is null. This tree is essentially no more more than a linear linked list.

The average search time for a randomly built binary search tree is O(logN) where N is the number of keys in the tree (16). The average search time for a degenerate binary search tree is O(N). The average time for insertion corresponds very closely to the search time, since there is a constant time after the correct null pointer is found. The average time to delete a key from a binary search tree is O(logN).

Height Balanced Trees

Unconstrained binary search trees have good characteristics when they are well-balanced, but the fact that they may be degenerate can cause problems. Height balanced trees have provisions, or constraints, for keeping the tree well-balanced (13, 16). A height balanced tree has some value, k, which is the maximum difference in heights of a node's two subtrees. When an insertion or a deletion causes the heights of a node's two subtrees to differ by more than k, an adjustment is made to rebalance the subtrees. Height balanced trees are often named by their value of k. For example, if a tree had a value of one for k, it would be called an HB(1) tree.

The adjustments made to the tree to keep it balanced are called rotations. There are only two types of rotations used in rebalancing. Each of these requires a fixed amount of time. This means that the average time for insertion and deletion remains O(logN), while the average search time decreases (if k < N).

There have been several variations of HB(k) trees developed. A partially height balanced tree (HB(k1,k2)) has two values for k: one for the bottom level of the tree and one for the upper levels (12). Weight balanced trees use the number of nodes contained in, rather than the height of, the subtrees to test for rebalancing (16).

Height balanced trees are appropriate primarily for the internal storage of data. Other methods are generally used for storing data on secondary storage. By storing more than one key in a node, the number of accesses to secondary storage can be significantly reduced.

B-trees

The B-tree was first developed by Bayer and McCreight (3) in 1972. Since then, B-trees and variations thereof have become common data structures for the storage of information on secondary storage devices.

A B-tree is a search tree that has the following properties:

1. Each path from the root to any leaf has the same length, h, also called the height of the tree.
2. Each node has at most m descendants.
3. Each node, except the root and the leaves, has at least CEIL(m/2) descendants. The root is a leaf node or has at least two descendants.
4. Each node holds between FLOOR((m-1)/2) and m-1 keys, except the root which holds between 1 and m-1 keys.
5. Each non-leaf node with k keys has k+1 descendants.

Since the path from the root to any leaf has the same length, every leaf node must reside on the same level. For this reason, the B-tree is said to have uniform height.

In B-trees, insertions occur only at the leaf node level. The leaf node level is also referred to as the bottom level. An insertion may cause a node to become overfull, that is, to contain more than m-1 keys. If this happens, the node may be split into two nodes, and the middle key of the overfull node inserted into the parent node. This operation is called node splitting. An alternative method of handling overfull nodes is overflow sharing. In this operation, some of the keys and pointers from the overfull node are moved into one of its siblings. Overflow sharing is not possible if both of the overfull node's siblings contain the maximum possible number of keys. If overflow sharing is used, when possible, it tends to keep more keys in each node. This causes the tree to be shallower and have better search characteristics.

When a key is deleted from a B-tree, it is deleted from a leaf node. If the key to be deleted is in an upper level node, it is first swapped with the next larger or next smaller key in the tree, which always appears on the bottom level. A deletion may cause a node to become underfull, that is, to contain less than FLOOR((m-1)/2) keys. If this happens, the underfull node may be merged with a sibling that contains FLOOR((m-1)/2) keys. This operation is called node merging. When a node becomes underfull and a merge is not possible, an underflow share is performed. Underflow sharing consists of moving some of the keys and pointers from a sibling into the underfull node.

B-tree Variants

Several variations of the B-tree have come about in recent years. There seems to be a lack of uniformity in the terminology used in the definition of these structures. The definition of some of the terms used in this paper follow.

The leaf nodes of a B-tree have no descendants in the tree, but do contain keys and pointers to external nodes. External nodes may be imaginary nodes without information, or data records associated with keys in the leaf nodes. The bottom level of the tree refers to the leaf node level, or the level of the tree at which all leaves are present. The upper levels of the tree are any levels other than the bottom level. Comer (6) and Wagner (21) use "sequence set" to refer to the bottom level, and "index set" to refer to the upper levels of certain B-tree variations. In discussing B-trees, Knuth (16) uses "leaf node" to refer to the external nodes defined above, but changes the definition to agree with the above when discussing modifications used in the B+-tree.

In a conventional B-tree, data stored with the key may be large enough to occupy a considerable portion of an index node. If a large amount of data is stored with the keys in the nodes, the order of the B-tree may be relatively small, and so the height relatively large. Also, in a B-tree, all pointers on the bottom level are not used. Since most of the pointers in the tree are on the bottom level, most pointers in the tree are not used. A solution to these problems is to store each key and associated data on the bottom level of the tree (16). When a leaf node splits, the middle key is duplicated and propagated to the next higher level. The original key and data remain on the bottom level. The upper level keys and pointers are merely a "roadmap" to the bottom level. A search in this structure is not complete until the bottom level is reached. If a key that has a duplicate in an upper level is deleted from a leaf node, the upper level key does not need to be deleted. It can still function to guide searches to the bottom of the tree.

Instead of storing data with each key on the bottom level of the tree, as suggested above, a pointer to an associated data record can be stored. The permits the same structure to be used for both leaf nodes and upper level nodes. However, if the same structure is used, one pointer on each leaf node is not used. These extra pointers can be used to link all the bottom level nodes horizontally to aid in the traversal of the tree (8, 16, 21).

In the trees just described, the upper level keys are used only to guide searches. These keys can be compressed, using any of several techniques, to allow a greater branching factor on upper levels (4, 14, 21). Key compression results in keys with variable lengths. Because of this, the number of bytes used in a node, rather than the number of keys a node contains, is used to determine underflow and overflow conditions in upper level nodes.

CHAPTER II

THE B^+-TREE INDEX

The structure presented in this chapter is the B^+-tree[1], described by Comer (8) and Knuth (16). A description of the B^+-tree is given, followed by a partial analysis of the storage characteristics of the B^+-tree. Empirical results are presented, showing the convergence of the density of the B^+-tree after alternate insertions and deletions of random keys.

Each node in the B^+-tree contains only keys and pointers. The bottom level pointers point to data records, or external nodes. On the bottom level, there is one pointer per key. Each leaf node has a link to the next leaf node to the right, except the rightmost node, whose link is null.

Each upper level key is copied from a bottom level key during a node split on insertion. From that time on, the upper level key is used only to direct searches to the bottom level. A successful search in a B^+-tree is detected only when a matching key is found at the bottom level.

Some implementations of the B^+-tree (16) have data stored with the keys in leaf nodes, making the structure of

----

[1] B^+-tree is read "B plus tree."

leaf nodes different from the structure of the upper level nodes. The B^+-tree structure discussed in this paper is one in which the same structure is used for both leaf nodes and upper level nodes. The B^+-tree has one key per pointer in the bottom level and one more pointer than keys in the upper levels. Since one more pointer per node is used on the upper levels than on the bottom level, there is an extra pointer on each leaf node. The unused pointer on each leaf node is used as a horizontal link to the "next" leaf node.

The horizontal links across the bottom level can be maintained without much difficulty. The only time a horizontal link is updated is during an node split or merge. In either case, no additional node accesses are required beyond those ordinarily required for a split or merge.

The horizontal links allow the "next" key and pointer to be accessed without using upper level nodes, after the initial search. This makes it possible to traverse several trees simultaneously, keeping only one node per tree in memory at a time.

In the B^+-tree, only the keys in leaf nodes are associated with data records. The upper level keys are duplicated from keys in the bottom level keys, and are only used to reference other nodes. The duplication of keys on the upper levels may cause the number of keys per node to be misleading. A B^+-tree with N external nodes has N keys in the bottom level of the tree, but the number of keys in upper levels may vary. This variance may be small, but a unit of measurement can be chosen which will show the overall storage characteristics of the tree more accurately than keys per node. Instead of using the total number of keys per node, the number of external nodes per internal node can be used, that is, the number of keys on the bottom level divided by the number of nodes in the tree. This unit will be called "effective keys per node." The number of effective keys per node in a tree of order m with N external nodes is represented as E(m,N).

Storage Characteristics

Best and Worst Cases

To find the upper and lower bounds on E(m,N) for a B^+-tree of order m with N external nodes, it is necessary to determine the maximum and minimum nodes in the tree. N can then be divided by these values to obtain the maximum and minimum value for E(m,N).

The minimum number of pointers in a node is

$$d = \mathrm{CEIL}(m/2).$$

The maximum number of leaf nodes is

$$\mathrm{FLOOR}(N/(d-1)).$$

If there are n nodes on a level, L, of a tree, and if n>1, then there are n pointers on level L-1, the level immediately above. This can be seen intuitively since each node except the root must have a pointer to it from the next higher level. The maximum number of nodes on an upper level with n pointers is

FLOOR(n/d).

Using the above, one can progress from the bottom level of the tree upward, counting the number of nodes on each level, until the number of nodes on a level is one. When the root node is reached (where the number of nodes on the level is one), the maximum possible number of nodes in a B⁺-tree for the given order, m, and size, N, is obtained.

The minimum possible number of nodes may be found similarly. The minimum number of leaf nodes is

CEIL(N/(m-1)).

The minimum number of nodes on an upper level with p pointers is

CEIL(p/m).

On an upper level, each node has one more pointer than key. Therefore, on an upper level with n nodes and p pointers, there are (p-n) keys. Using this, the maximum or minimum number of keys in the tree may be counted along with the nodes.

In each progression upward during the counting, the maximum or minimum number of levels in the tree may be tallied. The algorithm in Figure 1 may be used to find the maximum and minimum number of keys, nodes, and levels in a B⁺-tree of order M with N external nodes.

The functions for the maximum and minimum number of nodes and keys would be linear if the FLOOR and CEIL functions were not present, since J is divided by the same value each time through the loop. This implies that a linear approximation to the maximum and minimum number of nodes in a B⁺-tree can be obtained.

```
STAT: PROC (MAXNODES, MAXKEYS, MINNODES, MINKEYS,
  MAXLEV, MINLEV, M, N);

  MAXNODES,MAXKEYS,MAXLEV,MINNODES,MINKEYS,MINLEV = 0

  D = CEIL(m/2);
  /*  FIND MAXIMUMS  */
  J = FLOOR(N/(D-1))+N;
  DO WHILE J > 1;
    I = FLOOR(J/D);
    MAXNODES = MAXNODES+I;
    MAXKEYS = MAXKEYS+J-I;
    MAXLEV = MAXLEV+1;
    J = I;
    END;

  /*  FIND MINIMUMS  */
  J = CEIL(N/(M-1))+N;
  DO WHILE J > 1;
    I = CEIL(J/M);
    MINNODES = MINNODES+I;
    MINKEYS = MINKEYS+J-I;
    MINLEV = MINLEV+1;
    J = I;
    END;
  END STAT;
```

Figure 1. Algorithm to Find the Maximum and Minimum Keys, Nodes, and Levels in a B⁺-tree

If there are n nodes in a level of the tree, then there are n-1 keys in all levels above that level. This is shown by the following:

1. If a level has only one node, then it is the root node and there are no keys in the above levels.

2. A node is added to a level if and only if a key is added to the upper levels in the process of node splitting.

3. A node is deleted from a level if and only if a key is deleted from the upper levels in the process of node merging.

This implies that the maximum and minimum number of keys can be found using

N+FLOOR(N/(d-1))-1

for the maximum, and

N+FLOOR(N/(m-1))-1

for the minimum.

The linear approximation for the maximum and minimum number of nodes in a B^+-tree can also be found. As stated previously, the maximum number of leaf nodes is

FLOOR(N/(d-1)).

The maximum number of keys on upper levels of the tree is

FLOOR(N/(d-1)-1).

The maximum number of nodes in a B-tree with k keys is approximated by FLOOR(k/(d-1)), so the maximum number of nodes in the upper levels of a B^+-tree is approximately

FLOOR((FLOOR(N/(d-1))-1)/(d-1)).

The maximum number of nodes in the entire tree can be approximated by adding FLOOR(N/(d-1)) to the above, giving

FLOOR((n-1)/(d-1))+n

where

$$n = FLOOR(N/(d-1)).$$

The minimum effective keys per node may be estimated by dividing N by the above.

It can be shown that the linear approximation for the maximum number of nodes in a B+-tree has a maximum error of L-1, where L is the number of levels in the tree. There are two places where error is introduced. One stems from the fact that the root node may be less than 1/2 full. This tends to make the approximation less than the actual maximum. The other source of error is the fact that on each upper level, it may not be possible to have all nodes at minimum capacity. This tends to make the approximation greater than the actual maximum. The maximum error for this is one node for each level. The top level has two possible errors of one, but since the they are opposing, an error of only one may occur at this level.

An approximation of L, the number of levels in a B+-tree, is given by

$$L <= 1 + \log_d (N + FLOOR(N/(d-1))-1).$$

Therefore, the maximum error in the linear approximation of the maximum number of nodes in a B+-tree is

$$e <= \log_d (N + FLOOR(n/(d-1))-1).$$

A similar derivation for the minimum number of nodes in a B+-tree can be done, yielding

$$FLOOR((n-1)/(m-1))+n$$

where

$$n = FLOOR(N/(m-1))$$

for the minimum. The maximum error turns out to be

$$e \le \log_m (N + FLOOR(n/(m-1))-1).$$

Similarly, the number of nodes in a B*-tree with p keys per node may be approximated by

$$(N/p-1)/p+N/p.$$

<u>Average Storage Characteristics</u>

A B*-tree can be built by inserting random keys to obtain the storage characteristics of the tree. If such a tree is built, its density, or number of keys per node with respect to node capacity, is higher than that of a tree of identical size that has undergone a series of alternate insertions and deletions. By the same token, after a tree has undergone a series of deletions it tends to be more sparse than normal. As a newly built tree undergoes alternate insertions and deletions of random keys, its density converges to a value which will be called the average density. The average storage characteristics of a B*-tree are those of a tree that has undergone an infinite number of alternate insertions and deletions of random keys.

To obtain the average storage characteristics of a B*-tree, its density must be adjusted after the tree is built initially. This can be done by performing alternate insertions and deletions on the tree until the density nears the average. The density can also be adjusted by inserting more than the desired number of keys into the tree, and then deleting the difference.

A tree with a relatively small order, m, tends to approach the average density faster and smoother than a tree with a larger order. Consider a tree of order 201 with 13 nodes and 2000 keys on the bottom level. Consider further a leaf node at 90% capacity, containing 180 keys. In order for the node to share or split, there must be 21 more insertions into that node than deletions from it.

Consider a second case in which a tree of order 11 has 2000 keys on the bottom level and 27 leaf nodes. A node at 90% capacity on the bottom level contains 9 keys. Only 2 more insertions than deletions must occur in this node for a split or overflow share to take place. A split or share operation is much more likely to occur in this tree than in the tree of order 201. This means that after a given number of alternate insertions and deletions, the tree of order 11 will probably have a density closer to the average than that of the tree of order 201.

The two trees have different average densities. The average density of the tree of order 201 is slightly greater than that of the tree of order 11. Empirical results show that the average density of B+-trees increases at a decreasing rate as the order increases. Furthermore, if the number of alternate insertions and deletions is normalized to the node size, B+-trees of lower orders still converge to the average density faster than B+-trees of larger orders.

Trees of relatively large orders have nearly the same average density.

Any differences between the average storage characteristics in a B+-tree and a B-tree would be caused by the difference in the way insertion splits and deletion merges are done on the bottom level. When a leaf node splits in a B-tree, a key is removed from the bottom level and promoted to the next higher level. In a B+-tree, a leaf node split causes a key to be duplicated and promoted to the upper level, so one of the leaf nodes has one more key than it would in a B-tree. In a B-tree, when a node merge occurs, the key in the parent node that separates the two nodes being merged is moved into the middle position of the node resulting from the merge. When a merge occurs on the bottom level of a B+-tree, that key is merely deleted, since it does not refer directly to a data record. This leaves the merged node with one less key than it would have in a B-tree.

Most of the nodes of a B+-tree are at the bottom level. In the following discussion, it will be assumed that the upper levels of the tree reflect the characteristics of the bottom level. The root, with a different minimum number of keys than the other nodes, will be ignored. The difference in the densities of the bottom level of a B+-tree and the upper levels is expected to be negligable, especially since the upper levels usually contain a small percentage of nodes in the tree.

Empirical Results

Empirical data was gathered for B*-trees of several orders. The trees approach average densities of .76 to .80. The density of trees newly built averaged from .84 to .86, for trees of order 7, 11, 12, 13, 15, 24, 35, and 49. After each tree was built, a series of alternate insertions and deletions of random keys was done, recording the storage characteristics periodically. Next, a series of insertions, a series of deletions, and another series of alternate insertions and deletions were done. The results provide data for the approach of an underfull tree to the average.

The data from Table I was obtained by the method described above. The "Operations" column refers to the number of alternate insertions and deletions. One operation is defined as an insertion and a deletion pair.

Figures 2 and 3 illustrate the way relatively sparse and dense B*-trees approach average density. The tree of order 35 approaches the average density much slower than the tree of order 13, as expected. Empirical data for these trees, as well as trees of other orders, is given in appendix A.

The expected number of nodes in a B*-tree of order m with N keys and an average density of .78 is

$$(N/p-1)/p+N/p$$

where

$$p = .78(m-1).$$

N can be divided by this to obtain the expected effective keys per node:

$$E(m,N) = p^2 / (p - p/N + 1).$$

```
+--------+------------------------------------------------------------+
|Density |                                                            |
+--------+------------------------------------------------------------+
| 1.00-- |                                                            |
|        |                                                            |
|  .95-- |                                                            |
|        |                                                            |
|  .90-- |                                                            |
|        |                                                            |
|  .85-- |                                                            |
|        |                                                            |
|  .80-- |                                                            |
|        |                                                            |
|  .75-- |                                                            |
|        |                                                            |
|  .70-- |                                                            |
|        |                                                            |
|  .65-- |                                                            |
|        |                                                            |
|  .60-- |                                                            |
|        |                                                            |
|  .55-- |                                                            |
|        |                                                            |
|  .50-- |                                                            |
+--------+------------------------------------------------------------+
|   N:   |  1   2   3   4   5   6   7   8   9 10 11 12 13 14 15 16    |
+--------+------------------------------------------------------------+
```

Figure 2. Density of the Bottom Level of an Order 13 B+-tree After N*100 Operations

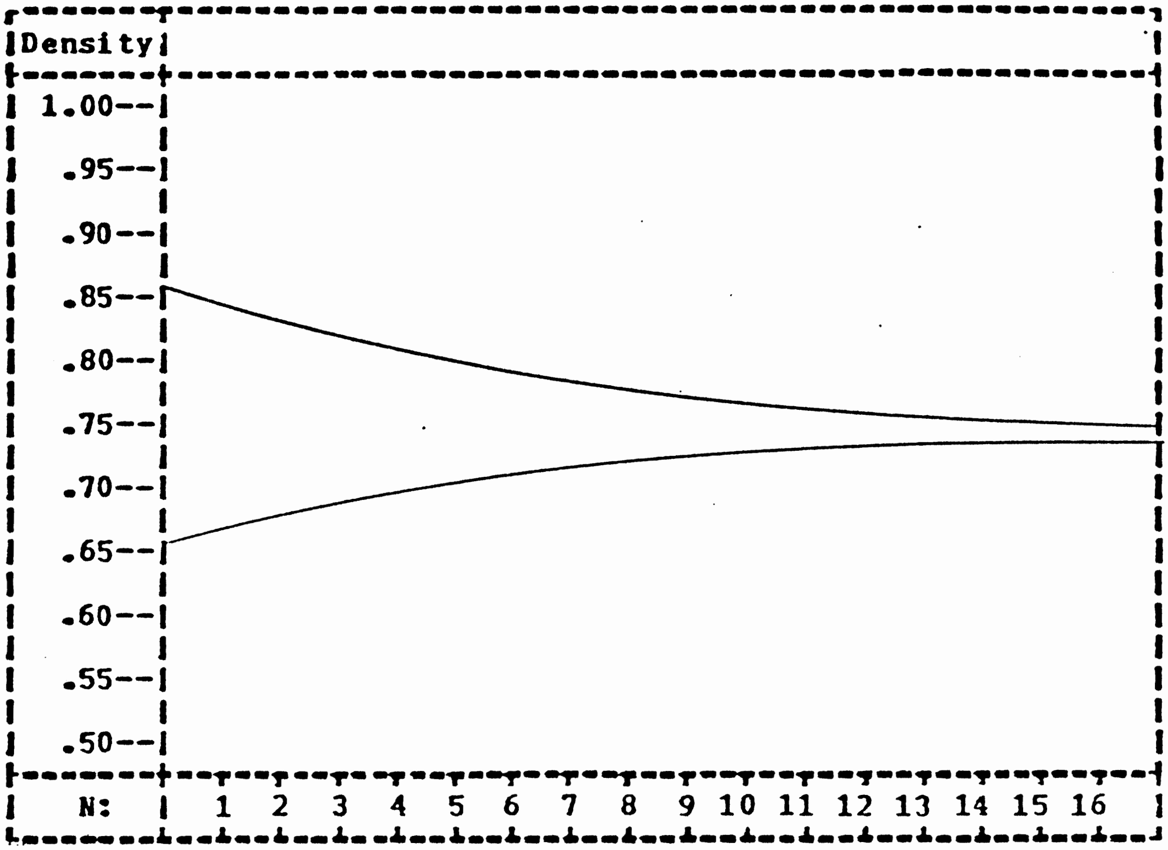


Figure 2. Density of the Bottom Level of an Order 13
B*-tree After N*100 Operations

TABLE I

STORAGE CHARACTERISTICS OF THE BOTTOM LEVEL OF A B+-TREE

Number of Operations	Number of Nodes	Density
0	199	.8375
10	199	.8375
20	201	.8292
30	202	.8251
40	204	.8170
50	204	.8170
60	203	.8210
70	203	.8210
80	203	.8210
90	203	.8210
100	203	.8210
120	204	.8170
140	204	.8170
160	207	.8052
180	209	.7974
200	210	.7937
220	210	.7937
240	210	.7937
260	211	.7899
280	211	.7899
300	211	.7899
350	212	.7862
400	214	.7788
450	212	.7862
500	214	.7788
550	215	.7752
600	213	.7825
650	214	.7788
700	213	.7825
750	216	.7716
800	216	.7716
900	216	.7716
1000	218	.7645
1100	217	.7680
1200	216	.7716
1300	215	.7752

Order = 13, Number of Keys = 2000

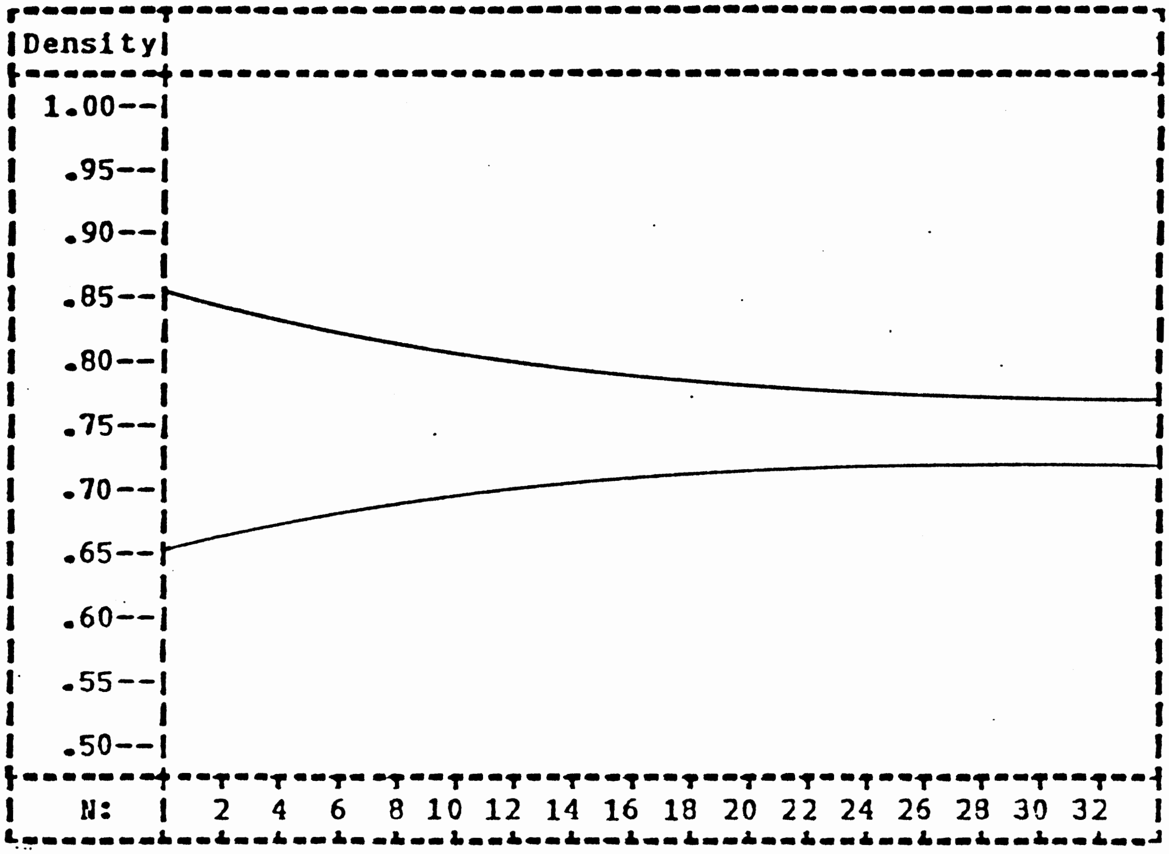


Figure 3. Density of the Bottom Level of an Order 35 B^+-tree After N*100 Operations

CHAPTER III

INDEX NODE BUFFERING

It is inefficient to access secondary storage for the root node each time the tree is used. It is preferable to keep the root node in memory until the program has completed its operations, and then output the root node to secondary storage. For some B⁺-trees, it may also be feasible to keep more than the root node in memory, especially if the root node has only a few descendants.

Least Recently Used Replacement Method

The first buffering method presented is the "least recently used replacement" method (18). Using this technique, the K most recently used nodes remain in memory. K pointers are set to point to the nodes in the buffer. The first pointer refers to the most recently used node and the Kth pointer to the least recently used. If the node referred to by the third pointer is requested as input, the first pointer is set to point to that node, the third pointer to the node previously referred to by the second, and the second pointer to the node previously referred to by the first pointer. This is illustrated in Figure 4. By adjusting the pointers in this fashion each time a node in the buffer is accessed, the relative time since the last access is retained for each node in the buffer.

Pointer	Page
3	1
1	2
2	3

Before Access to Page 2

Pointer	Page
2	1
3	2
1	3

After Access to Page 2

Figure 4. Pointers to Buffered Pages for Maintenance of Time Since Last Reference

If a node that is not in the buffer is requested as input, the node replaces the least recently used node in the buffer. The pointers to the nodes are then adjusted to retain the relative times since last reference. If the replaced node has been updated in memory, it must be output to secondary storage before its replacement. A flag for each node in the buffer is used to tell whether the node has been altered.

If a node in the buffer is requested as output, it is replaced with the structure that would normally be output to secondary storage. The node's flag is set to indicate that the node has been altered, and the pointers to the nodes are adjusted to reflect the access to the node. If the node requested as output is not in the buffer, then the least recently used node is replaced, after outputting it to secondary storage if necessary. The new node's flag is then set to indicate alteration of the node, and the node pointers are adjusted.

At the end of the program, any nodes with their alteration flags set must be output to secondary storage.

Analytical Performance

If the number of nodes in the buffer is greater than or equal to the height of the tree, then the root node will remain in memory during a series of searches. During updates that do not require any node shares, splits, or merges, the root will also remain in memory. Most of the insertions or deletions in a B^{+}-tree do not require shares, splits, or merges, so the root node will remain in memory during most updates to the tree. This reduces the number of accesses to secondary storage by at least one for each search, and by at least one for most insertions and deletions.

Consider a tree with a height equal to the number of nodes in the buffer. After each search, the nodes in memory will be the same nodes that were accessed in the search. Since the root node is always accessed in a search, it will always remain in memory. If the root node has two descendants, then there is a 50% chance of saving an access on the second level during each search. If the root has three descendants, an access will be saved one third of the time on the second level. This reasoning can be generalized for any number of nodes on all levels. If a series of searches is performed on the tree, then the average number of accesses saved by buffering the nodes is

$$\sum_{i=1}^{h} \frac{1}{n_i}$$

where h is the height of the tree and n is the number of nodes at each level. This is equal to

$$\sum_{i=1}^{h} \left(1 - \frac{n_i-1}{n_i}\right)$$

If the buffer size is p nodes and the height of the tree is h, then the average number of accesses saved is at least

$$\sum_{i=1}^{h} \left(1 - \left(\frac{n_i-1}{n_i}\right)^{p/h}\right)$$

This assumes that duplicate nodes may be present in memory, and levels are accessed in any order, both of which are false. Even so, the error in this approximation is relatively small for large trees.

Levels of the tree that have a large number of nodes will make little contribution to the savings in accesses to secondary storage. The top two levels are responsible for the major part of access reduction in trees of relatively large orders.

The discussion so far has been limited largely to random searching. As stated before, there is no change in the number of accesses required in insertions and deletions that do not cause shares, splits, or merges. However, if a node split, merge, or share is necessary, at least one sibling must be accessed. This increases the number of accesses for the operation, and causes another bottom level node to reside in memory. Since there are a lot of nodes on the bottom level, this may increase the number of accesses for the next operation by decreasing the number of upper level nodes in memory. When a split or a merge propagates activity up the tree, most of the parent nodes will already be in memory, since they were the last nodes accessed.

Empirical Performance

Empirical results were obtained for the number of accesses required for random searches of B*-trees. Tables II and III contain data for buffer sizes of 1, 5, 10, and 20 nodes, for trees of order 12 and 24. The trees were built with N random keys. After the trees were built, N alternate insertions and deletions were performed to decrease the density. Next, searches were performed on all the elements of the tree. The average number of accesses per search is given. The empirical results correspond closely to the analytical estimation given above.

The empirical results also show that node buffering cuts the number of accesses required for B⁺-tree updates. For example, a tree of order 24 and size 2400 was built, and 2400 alternate insertions and deletions were performed. With a buffer of size one, the number of accesses required for the alternate insertions and deletions was 20,058. Using a ten node buffer, the number of accesses required was reduced to 12,271, nearly a 39% reduction. The number of nodes in the tree after the operations was 140. The decrease in accesses was brought about by keeping only about 7% of the nodes in the tree in memory.

TABLE II

AVERAGE NUMBER OF ACCESSES PER SEARCH
FOR A B⁺-TREE OF ORDER 12

Tree Height	Number of Keys	Accesses Per Search K=1	K=5	K=10	K=20
2	50	2.00	.36	-	-
2	100	2.00	.60	.28	-
3	300	3.00	1.51	1.07	.62
3	600	3.00	1.73	1.44	.97
4	1200	4.00	2.44	1.87	1.46
4	2400	-	-	2.13	1.72

K = Buffer Size, in Nodes

TABLE III

AVERAGE NUMBER OF ACCESSES PER SEARCH
FOR A B*-TREE OF ORDER 24

Tree Height	Number of Keys	Accesses Per Search K=1	K=5	K=10	K=20
2	50	2.00	-	-	-
2	100	2.00	.17	-	-
2	300	2.00	.81	.47	-
3	600	3.00	1.16	.78	.54
3	1200	3.00	1.41	1.00	.78
3	2400	3.00	1.71	1.42	.97
3	5000	-	-	1.68	1.36

K = Buffer Size, in Nodes

Height Weighted Method

It is usually more advantageous to keep upper level index nodes in memory rather than leaf nodes. The level of the tree in which a node resides can be used, as well the the time since the node's last reference, to determine the next node in the buffer to be replaced. This will cause a greater percentage of the buffered nodes to be from the upper levels of the tree, which in turn will cause fewer accesses to secondary storage.

To use this "height weighted" buffering method, each node in the buffer must have its height present, as well as the time since it was last referenced. Each node's time since last reference is maintained in the same manner as in the least recently used replacement method.

Each time, t, has a value between 1 and K inclusively, where K is the buffer size. Each height, h, has a value between 1 and L inclusively, where L is the height of the tree. The most recently used node has t=1. The root node has h=1. The formula used to assign a priority, P, to a node is

$$P = t + h * x$$

where x is a weighting factor for the height. The node with the greatest value for P is replaced next.

If x > K, then t will be ignored except for nodes on the same level. Similarly, if x < 1/L, then h will not be used. A value for the weighting factor, x, should be determined so that

$$1/L <= x <= K.$$

If x = K is used, node replacement will depend almost entirely on the level of the tree in which each node resides. This causes the maximum possible number of upper level nodes to reside in the buffer. During a share, split, or merge on the bottom level, however, each time a node is read from or written to the buffer, an access to secondary storage will probably be necessary.

Empirical Results

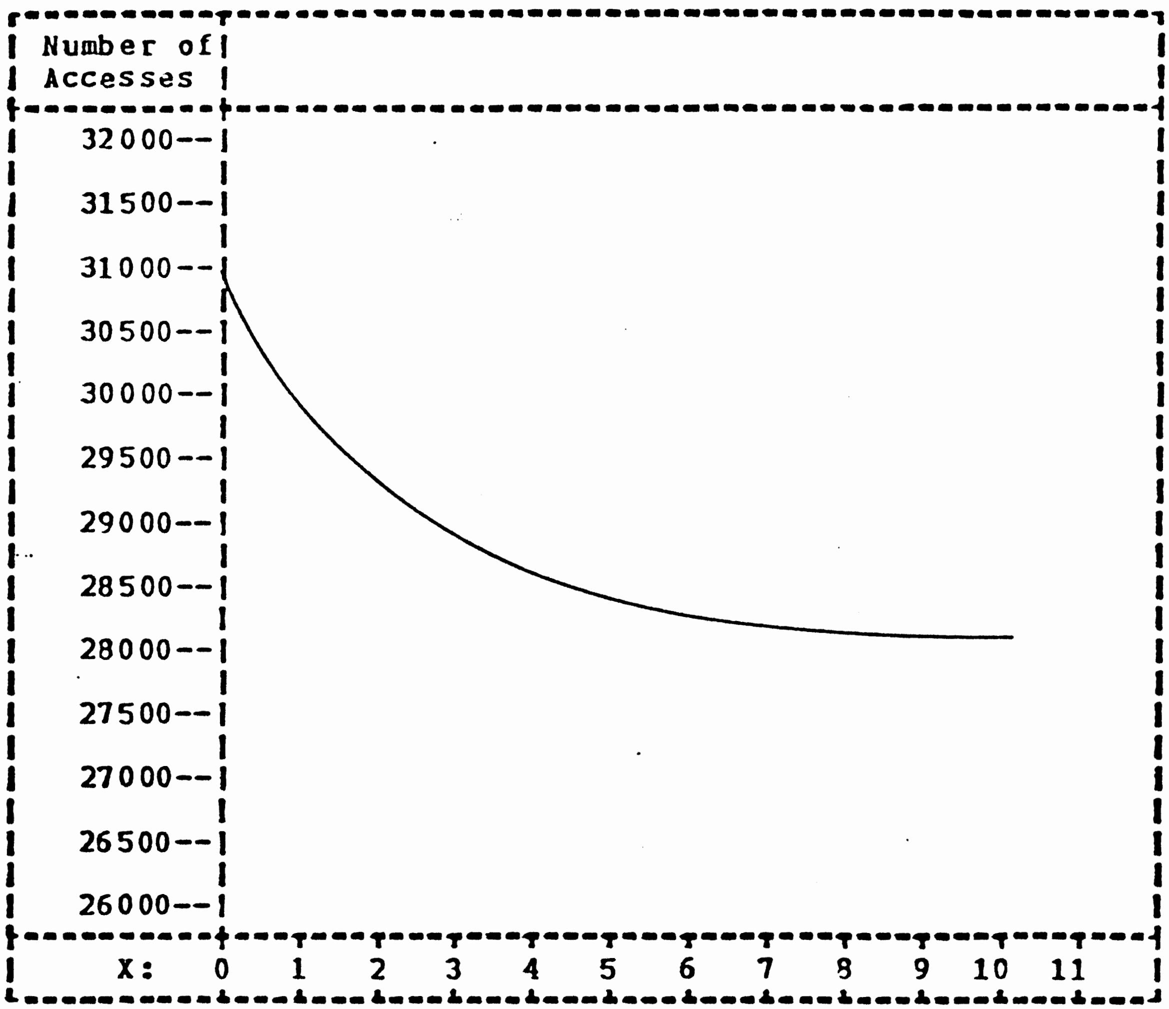
Figure 5 shows the results of testing different values for x on a B-tree of order 24. The buffer size used was ten nodes. The graph shows the number of accesses required for a sequence of insertions, alternate insertions and deletions, and searches on the B^+-tree. The actual operations performed can be found in appendix B. The best value for x seems to be between 8 and 10, for this case.

Considerations

The insertion of a key into a node requires a time of O(N), where N is the number of keys in the node. If there are several hundred keys per node, this time may be significant. By reducing the node size and increasing the buffer size, the total time required for updates might be reduced.

In a multi-user environment, the effect of node buffering changes. Because several different trees are likely to be used concurrently, the number of nodes in the buffer for each tree is reduced, which in turn reduces the access savings. The use of multiple buffers would probably be unfeasible for updates, because of problems with the duplication of nodes. A common buffer, with a "lockout mechanism" could be used, instead.

The efficiency of traversing a B^+-tree using horizontal links is not affected much by node buffering, since many nodes are accessed only one time. However, node buffering may be useful in B^+-tree traversals in a multi-user environment, since a node can be held in the buffer for each of several traversals. The best height weighting factor for buffering in such a system would probably not be the same as it would for a system with only one B^+-tree.



X = Height Weighting Factor

Figure 5. Number of Accesses vs. Height Weighting Factor for Operations on a B^+-tree of Order 24 with a 10 Node Buffer

CHAPTER IV

A STORAGE AND ACCESS SYSTEM DESIGN FOR A RELATIONAL DATABASE

A comprehensive relational database has been defined to have the following characteristics:

1. An interface for a high level, nonprocedural data language which provides the following capabilities for both application programmers and nontechnical users: query, data manipulation, data definition, and data control facilities.

2. Efficient file structures in which to store the database and efficient access paths to the stored database.

3. An efficient optimizer to help meet the response-time requirements of terminal users.

4. User views and snapshots of the stored database.

5. Integrity control - validation of semantic constraints on the database during data manipulation, and rejection of offending data manipulation statements.

6. Concurrency control - synchronization of simultaneous updates to a shared database by multiple users.

7. Selective access control - authorization of access privileges of one user's database to others.

8. Recovery from both soft and hard crashes.

9. A report generator for a highly stylized display of the results of interactions against the database and such application-

> oriented computational facilities as statistical analysis (15, p. 185-186).

A system design is presented here to supply the second item above for a relational database. The design of the file system is based on the B^+-tree, described in Chapter II.

Relational Database Structure

A relation is a set of n-tuples, or tuples. A relation may be thought of as a logical file, and a tuple as a logical record within that file. A tuple is a character string with one or more fields, or attributes. See Figure 6.

```
+--------------+--------------+-----+---------------+
| attribute 1  | attribute 2  | ... |  attribute N  |
+--------------+--------------+-----+---------------+
                       TUPLE


               +-------------------+
               |      tuple 1      |
               +-------------------+
               |      tuple 2      |
               +-------------------+
               |         .         |
               |         .         |
               |         .         |
               +-------------------+
               |      tuple N      |
               +-------------------+
                     RELATION
```

Figure 6. Structure of Tuples and Relations

A relational database contains a set of relations, on which operations such as joins, projections, and selections may be performed.

The storage and access system of a relational database system should provide the following capabilities:

1. Relation definition.
2. Access path definition.
3. Tuple addition, deletion, and update.
4. Tuple access.
5. Access path deletion.
6. Relation deletion.

A base relation is a relation that is not defined on any other relations. The base relation is the primary entity to be stored by a relational database storage and access system.

The definition of a relation involves the definition of the tuples and attributes of the tuples, such as the tuple length and the position and length of the attributes. Each relation defined must have a name by which it is to be referenced. The information on a newly defined base relation is stored on a secondary storage structure, such as a catalog.

There are several possible access paths to a base relation. The most straightforward is sequential access. Another method of access is the use of a set of direct links from tuples in one base relation to tuples in another.

Still another access method involves the use of an index, such as the B^{+}-tree, in which the bottom level pointers reference tuples, either directly or indirectly. The access paths must be maintained during the addition, deletion, and updating of tuples.

A high level design of a storage and access system for a relational database follows.

The Storage and Access System

In this system, a base relation contains tuples which are stored on pages. A page is a physical record from a file used by all base relations. A base relation page, as illustrated in Figure 7, contains a status word, a set of tag bits, and a set of tuples.

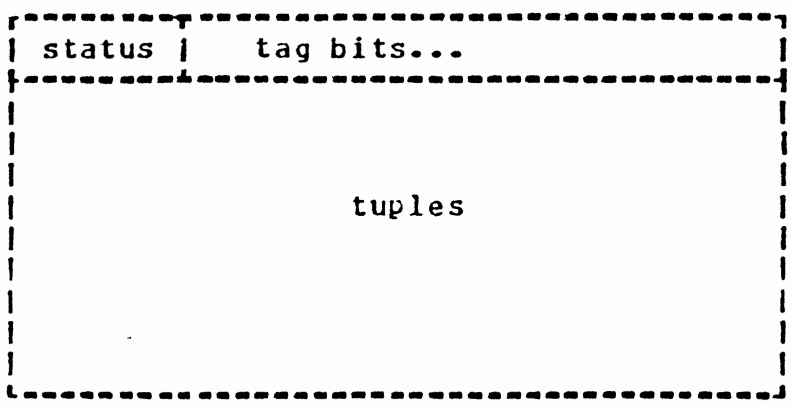


Figure 7. Base Relation Page

The status flag is set to -2 if the page is full, -1 if the page is partially full, and is a positive link in the list of available pages if the page is empty. Each tuple has a tag bit associated with it indicating whether the tuple is currently being used. Any full or partially full page is dedicated to a single relation. Any page on the available list is available to any relation.

When a page becomes empty, by the deletion of its last tuple, it is removed from the set of base relation pages and placed onto the available list. Similarly, if all the pages in a relation are full when a tuple is added, a page is taken from the available list and placed into the set of pages in the base relation.

The set of pages in a base relation are not necessarily contiguous. There must exist a method to access the pages of a relation sequentially, as well as find a partially full page, if there is one, for the addition of a new tuple. The capability must exist to access a page directly to update, delete, or read a given tuple. Also, there must be efficient means of adding pages to and deleting pages from a base relation. The solution to these problems is to use the B^+-tree for the management of pages for base relations.

There is a B^+-tree for each base relation with bottom level pointers referring to the pages of the base relation. The keys in the tree are the page numbers. Each base relation has the root of this page index stored in the catalog, along with other information. When a page is added to or deleted from the relation, the page number is added to or deleted from the index. When the relation is to be accessed sequentially, the page index is traversed.

The page index is typically only two or three levels in height. For example, if the index node size is 19,000 bytes, which is a common size for physical records on secondary storage devices, then over 2300 page numbers may be stored in each index node. This would make it highly unlikely for any page index to exceed three levels in height.

When a base relation is defined, information on attributes must be supplied. The length, name, and position of each attribute is required. Information on tuple indexes, if any, must also be present. Information on any binary links associated with the relation is supplied, as well as a possible clustering attribute, which will be described later. Each index has a root node, attribute name, and a flag that specifies whether the values of the attribute are to be unique. A base relation may be temporary or cataloged. A temporary base relation may be cataloged at a time other than when it is defined. The following information is stored in the catalog for base relations:

1. Base relation name.
2. Root node of page index.
3. Tuple length.
4. Number of attributes.
5. Attribute information.

   a. Attribute name.

   b. Position within the tuple.

   c. Length of the attribute.

6. Number of tuple indexes.

7. Tuple index information.

   a. Attribute name.

   b. Root node.

   c. Unique key flag.

8. Clustering attribute.

9. Number of sets of binary links.

10. Binary link information.

    a. Attribute name.

    b. Relation name.

    c. Root node.

The above information is kept in the catalog for all relations except temporary relations. The same information is stored in internal memory for temporary relations.

The deletion of a base relation involves deleting all the tuples and indexes, and removing the relation's entry, if any, from the catalog. The page index is traversed, placing each page of the relation onto the list of available pages. After the last access to each index node in the page index, the node is placed onto the available list for index nodes. The tuple indexes and binary link indexes are deleted in the same manner, except that no base relation pages need to be deleted.

Tuple Index

A tuple index is a B+-tree index in which all the values of an attribute in a relation are used as keys. Each tuple has one key. A tuple identifier is associated with each key input to the index. The tuple identifier is a fullword integer that contains the number of the page in which a tuple resides in the first halfword and the relative position of the tuple within the page in the second halfword. The tuple identifiers are the bottom level pointers of each tuple index.

Each key in a B+-tree must be unique, in order to allow deletions. It is sometimes necessary to have an index in a relational database in which some of the keys may be duplicated. Each tuple index has a flag associated with it that tells whether duplicate keys are allowed. If duplicate keys are allowed in an index, then the tuple identifier is concatenated with the original key to form a unique key, which is inserted into the tree. There are two types of searches in an index that allows duplicate keys. The first type is a search for the entire key, or a specific search. In this search, the original key and the concatenated tuple identifier are sought, and both must match for a successful search. This type of search is performed for the deletion of a tuple.

The second type of search is a generic search, or a search for only a portion of the key. Only the original key from the tuple is sought in this search. The tuple identifier is ignored. Since only a left hand portion of the key is considered, there may be more than one matching key in this type of search. The search is done by concatenating the lowest possible value in collating sequence to the primary key, in place of the tuple identifier. This results in the first match, if any, being obtained after a normal specific search for that key. From that point, the tree is traversed until the first key that does not match the primary key is found. The idea of a generic search may be generalized to allow a complete traversal of the tree by specifying a null primary key (7).

Each tuple index is maintained as the corresponding relation is updated. When an addition to the relation takes place, a key is inserted into the index. Similarly, the deletion of a tuple in the relation causes the deletion of that tuple's key from the index. A tuple update which changes the value of the attribute used by the index causes a deletion from and then an insertion into the index.

Clustering

When a relation is processed sequentially, using a page index, each page is read only once. When the same relation is processed inorder, using a tuple index, each page may be read several times - up to once per tuple. This can make processing relations inorder very inefficient, especially if the tuples are in random order with respect to the attribute the index is based on.

The physical order of the tuples can be maintained so that each page is read only once when the relation is processed in order of some attribute. A relation maintained in such a manner is said to be clustered on the attribute. A relation can be clustered on only one attribute. That attribute is called the clustering attribute.

A clustered relation has its page index modified to contain information on the largest attribute in each page, in addition to the information stated previously. Each key in the page index contains the maximum attribute value in the page, a flag indicating whether the page is full, and the page number.

The algorithm for insertion into a clustered relation is given in Figure 8. When a tuple is inserted into a clustered relation, the page index is searched for the first clustering attribute value greater than or equal to the attribute value in the tuple to be inserted. If the resulting page is not full, the tuple is inserted into that page. If the page is full, then it is split into two pages, each page containing half the tuples, so that each attribute value in one page is less than or equal to each attribute in the other. The tuple is then inserted into the appropriate page, and the page index is updated to contain an entry for the new page. When a page split takes place, any tuple indexes or binary links on the relation are changed to contain new tuple identifiers for all the tuples in one of the two pages, since the their old tuple identifiers would no longer be accurate.

```
INCLUSTER: PROC (TUPLE, ATTR_VALUE);
  SEARCH PAGE INDEX FOR ATTR_VALUE;
  IF PAGE IS NOT FULL THEN DO;
    INSERT TUPLE INTO PAGE;
    IF PAGE BECOMES FULL THEN UPDATE PAGE INDEX;
    END;
  ELSE DO;
    SORT TUPLES IN PAGE ON CLUSTERING ATTRIBUTE;
    PLACE THE UPPER 1/2 OF THE TUPLES INTO A NEW PAGE;
    UPDATE TUPLE IDENTIFIERS OF RELOCATED TUPLES IN
      BINARY LINKS AND TUPLE INDEXES;
    INSERT THE NEW TUPLE INTO THE APPROPRIATE PAGE;
    UPDATE THE PAGE INDEX;
    END;
  END INCLUSTER;
```

Figure 8. Algorithm for Insertion Into a Clustered Relation

Deletion in a clustered relation is fairly straightforward. A tuple is deleted from its page. Even if the tuple had the largest value for the clustering attribute in the page, the page index is not changed. The value in the page index can still be used to separate attribute values. If the tuple deletion leaves a page empty, then the entry for that page in the page index is deleted.

If a tuple update in a clustered relation results in a new value for the clustering attribute, the old tuple is deleted from and the new tuple inserted into the relation.

Splitting a page in a clustered relation can involve a considerable amount of overhead, since tuple indexes and binary links, as well as the page index, may have to be changed. Although a clustered relation is somewhat more costly to maintain than are other base relations, the benefits from inexpensive inorder processing may offset the high maintenance cost.

Binary Links

Binary links are sets of links that connect the tuples of two relations. Links go from each tuple in one relation to all the tuples in another relation that have the same value for some attribute. Similarly, links go from the tuples in the second relation to all tuples in the first relation with the attribute matching.

The binary links from one relation to another are stored in a B^+-tree. Each key consists of the "from" tuple identifier concatenated to the "to" tuple identifier. A generic search can be done on the binary link index to find all the tuples in a relation linked from a tuple in another relation. The search is done for the "from" tuple identifier.

The insertion of a tuple into a linked relation involves updating the binary link indexes going both to and from the relation. All matching tuples in the other relation are found. If a tuple index is available on the appropriate attribute, it is used. Otherwise a serial search of the relation is performed. As each matching tuple is found, an insertion is made into both binary link indexes.

When a tuple is deleted from a linked relation, every key in the two associated binary link indexes that contain that tuple identifier, in either the "from" tuple identifier or the "to" tuple identifier, must also be deleted. The maintenance of linked relations is moderately expensive, particularly when there is no tuple index on the linking attribute. However, binary links can play an important part in the implementation of views, and can be worthwhile in relatively stable relations.

Procedures

Figure 9 shows a block diagram of the procedures used in the storage and access system. There are three primary procedures: STORE, DEFINE, and ACCESS. High level pseudo-code, or program design language, descriptions are given for these procedures in Appendix D. STORE is used for tuple insertion, deletion, and updating in base relations. STORE also updates binary links and tuple indexes associated with the base relation being changed.

DEFINE is used for the definition of base relations and access paths. The following operations are supported by DEFINE:

1. Define a relation.
2. Define a tuple index.

3. Define binary links.

4. Delete binary links.

5. Delete a tuple index.

6. Delete a relation.

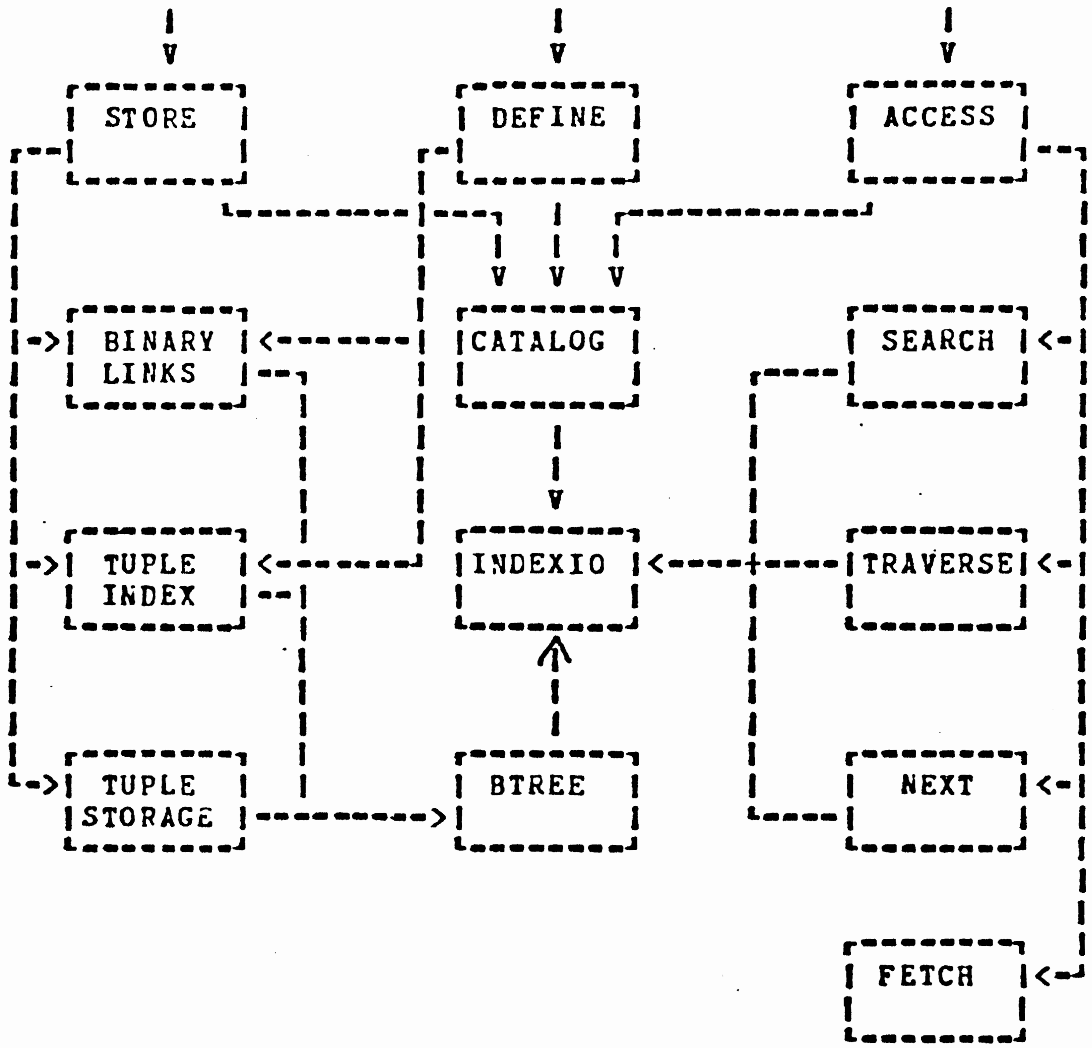


Figure 9. Block Diagram of Major Procedures

When one of the above items is defined, the information is placed into the catalog. If a tuple index or a set of binary links is defined on an existing relation, then the entire relation is processed, setting up the binary link index or tuple index as if each tuple encountered were a new tuple being inserted into the relation.

A set of binary links or a tuple index may be deleted without deleting the existing relation. When a relation is deleted, any binary links or tuple indexes associated with that relation are also deleted.

ACCESS provides for the access of tuples using any of the following operations:

1. Given a tuple identifier, get the tuple to which it refers.
2. Given a relational operator and a value of an attribute, get all the tuple identifiers of a relation whose tuples satisfy the restriction.
3. Given a tuple identifier of one relation, obtain tuple identifiers of another relation of tuples that match on a given attribute.
4. Using the page index for a relation, get the next tuple.
5. Using a tuple index for a relation, get the next tuple identifier.

In operation 2 above, a tuple index is used if one is available on the requested relation and attribute. Otherwise, a serial search is performed on the base relation. Similarly, binary links are used in operation 3 if the appropriate set exists. If not, and if one is available, a tuple index is used. If binary links and a tuple index both cannot be used, then a serial search of the base relation is performed.

For operations 4 and 5 above, a "cursor", containing the index node and offset within the node of the current key, is kept by the calling program. The cursor contains a null value on the first call. ACCESS "increments" the cursor to the next position in the node, or to the next node, each time the procedure is called. An end-of-file flag is set after the end of the relation has been encountered.

As shown in Figure 9, several supporting procedures are called by STORE, DEFINE, and ACCESS. TUPLE INDEX is called to insert, delete, or update a key in a tuple index. BINARY LINKS performs the same function for a binary link index. BTREE is a general purpose routine that provides for the insertion and deletion of keys in B+-tree indexes. INDEXIO is a buffered input/output procedure for index nodes.

CATALOG is a set of procedures which may be called to obtain or store information on relations and access paths. Certain information is kept on each relation and access path, as well as other items. At the beginning of the relational database's main program, the catalog information is read from secondary storage by a call to a catalog procedure. Similarly, at the end of the main database program, a call to a catalog procedure causes the catalog information to be written back out to secondary storage. At this time, the catalog procedure calls INDEXIO to write any index nodes remaining in memory out to secondary storage. Also, at the end of the program, the catalog procedure deletes any existing temporary relations. Catalog procedures are called by STORE and ACCESS to obtain tuple format and access paths for relations.

SEARCH is a procedure used to search an index for a given key. It is used with tuple indexes, binary link indexes, and page indexes for clustered relations. TRAVERSE is used to get the next tuple identifier, given the last "cursor", or index node and relative offset of the key within the index node. The cursor may be set to null to start with the first tuple identifier, or it may be set by a call to SEARCH, if the traversal is to begin somewhere other than the beginning. FETCH reads a tuple from the base relation page, given a tuple identifier. NEXT traverses the page index, much like TRAVERSE does the tuple index. It returns a tuple identifier to ACCESS, which, in turn, calls FETCH to retrieve the tuple itself.

It should be noted that the high level design of a complex system such as this should not be held completely static. If the reasons for a change outweigh the reasons not to change, then a change should be made. Some of the decisions in the design of this system were based upon expected properties of the calling program. As the calling routines are designed, it will probably be advantageous to modify this design to better suit them. The access routines are particularly susceptible to change, since they are directly dependent upon the needs of the calling program.

The storage and access system just described is meant for use with an intermediate processor for queries. The intermediate processor is to use this system to perform the storage and access of tuples. The intermediate processor that processes joins, view accesses, etc., will probably not be the same procedure that receives and analyzes source query statements. For further details on relational database systems, see (1, 15).

CHAPTER V

SUMMARY, CONCLUSIONS, AND SUGGESTIONS FOR FURTHER RESEARCH

Summary and Conclusions

The unit of storage utilization in a B^+-tree was defined to be the effective keys per node, or the number of bottom level keys divided by the number of nodes in the tree. An algorithm for determining the exact upper and lower bounds for storage utilization in B^+-trees was presented, along with a linear approximation of the bounds and an associated error limit.

An approximation for the average density of a B^+-tree was determined empirically to be between .76 and .80. An approximation of the average effective keys per node was derived from this. Empirical data showed that B^+-trees built from a series of insertions have higher densities than do B^+-trees that have undergone deletions as well as insertions. Densities of B^+-trees of larger orders decrease at slower rates than do B^+-trees of relatively small orders.

B^+-tree node buffering was tested using two methods. The least recently used replacement method picks the node that was used least recently to be replaced. The height weighted method uses the height of a node in the tree, as well as its time since last reference, to determine which node will be replaced. Empirical results showed that node buffering significantly reduces the number of accesses required for the searching and maintenance of B+-trees. Furthermore, the height weighted method proved to be more effective than the least recently used replacement method.

An application of B+-trees in a relational database was illustrated by the high level design of the storage and access system of a relational database. B+-tree indexes were used in the management of pages for relation storage, to order tuples of a relation by an attribute, and to store sets of many to many binary links between relations.

Suggestions for Further Research

The effect of three way splitting, keeping each node at least at 2/3 capacity instead of at 1/2, can be determined empirically for B+-trees. Also, sharing among a node and both its siblings, instead of just one, could be done to help reduce the amount of splitting and increase the density of the tree.

The average storage utilization of B+-trees has yet to be determined analytically.

Tests can be performed to determine empirically, as well as analytically, the concentration of B+-tree densities around the average. It was found that as B+-trees approach the average density under alternate insertions and deletions, they seem to exhibit a certain amount of hysteresis, which increases as the order increases. Hysteresis is resistance to change. For example, a B^+-tree of a high density may approach the average density under alternate insertions and stop short of the analytical average.

Several test cases of B^+-trees approaching their average densities can be run, and the resulting data fitted to a curve to aid in finding their characteristics analytically. The data from Appendix A was fitted to the function of a constant times an exponential. The data fit the curves fairly well, but the individual functions were not sufficiently similar to draw conclusions.

More empirical data can be obtained to find the effect of the two methods of node buffering on the number of accesses required for searching and updating B^+-trees. The amount of the effect could be given in terms of order, height, and buffer size. The analytical value for the above effect could also be determined.

Other variations of the two node buffering methods can be examined. For example, the number of keys in a node can be used in the weighting to determine replacement, along with the other parameters. Instead of using P=t+h*x for replacement determination, P=T+h**x could be used. This would correspond better to the number of nodes on each level. There could be two buffers used. The first buffer would be only 2 to 3 nodes in size, and would use the least recently used replacement method. The second buffer would be larger, and would use the height of and number of keys in each node to determine replacement. A decision would have to be made on where to put nodes that qualify to remain in both buffers.

After finding an optimal node buffering method, the characteristics of buffers of different sizes could be determined. It could possibly be more efficient, in terms of accesses to secondary storage, to use smaller nodes and larger buffers.

There is a vast amount of research to be done in the field of relational databases. Some suggestions for additional work on the storage and access system presented in this paper follow.

Index structures can be designed so that one index on a common attribute of two or more relations can refer to tuples in any of the relations. This method, and the method used in the present system, could be implemented and compared in terms of time and storage costs.

The implementation of view relations could be designed in such a way that binary links and predetermined projections make the access to a view relation very efficient. Allowing binary links between relations to be defined by a general join, rather than the equivalence of a single attribute, would facilitate this.

In a clustered relation, overfull pages split, much like index nodes. This analogy could be extended to keep pages at least at 1/2 capacity by using merging and underflow sharing in pages. This could be explored to determine whether the increased storage utilization would offset the overhead of changing tuple identifiers in binary links and tuple indexes, after each merge or share. The above concept could be carried one step farther, if it seemed worthwhile, to include overflow sharing and three way splitting, merging, and sharing.

Instead of separating the tuple pages and page index, the tuples themselves could be stored on the bottom level of the page index. The leaf nodes would have a different structure and order than that of the upper level nodes. Instead of using tuple identifiers, a unique attribute value for the tuple could be used for identification. Each relation would have a "key attribute", one in which no duplication of keys is allowed.

This would require a different node format for leaf nodes of tuple indexes. Binary links would contain key attribute values instead of tuple identifiers. The ability to cluster relations on any attribute would not be supported, since every relation would, in effect, be clustered on the key attribute.

BIBLIOGRAPHY

(1) Astrahan, M. M., et al. "System R: Relational Approach to Database Management." ACM Transactions on Database Systems, 1, 2 (June, 1976), 97-137.

(2) Bayer, R. "Symmetric Binary B-trees: Data Structure Maintenance Algorithms." Acta Informatica, 1 (1972), 290-306.

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

(4) Bayer, R. and M. Schkolnick. "Concurrency of Operations on B-trees." Acta Informatica, 9 (1977), 1-21.

(5) Bayer, R. and K. Unterauer. "Prefix B-trees." ACM Transactions on Database Systems, 2, 1 (March, 1977), 11-26.

(6) Blasgen, M. W. and K. P. Eswaran. "Storage and Access in Relational Data Bases." IBM Systems Journal, 16, 4 (1977), 363-377.

(7) Christian, D. D. "A B-tree Index Approach to Accessing Records on Direct Access Auxiliary Storage." (Unpub. M.S. thesis, Oklahoma State University, 1978.)

(8) Comer, D. "The Ubiquitous B-tree." Computing Surveys, 11, 2 (June, 1979), 21-138.

(9) Davis, W. S. "Empirical Behavior of B-trees." (Unpub. M.S. report, Oklahoma State University, 1974.)

(10) Fagin, R., J. Nievergelt, N. Pippenger, and H.R. Strong. "Extendible Hashing - A Fast Access Method for Dynamic Files." ACM Transactions on Database Systems, 4, 3 (September, 1979), 315-344.

(11) Held G., and M. Stonebreaker. "B-trees Re-examined." Communications of the ACM, 21, 2 (February, 1978), 139-143.

(12) Hernon, M. B. "The Design and Application of Research Tool for Height Balanced Trees." (Unpub. M.S. thesis, Oklahoma State University, 1979.)

(13) Horowitz, E. and S. Sahni. Fundamentals of Data Structures. Computer Science Press, Inc., Woodland Hills, California, 1976.

(14) Keehn, D. and J. Lacy. "VSAM Data Set Design Parameters." IBM Systems Journal, 13, 3 (1974), 185-212.

(15) Kim, W. "Relational Database Systems." ACM Computing Surveys, 11, 3 (September, 1979), 135-212.

(16) Knuth, D. The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley Publishing Co., Reading, Mass., 1973.

(17) Maruyama, K. and S. Smith "Analysis of Design Alternatives for Virtual Memory Indexes." Communications of the ACM, 20, 4 (April, 1977), 245-254.

(18) Mattson, R., Gecsei, J., Slutz, D., and Traiger, I. "Evaluation Techniques for Storage Heirarchies." IBM Systems Journal, 9, 2 (1970), 78-117.

(19) McCreight, E. "Pagination of B*-trees with Variable Length Records." Communications of the ACM, 20, 9 (September, 1977), 670-674.

(20) Senko, M. E., E. B. Altman, M. M. Astrahan, and P. L. Fehder. "Data Structures and Accessing in Database Systems." IBM Systems Journal, 12, 1 (1973), 30-93.

(21) Wagner, R. "Indexing Design Considerations." IBM Systems Journal, 12, 4 (1973), 351-367.

(22) Yao, A. C.-C. "On Random 2-3 Trees." Acta Informatica, 9 (1978), 159-170.

APPENDIX A

TEST RESULTS FOR B*-TREE DENSITIES

This data was obtained using the program TESTREE, listed in Appendix C. Each "Operation" is the insertion of a random key and the deletion of another random key. The "Number of Nodes" is the number of nodes on the bottom level of the tree. The "Density" is the density of the bottom level.

Case 1 - Order: 7 N: 2000

Number of Operations	Number of Nodes	Density
0	393	84.82
10	394	84.60
20	395	84.39
30	399	83.54
40	399	83.54
50	399	83.54
60	399	83.54
70	399	83.54
80	401	83.13
90	403	82.71
100	404	82.51
120	405	82.30
140	409	81.50
160	413	80.71
180	414	80.52
200	413	80.71
220	415	80.32
240	416	80.13
260	415	80.32
280	418	79.74
300	418	79.74
350	420	79.37
400	423	78.80
450	429	77.70
500	434	76.80
550	435	76.63
600	439	75.93
650	438	76.10
700	439	75.93
750	442	75.41
800	442	75.41
900	446	74.74
1000	447	74.57
1100	440	75.75
1200	438	76.10
1300	436	76.45

Case 2 - Order: 9 N: 1500

Number of Operations	Number of Nodes	Density
0	218	86.01
10	221	84.84
20	223	84.08
30	224	83.71
40	224	83.71
50	226	82.96
60	226	82.96
70	226	82.96
80	226	82.96
90	226	82.96
100	227	82.60
120	227	82.60
140	228	82.24
150	229	81.63
180	230	81.52
200	235	79.79
220	236	79.45
240	238	78.78
260	239	78.45
280	237	79.11
300	236	79.45
350	237	79.11
400	238	78.78
450	239	78.45
500	238	78.78
550	237	79.11
600	238	78.78
650	238	78.78
700	241	77.80
750	241	77.80
800	240	78.13
900	242	77.48
1000	243	77.16
1100	243	77.16
1200	247	75.91
1300	248	75.60

Case 3 - Order: 9   N: 1500

Number of Operations	Number of Nodes	Density
0	295	63.56
10	294	63.78
20	294	63.78
30	292	64.21
40	292	64.21
50	291	64.43
60	288	65.10
70	286	65.56
80	284	66.02
90	283	66.25
100	281	66.73
120	279	67.20
140	278	67.45
160	277	67.69
180	273	68.68
200	271	69.19
220	269	69.70
240	269	69.70
260	269	69.70
280	268	69.96
300	267	70.22
350	265	70.75
400	267	70.22
450	265	70.75
500	261	71.84
550	259	72.39
600	257	72.96
650	255	73.53
700	255	73.53
750	256	73.24
800	253	74.11
900	250	75.00
1000	250	75.00
1100	251	74.70
1200	243	77.16
1300	246	76.22

Case 4 - Order: 11   N: 1500

Number of Operations	Number of Nodes	Density
0	184	81.52
10	185	81.08
20	187	80.21
30	187	80.21
40	187	80.21
50	187	80.21
60	187	80.21
70	188	79.79
80	189	79.37
90	189	79.37
100	189	79.37
120	189	79.37
140	189	79.37
160	192	78.13
180	193	77.72
200	194	77.32
220	194	77.32
240	194	77.32
260	195	76.92
280	195	76.92
300	195	76.92
350	195	76.92
400	194	77.32
450	193	77.72
500	193	77.72
550	196	76.53
600	197	76.14
650	197	76.14
700	196	76.53
750	195	76.92
800	195	76.92
900	194	77.31
1000	196	76.53
1100	197	76.14
1200	194	77.32
1300	195	76.92

Case 5 - Order: 11 N: 1500

Number of Operations	Number of Nodes	Density
0	240	62.50
10	238	63.03
20	234	64.10
30	233	64.38
40	232	64.66
50	231	64.94
60	231	64.94
70	231	64.94
80	230	65.22
90	228	65.79
100	225	66.37
120	225	66.67
140	223	67.26
160	223	67.26
180	223	67.26
200	222	67.57
220	221	67.37
240	221	67.87
260	221	67.37
280	220	68.19
300	217	69.12
350	214	70.09
400	210	71.43
450	208	72.16
500	209	71.77
550	209	71.77
600	206	72.82
650	206	72.82
700	205	73.17
750	205	73.17
800	205	73.17
900	203	73.89
1000	202	74.25
1100	196	76.53
1200	198	75.76
1300	199	75.38

Case 6 - Order: 13   N: 2000

Number of Operations	Number of Nodes	Density
0	199	83.75
10	199	83.75
20	201	82.92
30	202	82.51
40	204	81.70
50	204	81.70
60	203	82.10
70	203	82.10
80	203	82.10
90	203	82.10
100	203	82.10
120	204	81.70
140	204	81.70
160	207	80.52
180	209	79.74
200	210	79.37
220	210	79.37
240	210	79.37
260	211	78.99
280	211	78.99
300	211	78.99
350	212	78.62
400	214	77.88
450	212	78.62
500	214	77.88
550	215	77.52
600	213	78.25
650	214	77.88
700	213	78.25
750	216	77.16
800	216	77.16
900	216	77.16
1000	218	76.45
1100	217	76.80
1200	216	77.16
1300	215	77.52

Case 7 - Order: 24 N: 2000

Number of Operations	Number of Nodes	Density
0	102	84.85
50	104	83.48
100	106	82.16
150	107	81.52
200	107	81.52
250	107	81.52
300	107	80.89
350	107	80.89
400	107	80.89
450	107	80.89
500	107	80.89
550	108	80.23
600	108	80.28
650	108	80.28
700	107	80.89
750	107	80.89
800	107	80.89
850	107	80.89
900	107	80.89
950	108	80.28
1000	108	80.23
1050	109	79.65
1100	108	80.28
1150	108	80.23
1200	110	79.05
1250	111	78.45
1300	111	78.45
1350	111	78.45
1400	110	79.05
1450	109	79.62
1500	109	79.62
1550	109	79.62
1600	109	79.62
1650	109	79.62
1700	109	79.62
1750	108	80.23
1800	108	80.23
1850	108	80.23
1900	108	80.23
1950	108	80.23
2000	108	80.23
2100	109	79.62
2200	109	79.62
2300	110	79.02
2400	110	79.02
2500	109	79.62

Case 8 - Order: 35   N: 3000

Number of Operations	Number of Nodes	Density
0	104	84.84
100	105	84.03
200	105	84.03
300	105	84.03
400	107	82.46
500	107	82.46
600	108	81.70
700	108	81.70
800	109	80.95
900	109	80.95
1000	109	80.95
1100	109	80.95
1200	109	80.95
1300	108	81.70
1400	108	81.70
1500	108	81.70
1600	108	81.70
1700	108	81.70
1800	108	81.70
1900	108	81.70
2000	108	81.70
2100	108	81.70
2200	109	80.95
2350	109	80.95
2500	109	80.95
2650	109	80.95
2800	109	80.95
2950	109	80.95

Case 9 - Order: 35 N: 3000

Number of Operations	Number of Nodes	Density
0	142	62.14
100	141	62.58
200	138	63.94
300	137	64.41
400	135	65.36
500	134	65.85
600	133	66.34
700	133	66.34
800	133	66.34
900	131	67.36
1000	128	68.93
1100	128	68.93
1200	128	68.93
1300	127	69.48
1400	127	69.48
1500	127	69.48
1600	127	69.48
1700	126	70.03
1800	124	71.16
1900	124	71.16
2000	123	71.74
2100	121	72.92
2200	121	72.92
2350	120	73.53
2500	120	73.53
2650	120	73.53
2800	120	73.53
2950	119	74.15

Case 10 - Order: 49 N: 3000

Number of Operations	Number of Nodes	Density
0	74	84.46
100	74	84.46
200	74	84.46
300	75	83.33
400	75	83.33
500	76	82.24
600	76	82.24
700	76	82.24
800	76	82.24
900	76	82.24
1000	77	81.17
1100	77	81.17
1200	77	81.17
1300	77	81.17
1400	77	81.17
1500	77	81.17
1600	77	81.17
1700	77	81.17
1800	77	81.17
1900	78	80.13
2000	78	80.13
2100	78	80.13
2200	78	80.13
2350	78	80.13
2500	78	80.13
2650	78	80.13
2800	78	80.13
2950	78	80.13

Case 11 - Order: 49 N: 3000

Number of Operations	Number of Nodes	Density
0	99	63.13
100	98	63.78
200	97	64.43
300	96	65.10
400	96	65.10
500	95	65.79
600	94	66.49
700	94	66.49
800	94	66.49
900	92	67.93
1000	92	67.93
1100	92	67.93
1200	92	67.93
1300	92	67.93
1400	90	69.44
1500	88	71.02
1600	88	71.02
1700	88	71.02
1800	88	71.02
1900	88	71.02
2000	88	71.02
2100	87	71.84
2200	87	71.84
2350	87	71.84
2500	87	71.84
2650	87	71.84
2800	87	71.84
2950	87	71.84

APPENDIX B

TEST RESULTS FOR BUFFERED B+-TREES

The following data was obtained from the program TESTREE, listed in Appendix C, for B+-trees of several sizes and orders. Three operations were performed: insertion of random keys (Insert), searching for random keys (Search), and alternate insertion and deletion of random keys (Alternate). Each "Alternate" operation consists of an insertion and deletion pair. Cases 1 through 16 are from runs using the Least Recently Used Replacement method. Cases 17 through 28 are from runs using the Height Weighted buffering method.

Case 1 - Order: 50 Buffer: 20

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	1200	122	100
Alternate	1200	866	858
Search	1200	420	2
Insert	1200	833	793
Alternate	2400	3534	3510
Search	2400	1695	0
Insert	2601	2631	2485
Alternate	5000	8714	8562
Search	5000	4309	0

Case 2 - Order: 50 Buffer: 10

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	100	4	0
Alternate	100	0	0
Search	100	0	0
Insert	200	5	0
Alternate	300	0	0
Search	300	0	0
Insert	300	66	64
Alternate	600	496	489
Search	600	257	1
Insert	600	440	415
Alternate	1200	1678	1656
Search	1200	836	0
Insert	1200	1247	1169
Alternate	2400	4422	4377
Search	2400	2124	0

Case 3 - Order: 50 Buffer: 5

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	100	4	0
Alternate	100	0	0
Search	100	0	0
Insert	200	40	38
Alternate	300	254	250
Search	300	152	2
Insert	300	238	226
Alternate	600	909	904
Search	600	429	0
Insert	600	608	563
Alternate	1200	2175	2155
Search	1200	1051	0
Insert	1200	1496	1354
Alternate	2400	5989	4899
Search	2400	2902	0

Case 4 - Order: 50 Buffer: 1

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	100	106	59
Alternate	100	403	205
Search	100	200	0
Insert	200	426	247
Alternate	300	1214	625
Search	300	600	0
Insert	300	668	403
Alternate	600	2420	1239
Search	600	1200	0
Insert	600	1351	829
Alternate	1200	4890	2551
Search	1200	2400	0
Insert	1200	2994	1683
Alternate	2400	14592	5113
Search	2400	7200	0

Case 5 - Order: 24 Buffer: 20

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	600	113	90
Alternate	600	625	608
Search	600	322	1
Insert	600	543	490
Alternate	1200	1954	1922
Search	1200	933	0
Insert	1200	1540	1372
Alternate	2400	5069	4695
Search	2400	1695	0
Insert	2601	4410	3623
Alternate	5000	13900	9856
Search	5000	6800	0

Case 6 - Order: 24 Buffer: 10

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	300	58	48
Alternate	300	297	279
Search	300	142	0
Insert	300	290	269
Alternate	600	1037	1018
Search	600	468	0
Insert	600	786	709
Alternate	1200	2547	2366
Search	1200	1200	0
Insert	1200	2103	1729
Alternate	2400	7155	5116
Search	2400	3405	0
Insert	5000	8563	6745
Alternate	5000	15918	10048
Search	5000	8386	0

Case 7 - Order: 24 Buffer: 5

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	100	7	2
Alternate	100	37	35
Search	100	17	3
Insert	200	173	158
Alternate	300	479	467
Search	300	243	0
Insert	300	436	378
Alternate	600	1512	1251
Search	600	694	0
Insert	600	1089	890
Alternate	1200	3564	2585
Search	1200	1688	0
Insert	1200	2562	1900
Alternate	2400	8514	5254
Search	2400	4092	0

Case 8 - Order: 24 Buffer: 1

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	50	58	35
Alternate	50	200	99
Search	50	100	0
Insert	50	117	77
Alternate	100	421	235
Search	100	200	0
Insert	200	472	307
Alternate	300	1244	675
Search	300	600	0
Insert	300	885	505
Alternate	600	3696	1355
Search	600	1800	0
Insert	600	2127	1003
Alternate	1200	7377	2671
Search	1200	3600	0
Insert	1200	4257	2001
Alternate	2400	14727	5331
Search	2400	7200	0

Case 9 - Order: 12 Buffer: 20

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	300	106	82
Alternate	300	396	374
Search	300	187	0
Insert	300	389	333
Alternate	600	1315	1164
Search	600	580	0
Insert	600	1099	886
Alternate	1200	3749	2695
Search	1200	1747	0
Insert	1200	2772	2050
Alternate	2400	8388	4783
Search	2400	4123	0

Case 10 - Order: 12 Buffer: 10

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	100	12	2
Alternate	100	59	57
Search	100	28	5
Insert	200	260	222
Alternate	300	632	584
Search	300	322	0
Insert	300	551	443
Alternate	600	1855	1371
Search	600	862	0
Insert	600	1338	1013
Alternate	1200	4656	2932
Search	1200	2241	0
Insert	1200	3241	2193
Alternate	2400	10448	4956
Search	2400	5119	0

Case 11 - Order: 12 Buffer: 5

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	50	9	4
Alternate	50	44	43
Search	50	18	2
Insert	50	44	41
Alternate	100	151	138
Search	100	60	0
Insert	200	423	349
Alternate	300	940	689
Search	300	454	0
Insert	300	706	534
Alternate	600	2223	1446
Search	600	1036	0
Insert	600	1553	1112
Alternate	1200	5708	3004
Search	1200	2926	0

Case 12 - Order: 12 Buffer: 1

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	50	91	61
Alternate	50	208	115
Search	50	100	0
Insert	50	131	91
Alternate	100	435	251
Search	100	200	0
Insert	200	767	413
Alternate	300	1902	751
Search	300	900	0
Insert	300	1127	563
Alternate	600	3808	1499
Search	600	1800	0
Insert	600	2293	1151
Alternate	1200	9061	3035
Search	1200	4800	0

Case 13 - Order: 6 Buffer: 20

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	300	455	356
Alternate	300	1051	750
Search	300	458	0
Insert	300	848	603
Alternate	600	2846	1685
Search	600	1322	0
Insert	600	2124	1328
Alternate	1200	6478	2628
Search	1200	3195	0

Case 14 - Order: 6 Buffer: 10

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	50	26	15
Alternate	50	88	76
Search	50	38	0
Insert	50	92	71
Alternate	100	268	206
Search	100	123	0
Insert	200	598	442
Alternate	300	1457	848
Search	300	655	0
Insert	300	1048	677
Alternate	600	3585	1760
Search	600	1720	0
Insert	600	2537	1371
Alternate	1200	7437	2655
Search	1200	3637	0

Case 15 - Order: 6 Buffer: 5

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	50	60	45
Alternate	50	149	107
Search	50	73	0
Insert	50	123	95
Alternate	100	379	261
Search	100	147	0
Insert	200	796	481
Alternate	300	1790	883
Search	300	801	0
Insert	300	1268	702
Alternate	600	4208	1788
Search	600	1956	0
Insert	600	3195	1385
Alternate	1200	8499	2669
Search	1200	4051	0

Case 16 - Order: 6   Buffer: 1

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	50	151	87
Alternate	50	326	133
Search	50	150	0
Insert	50	210	111
Alternate	100	670	293
Search	100	300	0
Insert	200	1069	497
Alternate	300	2623	895
Search	300	1200	0
Insert	300	1601	705
Alternate	600	5657	1795
Search	600	3000	0
Insert	600	3786	1391
Alternate	1200	12227	2679
Search	1200	6000	0

Case 17 - Order: 24   Buffer: 10
Height Weighting Factor: 1

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	300	54	43
Alternate	300	280	277
Search	300	135	8
Insert	300	289	267
Alternate	600	1027	1016
Search	600	460	6
Insert	600	751	680
Alternate	1200	2331	2305
Search	1200	1101	6
Insert	1200	1935	1659
Alternate	2400	6642	5127
Search	2400	3163	5

Case 18 - Order: 24 Buffer: 10
Height Weighting Factor: 2

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	300	54	43
Alternate	300	272	269
Search	300	143	7
Insert	300	296	269
Alternate	600	1015	1005
Search	600	463	5
Insert	600	745	672
Alternate	1200	2365	2338
Search	1200	1096	4
Insert	1200	1810	1592
Alternate	2400	6120	5120
Search	2400	2919	4

Case 19 - Order: 24 Buffer: 10
Height Weighting Factor: 4

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	300	54	43
Alternate	300	272	269
Search	300	143	7
Insert	300	298	268
Alternate	600	1011	1002
Search	600	462	4
Insert	600	756	639
Alternate	1200	2364	2337
Search	1200	1096	4
Insert	1200	1801	1588
Alternate	2400	6018	5143
Search	2400	2827	1

Case 20 - Order: 24 Buffer: 10
Height Weighting Factor: 6

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	300	54	43
Alternate	300	272	269
Search	300	143	7
Insert	300	301	270
Alternate	600	1011	1002
Search	600	462	4
Insert	600	757	679
Alternate	1200	2364	2337
Search	1200	1096	4
Insert	1200	1725	1541
Alternate	2400	5711	5134
Search	2400	2688	2

Case 21 - Order: 24 Buffer: 10
Height Weighting Factor: 8

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	300	54	43
Alternate	300	272	269
Search	300	143	7
Insert	300	298	268
Alternate	600	1004	997
Search	600	466	2
Insert	600	752	673
Alternate	1200	2369	2342
Search	1200	1103	2
Insert	1200	1705	1523
Alternate	2400	5668	5133
Search	2400	2687	2

Case 22 - Order: 24 Buffer: 10
Height Weighting Factor: 10

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	300	54	43
Alternate	300	272	269
Search	300	143	7
Insert	300	296	265
Alternate	600	1005	995
Search	600	467	1
Insert	600	762	684
Alternate	1200	2356	2332
Search	1200	1100	1
Insert	1200	1711	1532
Alternate	2400	5667	5133
Search	2400	2687	2

Case 23 - Order: 12 Buffer: 10
Height Weighting Factor: 1

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	100	13	3
Alternate	100	55	55
Search	100	29	8
Insert	200	261	223
Alternate	300	566	552
Search	300	274	7
Insert	300	509	427
Alternate	600	1711	1361
Search	600	785	7
Insert	600	1283	994
Alternate	1200	4348	2891
Search	1200	2030	4
Insert	1200	3038	2193
Alternate	2400	10064	6003
Search	2400	4594	5

Case 24 - Order: 12 Buffer: 10
Height Weighting Factor: 2

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	100	14	4
Alternate	100	60	60
Search	100	27	7
Insert	200	255	214
Alternate	300	560	544
Search	300	260	4
Insert	300	480	411
Alternate	600	1592	1349
Search	600	729	6
Insert	600	1236	978
Alternate	1200	4178	2883
Search	1200	1974	3
Insert	1200	2977	2170
Alternate	2400	9700	5985
Search	2400	4463	4

Case 25 - Order: 12 Buffer: 10
Height Weighting Factor: 4

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	100	14	4
Alternate	100	60	60
Search	100	27	7
Insert	200	247	207
Alternate	300	556	540
Search	300	260	3
Insert	300	474	407
Alternate	600	1566	1351
Search	600	717	4
Insert	600	1244	981
Alternate	1200	4173	2897
Search	1200	1966	2
Insert	1200	2973	2174
Alternate	2400	9659	5951
Search	2400	4454	4

Case 26 - Order: 12 Buffer: 10
Height Weighting Factor: 6

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	100	14	4
Alternate	100	60	60
Search	100	27	7
Insert	200	241	205
Alternate	300	557	541
Search	300	260	3
Insert	300	463	399
Alternate	600	1483	1345
Search	600	676	4
Insert	600	1205	969
Alternate	1200	4260	2382
Search	1200	2024	1
Insert	1200	3014	2181
Alternate	2400	9822	5938
Search	2400	4522	1

Case 27 - Order: 12 Buffer: 10
Height Weighting Factor: 8

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	100	14	4
Alternate	100	60	60
Search	100	27	7
Insert	200	240	199
Alternate	300	573	558
Search	300	258	2
Insert	300	453	393
Alternate	600	1459	1344
Search	600	676	3
Insert	600	1188	963
Alternate	1200	4301	2884
Search	1200	2031	2
Insert	1200	3012	2172
Alternate	2400	9823	5939
Search	2400	4522	1

Case 28 - Order: 12 Buffer: 10
Height Weighting Factor: 10

Operation Type	Number of Operations	Number of Reads	Number of Writes
Insert	100	14	4
Alternate	100	60	60
Search	100	27	7
Insert	200	244	202
Alternate	300	571	554
Search	300	260	1
Insert	300	456	391
Alternate	600	1456	1342
Search	600	676	3
Insert	600	1186	961
Alternate	1200	4301	2894
Search	1200	2053	1
Insert	1200	3007	2157
Alternate	2400	9833	5927
Search	2400	4520	1

APPENDIX C

LISTING OF COMPUTER PROGRAMS

This appendix contains listings of the PL/I program and procedures used to obtain empirical data given in Appendixes A and B. The programs were compiled on the PL/I Optimizing compiler and run on an IBM 370/168 computer. The main program, TESTREE, is listed, followed by the procedures BTREE, INDEXIO, GOFIND, TRAVEL, and RANF.

```
/*                        TESTREE                              */
TESTREE: PROC OPTIONS (MAIN);

/*

AUTHOR:  ROBERT WEBSTER
         DEPARTMENT OF COMPUTER SCIENCE
         OKLAHOMA STATE UNIVERSITY
         1979

THIS PROGRAM IS USED TO TEST THESE PROCEDURES:

  1. BTREE - INSERT AND DELETE FROM A MODIFIED BTREE.
  2. INDEXIO - PERFORM BUFFERED I/O ON INDEX NODES.
  3. GOFIND - SEARCH THE BTREE FOR A GIVEN KEY.
  4. TRAVEL - TRAVERSE THE TREE USING THE BOTTOM LEVEL

FILES REQUIRED:

  1. SYSIN
  2. SYSPRINT
  3. BINDEX - REGIONAL(1), BLKSIZE(1000)

INPUT:

  AT THE BEGINNING OF THE PROGRAM, THREE PARAMETERS ARE
  READ FROM SYSIN IN FREE FORMAT:

    1. MAX_BRANCH - MAXIMUM BRANCHING FACTOR OF THE TREE.
    2. MAX_KEYS - MAXIMUM NUMBER OF KEYS TO BE PLACED INTO
       THE TREE AT ONE TIME.

  AFTER THESE ARE INPUT, THE OPERATIONS MAY TAKE PLACE.


ANY OF 8 OPERATIONS MAY BE SPECIFIED FROM FILE SYSIN. SOME
SOME OF THESE USE A COUNT FIELD.  OPERATIONS ARE SPECIFIED
BY ENTERING A NUMBER FROM ONE TO EIGHT IN THE FIRST TWO
COLUMNS.  COUNTS ARE ENTERED IN COLUMNS 3 THROUGH 15.

THE OPERATIONS AVAILABLE ARE AS FOLLOWS:

    1. INSERT RANDOM ELEMENTS.  COUNT SPECIFIES HOW MANY.
    2. DELETE RANDOM ELEMENTS.  COUNT SPECIFIES HOW MANY.
    3. SEARCH FOR RANDOM ELEMENTS.  COUNT SPECIFIES HOW
       MANY.
    4. TRAVERSE THE TREE USING BOTTOM LEVEL LINKS.  COUNT
       IS NOT USED.
    5. SETUP A NEW TREE.  COUNT IS USED TO TELL HOW MANY
       NODES ARE TO BE PLACED IN THE AVAILABLE LIST.
    6. WRITE THE INDEX NODES REMAINING IN THE BUFFER OUT TO
       THE FILE.  THIS SHOULD ALWAYS BE DONE AT THE END OF
       THE PROGRAM.  COUNT IS NOT USED.
    7. COUNT THE NUMBER OF NODES AND KEYS AT EACH LEVEL,
```

```
      USING AN INORDER TRAVERSAL OF THE TREE.
      COUNT IS NOT USED./
   8. PERFORM ALTERNATE INSERTIONS AND DELETIONS OF RANDOM
      KEYS.  COUNT TELLS HOW MANY OF EACH OPERATION.


VARIABLES:

  ACTION - VARIABLE THAT TELLS WHICH OPERATION TO PERFORM.
  ARRAY - ARRAY OF RANDOM INTEGERS USED TO MAKE KEYS.
  COUNT - VARIABLE THAT TELLS HOW MANY TIMES TO DO AN
          OPERATION.
  EOF - FLAG TO SIGNAL THE END OF THE TRAVERSAL.
  FIRST - POINTER TO THE FIRST ELEMENT IN "ARRAY" THAT IS
          IN THE TREE.
  FOUND - FLAG SET BY "GOFIND" TELLING WHETHER A KEY WAS
          FOUND.
  J, K - TEMPORARY VARIABLES.
  KEY - CHARACTER KEY.
  KEY# - POSITION OF THE KEY WITHIN THE CURRENT RECORD.
  KEYLENGTH - CHARACTER LENGTH OF KEYS IN THE TREE.
  LAST - POINTER TO THE LAST ELEMENT IN "ARRAY" THAT IS IN
         THE TREE.
  MAX_BRANCH - ORDER OF THE TREE.
  MAX_KEYS - MAXIMUM NUMBER OF KEYS PER NODE.
           CURRENT KEY.
  MAX_NODES - MAXIMUM NUMBER OF NODES IN THE TREE.
  POINTER - POINTER ASSOSIATED WITH THE KEY IN THE TREE.
  RECORD# - INDEX NODE NUMBER THAT CONTAINS THE CURRENT KEY.
  RESULT - RESULT FLAG FROM "BTREE".
  ROOT - ROOT NODE OF THE TREE.
  STAT - 6 MEMBER ARRAY GIVING STATISTICS FOR AN OPERATION:
         STAT(1) - NUMBER OF NODE READS
         STAT(2) - NUMBER OF NODE WRITES
         STAT(3) - NUMBER OF NODE SPLITS
         STAT(4) - NUMBER OF INSERTION SHARES
         STAT(5) - NUMBER OF DELETION SHARES
         STAT(6) - NUMBER OF NODE MERGES

  1 NODE - NODE STRUCTURE USED IN "INDEXIO"
    2 NO_KEYS - NUMBER OF KEYS (OR LINK ON AVAILABLE LIST)
    2 KEYS - KEYS OF THE NODE
    2 PTRS - POINTERS OF THE NODE

*/

DCL
(MAX_BRANCH, MAX_KEYS)  FIXED BIN,
KEYLENGTH  FIXED BIN  INIT(9),
BINDEX  FILE  ENV (REGIONAL(1),RECSIZE(1000)),

INDEXIO EXTERNAL ENTRY (FIXED BIN, 1, 2 FIXED BIN,
  2 (*) CHAR (*), 2 (*) FIXED BIN (31,0), FIXED BIN,
```

```
  FIXED BIN, FIXED BIN, (*) FIXED BIN);

BTREE EXTERNAL ENTRY (CHAR(*), FIXED BIN (31,0), FIXED BIN,
 FIXED BIN, FIXED BIN, FIXED BIN, FIXED BIN, (*) FIXED BIN),

GOFIND EXTERNAL ENTRY (CHAR(*), FIXED BIN, FIXED BIN,
  FIXED BIN, FIXED BIN, FIXED BIN, FIXED BIN (31,0),
  BIT(*), (*) FIXED BIN),

TRAVEL EXTERNAL ENTRY (FIXED BIN,FIXED BIN,FIXED BIN(31,0),
  CHAR(*), FIXED BIN, FIXED BIN, (*) FIXED BIN, BIT(*)),

RANF  EXTERNAL ENTRY (FIXED BIN (31,0)) RETURNS (FLOAT BIN),

TRUE  BIT(1) INIT ('1'B),
FALSE BIT(1) INIT ('0'B);



GET FILE (SYSIN) LIST (MAX_BRANCH, MAX_KEYS);

BEGIN;

DCL
(ARRAY(MAX_KEYS), POINTER)  FIXED BIN (31,0),
KEY  CHAR (KEYLENGTH),
(FIRST,LAST,STAT(6),J,K,ROOT,ACTION,COUNT,RESULT,RECORD#,
  KEY#, NUMKEYS(30), NUMNODES(30))  FIXED BIN,
LOW  BUILTIN,
(EOF, FOUND)  BIT(1),

1 NODE,
  2 NO_KEYS  FIXED BIN,
  2 KEYS (MAX_BRANCH-1)  CHAR(KEYLENGTH),
  2 PTRS (0:MAX_BRANCH-1)  FIXED BIN (31,0);


/*  SET UP KEY ARRAY  */
DO J = 1 TO MAX_KEYS;
  ARRAY(J) = RANF(0) * 10 ** (KEYLENGTH - 1);
  END;

/*  SETUP FILE  */
J = MAX_KEYS / MAX_BRANCH * 2;
CALL SETUP (J);

ON ENDFILE (SYSIN) STOP;

/*  MAIN LOOP  */
DO WHILE (TRUE);

  GET FILE (SYSIN) EDIT (ACTION,COUNT)(COL(1),F(2),F(13));
  STAT = 0;
```

```
SELECT (ACTION);

  WHEN (1) DO;  /*  INSERT  */
    PUT EDIT ('INSERT ', COUNT) (SKIP(5), A, F(5));
    DO J = 1 TO COUNT;
      /*  INCREMENT POINTER TO LAST KEY IN TREE */
      LAST = LAST + 1;
      IF LAST > MAX_KEYS THEN LAST = 1;
      PUT STRING (KEY) EDIT (ARRAY(LAST)) (F(KEYLENGTH));
      CALL BTREE (KEY, ARRAY(LAST), 1, ROOT, KEYLENGTH,
        MAX_BRANCH, RESULT, STAT);
      /*  CHECK FOR ERROR  */
      IF RESULT ¬= 0 THEN
        PUT EDIT ('** ERROR ** RESULT,KEY: ',RESULT,KEY)
        (SKIP(2), A, F(5), A);
      END;
    END;  /*  INSERT  */

  WHEN (2) DO;  /*  DELETE  */
    PUT EDIT ('DELETE ', COUNT) (SKIP(5), A, F(5));
    DO J = 1 TO COUNT;
      PUT STRING (KEY) EDIT (ARRAY(FIRST)) (F(KEYLENGTH));
      CALL BTREE (KEY, ARRAY(FIRST), 2, ROOT, KEYLENGTH,
        MAX_BRANCH, RESULT, STAT);
      /*  CHECK FOR ERROR  */
      IF RESULT ¬= 0 THEN
        PUT EDIT ('** ERROR ** RESULT, KEY: ', RESULT, KEY)
        (SKIP(2), A, F(5), A);
      /*  INCREMENT POINTER TO FIRST KEY IN TREE */
      FIRST = FIRST + 1;
      IF FIRST > MAX_KEYS THEN FIRST = 1;
      END;
    END;  /*  DELETE  */

  WHEN (3) DO;  /*  SEARCH  */
    PUT EDIT ('SEARCH ', COUNT) (SKIP(5), A, F(5));
    /*  K IS THE POINTER TO THE NEXT KEY TO BE HUNTED  */
    K = FIRST - 1;
    DO J = 1 TO COUNT;
      K = K + 1;
      IF K > MAX_KEYS THEN K = 1;
      PUT STRING (KEY) EDIT (ARRAY(K)) (F(KEYLENGTH));
      CALL GOFIND (KEY,ROOT,KEYLENGTH,MAX_BRANCH,RECORD#,
        KEY#, POINTER, FOUND, STAT);
      IF FOUND & POINTER ¬= ARRAY(K) THEN
        PUT EDIT ('**ERROR** KEY AND POINTER DO NOT MATCH',
        'KEY,POINTER: ',KEY,POINTER) (SKIP(2),A,A,A,F(9));
      ELSE IF ¬ FOUND THEN
        PUT EDIT ('KEY NOT FOUND: ', KEY, POINTER)
        (SKIP, A, A, F(9));
      END;
    END;  /*  SEARCH  */

  WHEN (4) DO;  /*  TRAVERSE  */
```

```
    PUT EDIT ('TRAVERSE') (SKIP(5), A);
    KEY = LOW(KEYLENGTH);
    CALL GOFIND (KEY, ROOT, KEYLENGTH, MAX_BRANCH, RECORD#,
      KEY#, POINTER, FOUND, STAT);
    IF FOUND THEN PUT EDIT ('** ERROR ** LOW KEY FOUND: ',
      POINTER) (SKIP(2), A, F(9));

    EOF = FALSE;
    DO J = 1 TO MAX_KEYS WHILE (¬ EOF);
      CALL TRAVEL (RECORD#, KEY#, POINTER, KEY, KEYLENGTH,
        MAX_BRANCH, STAT, EOF);
      IF ¬ EOF THEN PUT EDIT (KEY) (A);
      END;
    END;  /*  TRAVERSE  */

  WHEN (5) DO;  /*  SETUP NEW TREE  */
    PUT EDIT ('SETUP NEW TREE', COUNT) (SKIP(5), A, F(5));
    CALL SETUP (COUNT);
    END;  /*  SETUP  */

  WHEN (6) DO;  /*  WRITE OUT BUFFERS  */
    PUT EDIT ('WRITE OUT BUFFERS') (SKIP(5), A);
    CALL INDEXIO(5,NODE,RECORD#,KEYLENGTH,MAX_BRANCH,STAT);
    /*  RECORD# IS NOT USED IN THE ABOVE CALL  */
    END;  /*  WRITE OUT BUFFERS  */

  WHEN (7) DO;  /*  TRAVERSE, COUNT KEYS & NODES  */
    PUT SKIP(5) LIST ('STORAGE CHARACTERISTICS');
    NUMKEYS = 0;
    NUMNODES = 0;
    CALL TRAVERSE (ROOT, NUMKEYS, NUMNODES, 0);
    PUT EDIT ('LEVEL', 'KEYS', 'NODES')
      (SKIP(2), A, COL(8), A, COL(14), A);
    DO J = 1 TO 30 WHILE (NUMNODES(J) > 0);
      PUT EDIT (J, '.', NUMKEYS(J), NUMNODES(J))
        (SKIP, F(2), A, COL(6), F(6), COL(13), F(5));
      END;
    END;  /*  TRAVERSE  */

  WHEN (8) DO;  /* ALTERNATE INSERTIONS AND DELETIONS  */
    PUT EDIT ('ALTERNATE INSERTIONS AND DELETIONS', COUNT)
      (SKIP(5), A, F(5));
    DO J = 1 TO COUNT;

      /*  DELETE A KEY  */
      PUT STRING (KEY) EDIT (ARRAY(FIRST)) (F(KEYLENGTH));
      CALL BTREE (KEY, ARRAY(FIRST), 2, ROOT, KEYLENGTH,
        MAX_BRANCH, RESULT, STAT);
      /*  CHECK FOR ERROR  */
      IF RESULT ¬= 0 THEN
        PUT EDIT ('** ERROR ** RESULT, KEY: ', RESULT, KEY)
        (SKIP(2), A, F(3), A);
      /*  INCREMENT POINTER TO FIRST KEY IN TREE */
      FIRST = FIRST + 1;
```

```
          IF FIRST > MAX_KEYS THEN FIRST = 1;

          /*  INSERT A KEY   */
          /*  INCREMENT POINTER TO LAST KEY IN TREE */
          LAST = LAST + 1;
          IF LAST > MAX_KEYS THEN LAST = 1;
          PUT STRING (KEY) EDIT (ARRAY(LAST)) (F(KEYLENGTH));
          CALL BTREE (KEY, ARRAY(LAST), 1, ROOT, KEYLENGTH,
            MAX_BRANCH, RESULT, STAT);
          /*  CHECK FOR ERROR   */
          IF RESULT ¬= 0 THEN
            PUT EDIT ('** ERROR ** RESULT, KEY: ', RESULT, KEY)
            (SKIP(2), A, F(5), A);

          END;
        END;  /*  ALTERNATE INSERTIONS AND DELETIONS   */

      OTHERWISE PUT EDIT ('INVALID OPERATION: ', ACTION)
        (SKIP(3), A, F(5));

      END;  /*  SELECT   */

    PUT EDIT('NODE READS: ',STAT(1))(SKIP(3),A,COL(20),F(5));
    PUT EDIT('NODE WRITES: ',STAT(2))(SKIP,A,COL(20),F(5));
    PUT EDIT('NODE SPLITS: ',STAT(3))(SKIP,A,COL(20),F(5));
    PUT EDIT('INSERTION SHARES:',STAT(4))(SKIP,A,COL(20),F(5));
    PUT EDIT('DELETION SHARES:',STAT(5))(SKIP,A,COL(20),F(5));
    PUT EDIT('NODE MERGES:',STAT(6))(SKIP,A,COL(20),F(5));

    END;  /*  MAIN LOOP   */

  SETUP: PROC (MAX_NODES);

    /*  THIS PROCEDURE SETS UP A THE LINKED LIST OF AVAILABLE
  NODES FOR THE PROCEDURE "INDEXIO" TO USE.  MAX_NODES
  TELLS HOW MANY NODES TO PLACE IN THE AVAILABLE LIST.

  THE INDEX FILE MUST HAVE A BLOCKSIZE OF 1000 BYTES.

  */

  DCL 1 NODE,        /*  I/O STRUCTURE FOR "BINDEX"   */
        2 LINK  FIXED BIN,
        2 REST  CHAR(998)  INIT (' '),
  (J, MAX_NODES)  FIXED BIN;

  OPEN FILE (BINDEX) DIRECT OUTPUT;

  LAST, ROOT = 0;
  FIRST = 1;

  DO J = 1 TO MAX_NODES;
    LINK = J;
    WRITE FILE (BINDEX) FROM (NODE) KEYFROM (J-1);
```

```
  END;

LINK = 0;
WRITE FILE (BINDEX) FROM (NODE) KEYFROM (J);

CLOSE FILE (BINDEX);
OPEN FILE (BINDEX) DIRECT UPDATE;

END;  /*  SETUP  */

TRAVERSE: PROC (RECORD#, NUMKEYS, NUMNODES, LEV) RECURSIVE;

/*  THIS PROCEDURE TRAVERSES THE TREE INORDER RECURSIVELY
AND COUNTS THE NUMBER OF KEYS AND NODES AT EACH LEVEL.

  PARAMETERS
    RECORD# - CURRENT INDEX NODE NUMBER
    NUMKEYS - NUMBER OF KEYS ON AT EACH LEVEL
    NUMNODES - NUMBER OF NODES AT EACH LEVEL
    LEV - CURRENT LEVEL (ROOT = 1).


GLOBAL VARIABLES:
  KEYLENGTH - LENGTH OF KEYS
  MAX_BRANCH - MAXIMUM BRANCHING FACTOR FOR TREE.
  STAT - STATISTICS FOR TREE.
*/
DCL
1 NODE,
  2 NO_KEYS  FIXED BIN,
  2 KEYS(MAX_BRANCH-1)  CHAR(KEYLENGTH),
  2 PTRS(MAX_BRANCH)  FIXED BIN (31,0),

(RECORD#, NUMKEYS(*), NUMNODES(*), LEV)  FIXED BIN,
DEBUG BIT(1) INIT ('0'B),
J  FIXED BIN;

IF DEBUG THEN PUT SKIP LIST('TRAVERSE', LEV);
IF RECORD# <= 0 THEN RETURN;
CALL INDEXIO (1,NODE,RECORD#,KEYLENGTH,MAX_BRANCH,STAT);
LEV = LEV + 1;
NUMKEYS(LEV) = NUMKEYS(LEV) + NO_KEYS;
NUMNODES(LEV) = NUMNODES(LEV) + 1;
IF PTRS(2) > 0 THEN DO J = 1 TO NO_KEYS + 1;
  CALL TRAVERSE (PTRS(J), NUMKEYS, NUMNODES, LEV);
  END;
LEV = LEV - 1;
RETURN;
END;  /*  TRAVERSE  */

END;  /*  BEGIN BLOCK  */

END;  /*  TESTREE  */
```

```
/*                          BTREE                                   */
BTREE: PROC (KEY, KEYPOS, ACTION, ROOT, KEYLENGTH,
             MAX_BRANCH, RESULT, STAT);

  /*

 AUTHOR:  ROBERT WEBSTER
          DEPARTMENT OF COMPUTER SCIENCE
          OKLAHOMA STATE UNIVERSITY
          1979

  THIS PROCEDURE PERFORMS MAINTENANCE ON A MODIFIED B-TREE
STRUCTURE.  KEYS AND POINTERS MAY BE INSERTED AND DELETED.

   THE STRUCTURE USED IS A B+TREE.  THIS IS A NORMAL B-TREE
ON UPPER LEVELS, EXCEPT THAT ONLY KEYS AND POINTERS TO OTHER
NODES ARE STORED.  ON THE BOTTOM LEVEL OF THE TREE, ALL THE
POINTERS ARE NEGATIVE.  THE FIRST POINTER OF EACH BOTTOM
LEVEL NODE POINTS TO ITS RIGHT SIBLING.  THE FIRST POINTER ON
THE RIGHTMOST BOTTOM LEVEL NODE IS ZERO.  THE OTHER POINTERS
ON THE BOTTOM LEVEL NODES POINT TO EXTERNAL RECORDS
REPRESENTED BY THEIR ASSOCIATED KEY.  EACH KEY HAS A POINTER
("KEYPOS") THAT WAS INPUT AT THE TIME THE KEY WAS INSERTED
INTO THE TREE.  WHEN A NODE IS SPLIT ON THE BOTTOM LEVEL,
THE KEY PROPAGATED TO THE UPPER LEVEL IS NOT REMOVED FROM THE
BOTTOM LEVEL.  SIMILARLY, KEYS ARE NOT REMOVED FROM OR ADDED
TO THE BOTTOM LEVEL DURING A SHARING OPERATION.  THIS MEANS
THAT ALL KEYS ON UPPER LEVELS WERE DUPLICATED FROM BOTTOM
LEVEL KEYS.


  INPUT PARAMETERS:
    KEY - KEY TO BE INSERTED OR DELETED FROM THE TREE.
    KEYPOS - POINTER TO BE SET AT THE BOTTOM OF THE TREE.
    ACTION - FLAG TELLING WHETHER TO INSERT OR DELETE.
             1 IS FOR INSERTION, 2 IS FOR DELETION.
    ROOT - IS THE NUMBER OF THE ROOT NODE OF THE TREE.
    KEYLENGTH - LENGTH OF KEYS IN THE TREE.
    MAX_BRANCH - MAXIMUM BRANCHING FACTOR OF THE TREE.


  OUTPUT PARAMETERS:

    RESULT - RESULT CODE:
             0 => SUCCESS
             1 => KEY ALREADY EXISTS; INSERTION NOT DONE
             2 => KEY NOT FOUND; DELETION NOT DONE
             3 => OUT OF NODES; TRANSACTION NOT DONE
    STAT - ARRAY GIVING COUNTS OF ACTIONS WITHIN THE PROGRAM:
             STAT(1) - NUMBER OF NODE READS
             STAT(2) - NUMBER OF NODE WRITES
             STAT(3) - NUMBER OF NODE SPLITS
             STAT(4) - NUMBER OF SHARES DURING INSERTION
             STAT(5) - NUMBER OF SHARES DURING DELETION
```

STAT(6) - NUMBER OF NODE MERGES

PROCEDURES CALLED: INDEXIO

INTERNAL PROCEDURES:

BTREE - MAIN PROCEDURE
SEARCH - SEARCHES THE TREE FOR A KEY, RETURNS ITS POSITION IF THE KEY IS FOUND, AND THE POSITION OF THE NEXT HIGHER KEY IF IT IS NOT FOUND.
INSERT - DOES AN INSERTION INTO THE TREE.
DEL - DOES A DELETION FROM THE TREE.
OVERLEFT - PERFORMS OVERFLOW OR UNDERFLOW SHARING TO THE LEFT./
OVERRIGHT - PERFORMS OVERFLOW OR UNDERFLOW SHARING TO THE RIGHT.
COMBINE - MERGES TWO KEYS FOR UNDERFLOW ON DELETION.

VARIABLES:

ACTION - INPUT PARAMETER THAT TELLS WHETHER TO INSERT OR DELETE.
CURRENT - NUMBER OF THE NODE "CUR"
DEBUG - DEBUGGING OUTPUT FLAG
DOWNPTR - POINTER THAT POINTS TO THE NODE BELOW "CUR" IF ON AN UPPER LEVEL, AND CONTAINS "KEYPOS" IF ON THE BOTTOM LEVEL.
HEAD - HEAD OF THE AVAILABLE LIST OF NODES.
HORZ_PTR - TEMPORARY VARIABLE TO HOLD THE HORIZONTAL POINTER OF THE BOTTOM LEVEL.
I, J, K - TEMPORARY VARIABLES
KEY - INPUT PARAMETER, KEY TO BE INSERTED OR DELETED.
KEYLENGTH - INPUT PARAMETER, LENGTH OF KEY IN THE TREE.
KEYPOS - INPUT PARAMETER, POINTER TO BE ASSOCIATED WITH THE KEY AT THE BOTTOM LEVEL OF THE TREE.
LEV - LEVEL OF THE NODE "CUR" IN THE TREE.
MAX_BRANCH - INPUT PARAMETER, MAXIMUM BRANCHING FACTOR FOR THE TREE (ORDER).
MIN_KEY - MINIMUM NUMBER OF KEYS THAT MAY BE IN A NODE.
PARENT - ARRAY OF THE NUMBERS OF ALL THE PARENT NODES USED IN SEARCHING FOR THE CURRENT KEY.
PARPOS - ARRAY OF POINTERS FOLLOWED IN THE PARENT NODES USED IN SEARCHING FOR THE CURRENT KEY.
POS - POSITION OF THE KEY IN THE NODE "CUR".
RESULT - OUTPUT RESULT CODE.
ROOT - INPUT PARAMETER, ROOT NODE NUMBER OF THE TREE.
SIBLING - NUMBER OF THE NODE "SIB".
STAT - COUNTERS FOR ACTIONS IN THE TREE.

```
*/

DCL
KEY  CHAR(*),
(KEYPOS, NEXT)  FIXED BIN (31,0),
(KEYLENGTH, ROOT, MAX_BRANCH, ACTION, RESULT, STAT(*))
  FIXED BIN,

1 CUR,
  2 CURNOKEYS  FIXED BIN  INIT (0),
  2 CURKEY(MAX_BRANCH)  CHAR(KEYLENGTH)  INIT ((MAX_BRANCH) (' ')),
  2 CURPTR(MAX_BRANCH+1)  FIXED BIN (31,0)
      INIT ((MAX_BRANCH + 1) 0),

1 SIB,
  2 SIBNOKEYS  FIXED BIN,
  2 SIBKEY(MAX_BRANCH)  CHAR(KEYLENGTH)
      INIT ((MAX_BRANCH) (' ')),
  2 SIBPTR(MAX_BRANCH+1)  FIXED BIN (31,0)
      INIT ((MAX_BRANCH + 1) 0),

1 PAR,
  2 PARNOKEYS  FIXED BIN,
  2 PARKEY(MAX_BRANCH)  CHAR(KEYLENGTH)
      INIT ((MAX_BRANCH) (' ')),
  2 PARPTR(MAX_BRANCH+1)  FIXED BIN (31,0)
      INIT ((MAX_BRANCH + 1) 0),

DEBUG FIXED BIN INIT (0),
FLOOR  BUILTIN,
(I, J, LEV, MIN_KEY, PARENT(50), PARPOS(50), POS, CURRENT,
SIBLING)  FIXED BIN,

INDEXIO EXTERNAL ENTRY (FIXED BIN, 1, 2 FIXED BIN,
  2 (*) CHAR (*), 2 (*) FIXED BIN (31,0), FIXED BIN,
  FIXED BIN, FIXED BIN, (*) FIXED BIN);


RESULT = 0;
MIN_KEY = FLOOR((MAX_BRANCH-1) / 2);

/*  ACTION = 1 IS FOR INSERT, ACTION = 2 IS FOR DELETE  */
IF ACTION = 1 THEN CALL INSERT;
ELSE IF ACTION = 2 THEN CALL DEL;

RETURN;

SEARCH: PROC (KEY, POS, SUCCESS);
  /*
  THIS PROCEDURE SEARCHES THE B+TREE FOR "KEY".  IF IT IS FOUND,
SUCCESS IS SET TO ONE, OTHERWISE ZERO.  "POS" IS THE POSITION OF
THE KEY IN THE NODE, IF IT IS FOUND.  IF IT IS NOT FOUND, "POS"
IS WHERE IT BELONGS.
```

```
  GLOBAL VARIABLES:
  LEV, ROOT, PARENT, PARPOS, PAR, CUR, DEBUG.

  PROCEDURES CALLED:  GETNODE
  */
  DCL
  KEY CHAR(*),
  (POS, SUCCESS, LWB, UPB)  FIXED BIN;


  IF DEBUG = 1 THEN PUT SKIP LIST('SEARCH', KEY);
  CURRENT, LEV = 0;
  POS = 1;
  NEXT = ROOT;
  DO WHILE (NEXT > 0);
    LEV = LEV + 1;
    PARENT(LEV) = CURRENT;
    PARPOS(LEV) = POS;
    IF LEV > 1 THEN PAR = CUR;
    CURRENT = NEXT;
    CALL GTNODE (CUR, CURRENT);

    /*  FIND THE KEY IN THE NODE  */
    LWB = 1;
    UPB = CURNOKEYS;
    DO WHILE (LWB <= UPB);
      POS = (LWB + UPB) / 2;
      IF KEY < CURKEY(POS) THEN UPB = POS - 1;
      ELSE IF KEY > CURKEY(POS) THEN LWB = POS + 1;
      ELSE GO TO OUT;
      END;
    POS = LWB;
OUT:;

    NEXT = CURPTR(POS);
    END;

  SUCCESS = 0;
  IF CURRENT > 0 THEN IF POS <= CURNOKEYS THEN
    IF KEY = CURKEY(POS) THEN SUCCESS = 1;
  IF DEBUG = 1 THEN PUT SKIP LIST(SUCCESS, POS, LEV);
  RETURN;
  END;  /*  SEARCH  */

INSERT: PROC;

/*
   THIS PROCEDURE INSERTS A KEY, "KEY", INTO THE TREE.
THERE ARE SEVERAL GLOBAL VARIABLES.
  */

  DCL
  (J, SUCCESS)  FIXED BIN,
  DOWNPTR  FIXED BIN (31,0);
```

```
     IF DEBUG = 1 THEN PUT SKIP LIST('INSERT', KEY);
     DOWNPTR = -KEYPOS;

     /*  FIND THE KEY POSITION  */
     CALL SEARCH (KEY, POS, SUCCESS);
     IF SUCCESS = 1 THEN DO;
       RESULT = 1;
       RETURN;
       END;

     /*  INITIALIZE ROOT NODE FOR A NEW TREE  */
     IF LEV = 0 THEN DO;
       LEV = 1;
       CALL FETCH (CURRENT);
       ROOT = CURRENT;
       CURNOKEYS = 0;
       CURPTR = 0;
       CURKEY = ' ';
       END;


   LOOP:;
     /*  INSERT THE KEY INTO CURRENT NODE AT POS  */
     IF DEBUG = 1 THEN PUT SKIP LIST('LOOP', LEV);
     CURNOKEYS = CURNOKEYS + 1;
     DO J = CURNOKEYS TO POS + 1 BY -1;
       CURPTR(J+1) = CURPTR(J);
       CURKEY(J) = CURKEY(J-1);
       END;
     CURKEY(POS) = KEY;
     CURPTR(POS+1) = DOWNPTR;

     /*  STORE THE NODE AND RETURN IF IT IS NOT OVERFULL  */
     IF CURNOKEYS < MAX_BRANCH THEN DO;
       CALL PTNODE (CUR, CURRENT);
       IF DEBUG = 1 THEN PUT SKIP LIST ('NO REBALANCING');
       RETURN;
       END;

     /*  IF AT THE TOP, THEN MAKE A NEW ROOT  */
     IF PARENT(LEV) = 0 THEN DO;
       IF DEBUG = 1 THEN PUT SKIP LIST('NEW ROOT');
       CALL FETCH (SIBLING);
       CALL FETCH (PARENT(LEV));
       PARKEY(1) = CURKEY(MIN_KEY + 1);
       IF CUPPTR(2) > 0 THEN CURNOKEYS = MIN_KEY;
       ELSE CURNOKEYS = MIN_KEY + 1;
       I = 1;

       /*  MOVE THE KEYS, POINTERS DOWN  */
       DO J = MIN_KEY + 2 TO MAX_BRANCH;
         SIBKEY(I) = CURKEY(J);
         SIBPTR(I) = CURPTR(J);
         I = I + 1;
```

```
    END;

  SIBPTR(I) = CURPTR(MAX_BRANCH+1);
  SIBNOKEYS = I - 1;
  PARNOKEYS = 1;
  PARPTR(1) = CURRENT;
  PARPTR(2) = SIBLING;

  /*  IF NODE IS A LEAVE THEN SET HORIZONTAL POINTERS  */
  IF CURPTR(2) <= 0 THEN DO;
    SIBPTR(1) = CURPTR(1);
    CURPTR(1) = -SIBLING;
    END;

/*  INCREMENT SPLIT COUNTER  */
STAT(3) = STAT(3) + 1;

  /*  STORE THE NODES  */
  CALL PTNODE (SIB, SIBLING);
  CALL PTNODE (PAR, PARENT(LEV));
  CALL PTNODE (CUR, CURRENT);
  ROOT = PARENT(LEV);
  RETURN;
  END;  /*  NEWROOT  */


/*  LEFT SIDE  */
IF PARPOS(LEV) > 1 THEN DO;
  IF DEBUG = 1 THEN PUT SKIP LIST ('LEFT SIDE');
  SIBLING = PARPTR(PARPOS(LEV)-1);
  CALL GTNODE (SIB, SIBLING);
  IF SIBNOKEYS < MAX_BRANCH - 1 THEN DO;

    /*  SHARE ON LEFT  */
    IF DEBUG = 1 THEN PUT SKIP LIST ('SHARE LEFT');
    CALL OVERLEFT (PAR, SIB, CUR, PARPOS(LEV) - 1);

    /*  INCREMENT OVERFLOW SHARE COUNTER  */
    STAT(4) = STAT(4) + 1;

    /*  STORE THE NODES  */
    CALL PTNODE (SIB, SIBLING);
    CALL PTNODE (PAR, PARENT(LEV));
    CALL PTNODE (CUR, CURRENT);

    RETURN;
    END; /*  SHARE LEFT  */
  END;

/*  RIGHT SIDE  */
IF PARPOS(LEV) <= PARNOKEYS THEN DO;
IF DEBUG = 1 THEN PUT SKIP LIST('RIGHT SIDE');
  SIBLING = PARPTR(PARPOS(LEV)+1);
  CALL GTNODE (SIB, SIBLING);
```

```
  IF SIBNOKEYS < MAX_BRANCH - 1 THEN DO;

    /*  SHARE ON RIGHT  */
    IF DEBUG = 1 THEN PUT SKIP LIST ('SHARE RIGHT');
    CALL OVERRIGHT (PAR, CUR, SIB, PARPOS(LEV));

    /*  INCREMENT THE OVERFLOW SHARE COUNTER  */
    STAT(4) = STAT(4) + 1;

    /*  STORE THE NODES  */
    CALL PTNODE (SIB, SIBLING);
    CALL PTNODE (PAR, PARENT(LEV));
    CALL PTNODE (CUR, CURRENT);

    RETURN;
    END;  /*  SHARING RIGHT  */
  END;

/*  SPLIT  */
/* PUT UPPER KEYS, PTRS INTO SIB, SPLIT CUR  */
IF DEBUG = 1 THEN PUT SKIP LIST('SPLIT');
CALL FETCH (SIBLING);
KEY = CURKEY(MIN_KEY + 1);
DOWNPTR = SIBLING;
IF CURPTR(2) > 0 THEN CURNOKEYS = MIN_KEY;
ELSE CURNOKEYS = MIN_KEY + 1;
I = 1;

/*  MOVE THE KEYS, POINTERS OVER  */
DO J = MIN_KEY + 2 TO MAX_BRANCH;
  SIBKEY(I) = CURKEY(J);
  SIBPTR(I) = CURPTR(J);
  I = I + 1;
  END;
SIBPTR(I) = CURPTR(MAX_BRANCH + 1);
SIBNOKEYS = I - 1;

/*  IF NODE IS A LEAVE THEN SET HORIZONTAL POINTERS  */
IF CURPTR(2) <= 0 THEN DO;
  SIBPTR(1) = CURPTR(1);
  CURPTR(1) = -SIBLING;
  END;

/*  INCREMENT THE SPLIT COUNTER  */
STAT(3) = STAT(3) + 1;


/*  STORE THE NODES  */
CALL PTNODE (CUR, CURRENT);
CALL PTNODE (SIB, SIBLING);

/*  GET READY AND GO BACK FOR INSERTION INTO THE PARENT  */
POS = PARPOS(LEV);
CUR = PAR;
```

```
  CURRENT = PARENT(LEV);
  LEV = LEV - 1;
  IF PARENT(LEV) > 0 THEN
    CALL GTNODE (PAR, PARENT(LEV));
  GO TO LOOP;

  END;  /*  INSERT  */
DEL: PROC;

  /*  THIS PROCEDURE DELETES A KEY FROM THE TREE  */

  DCL (J, SUCCESS) FIXED BIN;

  IF DEBUG = 1 THEN PUT SKIP LIST('DELETE', KEY);

  /*  FIND THE KEY  */
  CALL SEARCH(KEY, POS, SUCCESS);
  IF SUCCESS = 0 THEN DO;
    RESULT = 2;
    RETURN;
    END;

  /*  MAKE SURE CUR IS A LEAVE  */
  IF CURPTR(2) > 0 THEN DO;
    PUT SKIP LIST ('ERROR IN DELETE - NOT AT BOTTOM OF TREE');
    PUT SKIP DATA (CURRENT, CUR, PARENT, PAR, KEY);
    RETURN;
    END;


LOOP:;

  /*  DELETE THE KEY FROM THE NODE 'CUR'  */
  IF DEBUG = 1 THEN PUT SKIP LIST('LOOP', LEV);
  DO J = POS + 1 TO CURNOKEYS;
    CURKEY(J - 1) = CURKEY(J);
    CURPTR(J) = CURPTR(J + 1);
    END;
  CURNOKEYS = CURNOKEYS - 1;

  IF LEV = 1 & CURNOKEYS = 0 THEN DO;
  /*  NEW ROOT  */
    ROOT = CURPTR(1);
    CALL RELEASE (CURRENT);
    RETURN;
    END;

  /*  IF NOT UNDERFULL THEN STORE THE NODE AND RETURN  */
  IF LEV = 1 | CURNOKEYS >= MIN_KEY THEN DO;
    CALL PTNODE (CUR, CURRENT);
    RETURN;
    END;
```

```
/*  LEFT SIDE  */
IF DEBUG = 1 THEN PUT SKIP LIST('LEFT SIDE');
IF PARPOS(LEV) > 1 THEN DO;
  SIBLING = PARPTR(PARPOS(LEV) - 1);
  CALL GTNODE (SIB, SIBLING);
  IF SIBNOKEYS > MIN_KEY THEN DO;

    /*  SHARE FROM LEFT  */
    IF DEBUG = 1 THEN PUT SKIP LIST ('SHARE LEFT');
    CALL OVERRIGHT (PAR, SIB, CUR, PARPOS(LEV)-1);

    /*  INCREMENT THE UNDERFLOW SHARE COUNTER  */
    STAT(5) = STAT(5) + 1;

    /*  STORE THE NODES  */
    CALL PTNODE (SIB, SIBLING);
    CALL PTNODE (PAR, PARENT(LEV));
    CALL PTNODE (CUR, CURRENT);
    RETURN;
    END;  /*  SHARING LEFT  */

  ELSE DO;
    /*  COMBINE ON LEFT  */
    IF DEBUG = 1 THEN PUT SKIP LIST ('COMBINE LEFT');
    CALL RELEASE (CURRENT);
    CALL COMBINE (PAR, SIB, CUR, PARPOS(LEV)-1);

    /*  INCREMENT NODE COMBINING COUNTER  */
    STAT(6) = STAT(6) + 1;

    /*  STORE THE NODE  */
    CALL PTNODE (SIB, SIBLING);

    /*  GET READY AND GO BACK TO DELETE FROM PARENT  */
    CURRENT = PARENT(LEV);
    POS = PARPOS(LEV) - 1;
    CUR = PAR;
    LEV = LEV - 1;
    IF LEV > 1 THEN
      CALL GTNODE (PAR, PARENT(LEV));
    GO TO LOOP;
    END;  /*  COMBINING LEFT  */
  END;  /*  LEFT SIDE  */

/*  RIGHT SIDE  */
IF DEBUG = 1 THEN PUT SKIP LIST('RIGHT SIDE');
SIBLING = PARPTR(PARPOS(LEV) + 1);
CALL GTNODE (SIB, SIBLING);
IF SIBNOKEYS > MIN_KEY THEN DO;

  /*  SHARE FROM RIGHT  */
  IF DEBUG = 1 THEN PUT SKIP LIST ('SHARE RIGHT');
  CALL OVERLEFT(PAR, CUR, SIB, PARPOS(LEV));
```

```
      /*  INCREMENT UNDERFLOW SHARE COUNTER  */
      STAT(5) = STAT(5) + 1;

      /*  STORE THE NODES  */
      CALL PTNODE (SIB, SIBLING);
      CALL PTNODE (PAR, PARENT(LEV));
      CALL PTNODE (CUR, CURRENT);

      RETURN;
      END;  /*  SHARING RIGHT  */
    ELSE DO;

      /*  COMBINE RIGHT  */
      IF DEBUG = 1 THEN PUT SKIP LIST ('COMBINE ON RIGHT');
      CALL RELEASE (SIBLING);
      CALL COMBINE (PAR, CUR, SIB, PARPOS(LEV));

      /*  INCREMENT NODE COMBINING COUNTER  */
      STAT(6) = STAT(6) + 1;

      /*  STORE THE NODE  */
      CALL PTNODE (CUR, CURRENT);

      /*  GET READY AND GO BACK TO DELETE FROM PARENT  */
      CURRENT = PARENT(LEV);
      CUR = PAR;
      POS = PARPOS(LEV);
      LEV = LEV - 1;
      IF LEV > 1 THEN
        CALL GTNODE (PAR, PARENT(LEV));
      GO TO LOOP;
      END;  /*  COMBINING RIGHT  */

    END;  /*  DELETE  */


OVERLEFT: PROC (PARENT, LEFT, RIGHT, POS);

 /*
   THIS PROCEDURE PERFORMS OVERFLOW OR UNDERFLOW SHARING
ON TWO NODES OF THE TREE.  THE SHARING GOES FROM RIGHT TO
LEFT.  "LEFT" IS THE LEFT SIBLING NODE, AND "RIGHT" IS THE
RIGHT SIBLING.  SHARING IS DONE UNTIL THERE IS AN EQUAL
(OR NEARLY EQUAL) NUMBER OF KEYS IN EACH SIBLING. "POS"
IS THE POSITION OF THE KEY IN THE PARENT NODE THAT
DIVIDES THE TWO SIBLINGS.

  INTERNAL VARIABLES:

      NLEFT - NUMBER OF KEYS TO END UP IN THE LEFT SIBLING
      NRIGHT - NUMBER OF KEYS TO END UP IN THE RIGHT SIBLING
      J, K  - TEMPORARY VARIABLES

  */
```

```
DCL
1 PARENT,
  2 PARNOKEYS  FIXED BIN,
  2 PARKEY(*) CHAR(*),
  2 PARPTR(*) FIXED BIN(31,0),

1 LEFT,
  2 LNOKEYS  FIXED BIN,
  2 LKEY(*)  CHAR(*),
  2 LPTR(*) FIXED BIN(31,0),

1 RIGHT,
  2 RNOKEYS  FIXED BIN,
  2 RKEY(*)  CHAR(*),
  2 RPTR(*)  FIXED BIN(31,0),

POS  FIXED BIN,
 (HORZ_PTR, NLEFT, NRIGHT, J, K)  FIXED BIN;


/*  CHECK FOR VALID NODE SIZES  */
J = RNOKEYS + LNOKEYS;
NLEFT = J / 2;
NRIGHT = J - NLEFT;
IF NRIGHT >= RNOKEYS THEN DO;
  PUT SKIP(2) LIST
    ('ERROR IN OVERLEFT - NO SHARING POSSIBLE.');
  PUT SKIP DATA (PARENT, LEFT, RIGHT, POS);
  RETURN;
  END;

/*  IF NODES ARE LEAVES, THEN STORE AWAY THE
    RIGHT HORIZONTAL POINTER  */
IF RPTR(2) <= 0 THEN HORZ_PTR = RPTR(1);

/*  FIX UP FIRST KEY  */
J = LNOKEYS + 1;
IF LPTR(2) > 0 THEN DO;
  /*  IF UPPER LEVEL, THEN MOVE DOWN PARENT KEY  */
  LKEY(J) = PARKEY(POS);
  J = J + 1;
  LPTR(J) = RPTR(1);
  END;

K = 1;
/*  MOVE KEYS & POINTERS FROM RIGHT NODE TO LEFT  */
DO J = J TO NLEFT;
  LKEY(J) = RKEY(K);
  LPTR(J + 1) = RPTR(K + 1);
  K = K + 1;
  END;

/*  STORE NEW PARENT KEY  */
IF LPTR(2) > 0 THEN DO;
```

```
      PARKEY(POS) = RKEY(K);
      K = K + 1;
      END;
    ELSE PARKEY(POS) = LKEY(NLEFT);

    /*  MOVE THE RIGHT NODE'S KEYS DOWN  */
    J = 1;
    DO K = K TO RNOKEYS;
      RKEY(J) = RKEY(K);
      RPTR(J) = RPTR(K);
      J = J + 1;
      END;
    RPTR(J) = RPTR(K);

    /*  IF IT IS A LEAVE, THEN RESTORE THE HORIZONTAL
        THE RIGHT SIBLING  */
    IF RPTR(2) <= 0 THEN RPTR(1) = HORZ_PTR;

    /*  SET THE NUMBER OF KEYS IN THE NODES  */
    LNOKEYS = NLEFT;
    RNOKEYS = NRIGHT;
    END;  /*  OVERLEFT  */

OVERRIGHT: PROC (PARENT, LEFT, RIGHT, POS);

  /*
     THIS PROCEDURE PERFORMS OVERFLOW OR UNDERFLOW SHARING
ON TWO NODES OF THE TREE.  THE SHARING GOES FROM LEFT TO
RIGHT.  "LEFT" IS THE LEFT SIBLING NODE, AND "RIGHT" IS THE
RIGHT SIBLING.  SHARING IS DONE UNTIL THERE IS AN EQUAL
(OR NEARLY EQUAL) NUMBER OF KEYS IN EACH SIBLING. "POS"
IS THE POSITION OF THE KEY IN THE PARENT NODE THAT
DIVIDES THE TWO SIBLINGS.

  INTERNAL VARIABLES:

      NLEFT - NUMBER OF KEYS TO END UP IN THE LEFT SIBLING
      NRIGHT - NUMBER OF KEYS TO END UP IN THE RIGHT SIBLING
      J, K  - TEMPORARY VARIABLES

    */

    DCL
    1 PARENT,
      2 PARNOKEYS  FIXED BIN,
      2 PARKEY(*) CHAR(*),
      2 PARPTR(*) FIXED BIN(31,0),

    1 LEFT,
      2 LNOKEYS  FIXED BIN,
      2 LKEY(*)  CHAR(*),
      2 LPTR(*) FIXED BIN(31,0),
```

```
1 RIGHT,
  2 RNOKEYS  FIXED BIN,
  2 RKEY(*)  CHAR(*),
  2 RPTR(*)  FIXED BIN(31,0),

POS  FIXED BIN,
(HORZ_PTR, NLEFT, NRIGHT, J, K)  FIXED BIN;


/*  SET UP NEW NODE SIZES  */
J = RNOKEYS + LNOKEYS;
NRIGHT = J / 2;
NLEFT = J - NRIGHT;

/*  CHECK FOR VALIDITY OF NODES  */
IF NLEFT >= LNOKEYS THEN DO;
  PUT SKIP(2) LIST
    ('ERROR IN OVERRIGHT - NO SHARING POSSIBLE.');
  PUT SKIP DATA (PARENT, LEFT, RIGHT, POS);
  RETURN;
  END;

/*  MOVES KEYS DOWN TO MAKE ROOM IN RIGHT NODE  */
RPTR(NRIGHT+1) = RPTR(RNOKEYS+1);
K = NRIGHT;
DO J = RNOKEYS TO 1 BY -1;
  RKEY(K) = RKEY(J);
  RPTR(K) = RPTR(J);
  K = K - 1;
END;

/*  MOVE PARENT KEY DOWN  */
IF RPTR(2) > 0 THEN DO;
  RKEY(K) = PARKEY(POS);
  RPTR(K) = LPTR(LNOKEYS + 1);
  K = K - 1;
  END;

/*  IF LEAVE, THEN STORE AWAY HORIZONTAL POINTER  */
ELSE DO;
  HORZ_PTR = RPTR(1);
  RPTR(K+1) = LPTR(LNOKEYS+1);
  END;

/*  TRANSFER KEYS FROM LEFT TO RIGHT  */
IF K > 0 THEN RPTR(K) = LPTR(LNOKEYS+1);
J = LNOKEYS;
DO K = K TO 1 BY -1;
  RKEY(K) = LKEY(J);
  RPTR(K) = LPTR(J);
  J = J - 1;
  END;

/*  STORE NEW PARENT KEY  */
```

```
    IF LPTR(2) > 0 THEN DO;
      PARKEY(POS) = LKEY(J);
      RPTR(1) = LPTR(J + 1);
      END;
    ELSE PARKEY(POS) = LKEY(NLEFT);

    /*  SET NUMBER OF KEYS IN LEFT AND RIGHT NODES  */
    LNOKEYS = NLEFT;
    RNOKEYS = NRIGHT;

    /*  IF LEAVE, THEN RESTORE HORIZONTAL POINTER  */
    IF RPTR(2) <= 0 THEN RPTR(1) = HORZ_PTR;

    END;    /*  OVERRIGHT  */

  COMBINE: PROC (PARENT, LEFT, RIGHT, POS);
    DCL
    1 PARENT,
      2 PARNOKEYS  FIXED BIN,
      2 PARKEY(*) CHAR(*),
      2 PARPTR(*) FIXED BIN(31,0),

    1 LEFT,
      2 LNOKEYS  FIXED BIN,
      2 LKEY(*)  CHAR(*),
      2 LPTR(*) FIXED BIN(31,0),

    1 RIGHT,
      2 RNOKEYS  FIXED BIN,
      2 RKEY(*)  CHAR(*),
      2 RPTR(*)  FIXED BIN(31,0),

    POS  FIXED BIN,
    (J, I)  FIXED BIN;

    /*  CHECK FOR VALID NODE SIZES  */
    IF LNOKEYS + RNOKEYS >= MAX_BRANCH-1 & LPTR(2) > 0 |
      LNOKEYS + RNOKEYS >= MAX_BRANCH THEN DO;
      PUT SKIP LIST
        ('ERROR IN COMBINE - COMBINATION NOT POSSIBLE');
      PUT SKIP DATA (PAR, LEFT, RIGHT, POS);
      RETURN;
      END;

    J = LNOKEYS + 1;
    /*  IF NOT LEAVE THEN MOVE PARENT KEY AND LEFTMOST
        POINTER OF RIGHT  */
    IF LPTR(2) > 0 THEN DO;
      LKEY(LNOKEYS+1) = PARKEY(POS);
      LPTR(LNOKEYS+2) = RPTR(1);
      J = J + 1;
      END;

    /*  IF NODES ARE LEAVES, THEN SET HORIZONTAL POINTERS  */
```

```
  ELSE LPTR(1) = RPTR(1);

  /*  MOVE THE KEYS AND POINTERS OVER  */
  DO I = 1 TO RNOKEYS;
    LKEY(J) = RKEY(I);
    LPTR(J+1) = RPTR(I+1);
    J = J + 1;
    END;

  /*  SET NUMBER OF KEYS IN LEFT  */
  LNOKEYS = J - 1;

  END;   /*   COMBINE   */

PTNODE: PROC (NODE, RECORD#);

  /*  THIS PROCEDURE CALLS INDEXIO TO WRITE
     AN INDEX NODE   */

 DCL
  RECORD#  FIXED BIN,
  1 NODE,
    2 NO_KEYS  FIXED BIN,
    2 KEYS(*)  CHAR(*),
    2 PTRS(*)  FIXED BIN(31,0),

  1 PNODE,
    2 PNO_KEYS  FIXED BIN,
    2 PKEYS(MAX_BRANCH-1)  CHAR(KEYLENGTH),
    2 PPTRS(MAX_BRANCH)  FIXED BIN (31,0);

  IF DEBUG = 1 THEN PUT SKIP LIST ('PTNODE:', NODE);

  /*  ASSIGN NODE TO PARAMETER NODE  */
  PNO_KEYS = NO_KEYS;
  DO J = 1 TO MAX_BRANCH - 1;
    PPTRS(J) = PTRS(J);
    PKEYS(J) = KEYS(J);
    END;
  PPTRS(MAX_BRANCH) = PTRS(MAX_BRANCH);

  CALL INDEXIO (2, PNODE, RECORD#, KEYLENGTH, MAX_BRANCH,
    STAT);

  END;  /*  PNODE  */

GTNODE: PROC (NODE, RECORD#);

  /* THIS PROCEDURE GETS A NODE USING INDEXIO  */

  DCL
  RECORD#  FIXED BIN,

  1 NODE,
```

```
      2 NO_KEYS   FIXED BIN,
      2 KEYS(*)   CHAR(*),
      2 PTRS(*)   FIXED BIN(31,0),

    1 PNODE,
      2 PNO_KEYS   FIXED BIN,
      2 PKEYS(MAX_BRANCH-1)  CHAR(KEYLENGTH),
      2 PPTRS(MAX_BRANCH)  FIXED BIN (31,0);


    CALL INDEXIO (1, PNODE, RECORD#, KEYLENGTH, MAX_BRANCH,


    /*  ASSIGN PARAMETER TO NODE   */
    NO_KEYS = PNO_KEYS;
    DO J = 1 TO MAX_BRANCH - 1;
      PTRS(J) = PPTRS(J);
      KEYS(J) = PKEYS(J);
      END;
    PTRS(MAX_BRANCH) = PPTRS(MAX_BRANCH);

    IF DEBUG = 1 THEN PUT LIST ('GTNODE:', NODE);

    END;  /*  GTNODE   */

  FETCH: PROC (RECORD#);

   /* THIS PROCEDURE USES INDEXIO TO GET AN INDEX NODE FROM
  THE AVAILABLE LIST.  */

    DCL
    RECORD#  FIXED BIN,
    1 PNODE,
      2 NO_KEYS FIXED BIN,
      2 KEYS(MAX_BRANCH-1) CHAR(KEYLENGTH),
      2 PTRS(MAX_BRANCH) FIXED BIN (31,0);

    CALL INDEXIO (3, PNODE, RECORD#, KEYLENGTH, MAX_BRANCH,


    END;  /*  FETCH   */

  RELEASE: PROC (RECORD#);

    /*  THIS PROCEDURE USES INDEXIO TO PLACE AN INDEX NODE
    BACK ONTO THE AVAILABLE LIST   */

    DCL
    RECORD#  FIXED BIN,
    1 PNODE,
      2 NO_KEYS FIXED BIN,
      2 KEYS(MAX_BRANCH-1) CHAR(KEYLENGTH),
      2 PTRS(MAX_BRANCH) FIXED BIN (31,0);
```

```
CALL INDEXIO (4, PNODE, RECORD#, KEYLENGTH, MAX_BRANCH,
  STAT);

END;  /*  RELEASE  */

END;  /*  BTREE  */
```

```
/*                        INDEXIO                          */
INDEXIO: PROC (FUNCTION, NODE, RECORD#, KEYLENGTH, MAX_BRANCH,
  STAT);

  /*

  AUTHOR:  ROBERT WEBSTER
           DEPARTMENT OF COMPUTER SCIENCE
           OKLAHOMA STATE UNIVERSITY
           1979

  THIS PROCEDURE DOES INPUT AND OUTPUT ON INDEX NODES, USING
THE "LEAST RECENTLY USED REPLACEMENT" METHOD.  THE NUMBER OF
NODES KEPT IN MEMORY IS "NO_PAGES".  THE INDEX FILE MUST BE
PREVIOUSLY SETUP WITH A LINKED LIST OF AVAILABLE NODES.  THE
LINKS APPEAR IN "NO_KEYS" OF THE UNUSED RECORDS.  RECORD ZERO
CONTAINS THE HEAD OF THE AVAILABLE LIST.  IT IS NOT USED.
THERE ARE SEVERAL CONSTANTS IN THE DECLARATIONS THAT HAVE THE
SAME VALUE AS "NO_PAGES".  THESE SHOULD BE AT LEAST AS BIG AS
THE VALUE OF NO_PAGES.  ALSO, THE SIZE OF "NODES" AND "TMPNODE"
MUST CORRESPOND TO THE RECORD SIZE OF THE INDEX FILE.

  PARAMETERS:

  FUNCTION - ONE OF THE FIVE FUNCTIONS PERFORMED BY "INDEXIO"
  NODE - INDEX NODE PASSED TO OR FROM "INDEXIO".
  RECORD# - RECORD NUMBER OF THE INDEX NODE.
  KEYLENGTH - LENGTH OF THE KEYS IN "NODE"
  MAX_BRANCH - ORDER OF THE TREE.
  STAT - ARRAY FOR STATISTICS. STAT(5) IS THE NUMBER OF READS,
         AND ARRAY(6) IS THE NUMBER OF WRITES.


THERE ARE FIVE FUNCTIONS IN THIS PROCEDURE:
  GETNODE, PUTNODE, FETCH, FREE, AND DUMPLAST.

  1. GETNODE GETS A RECORD FROM "NODES", IF IT IS THERE, OR
     FROM THE FILE ITSELF, IF THE RECORD IS NOT IN "NODES",
     AND PUTS IT INTO THE INPUT STRUCTURE "NODE".  GETNODE
     IS PERFORMED WHEN THE INPUT PARAMETER "FUNCTION" IS
     EQUAL TO 1.

  2. PUTNODE RECIEVES THE INPUT STRUCTURE "NODE" AND PLACES
     IT INTO "NODES".  IF THE RECORD IS NOT ALREADY PRESENT
     IN "NODES", THEN ANOTHER RECORD IS REPLACED.  PUTNODE
     IS EXECUTED WHEN "FUNCTION" IS TWO.

  3. FETCH GETS A RECORD OFF THE AVAILABLE LIST AND PUTS
     IT INTO "NODES".  THE HEAD OF THE AVAILABLE LIST IS
     UPDATED IN MEMORY.  FETCH IS EXECUTED WHEN "FUNCTION" IS
     THREE.

  4. RELEASE PUTS A RECORD BACK INTO THE LINKED LIST OF
```

```
   AVAILABLE RECORDS.  THE RECORD IS ONLY PLACED INTO
   "NODES", HOWEVER.  WHEN IT IS REPLACED, IT IS WRITTEN
   TO THE FILE.  RELEASE IS EXECUTED WHEN "FUNCTION" IS 4.

5. DUMPLAST WRITE ALL THE NODES CURRENTLY IN MEMORY OUT
   TO THE INDEX FILE, AND UPDATES THE HEADER RECORD. THIS
   IS TO BE USED AT THE END OF THE PROGRAM.  DUMPLAST IS
   EXECUTED WHEN "FUNCTION" IS FIVE.

  */

DCL
NO_PAGES  FIXED BIN STATIC INIT(20),
NODES(20) CHAR(1000) STATIC,
(ADDR, CSTG, SUBSTR)  BUILTIN,
1 NODE CONNECTED,
  2 NO_KEYS  FIXED BIN,
  2 KEYS(*)  CHAR(*),
  2 PTRS(*)  FIXED BIN(31,0),


TMPNODE  CHAR(1000)  BASED (ADDR(NODE.NO_KEYS)),
(FUNCTION,RECORD#,KEYLENGTH,MAX_BRANCH,STAT(*)) FIXED BIN,
(LENGTH,I,J,K,RECNUM(20),IX(20),HEAD)  FIXED BIN STATIC,
(ALTERED(20), DEBUG) BIT(1) STATIC,
TRUE   BIT(1) STATIC  INIT ('1'B),
FALSE  BIT(1) STATIC  INIT ('0'B),
FIRST  BIT(1) STATIC  INIT ('1'B),

BINDEX FILE ENV(REGIONAL(1));

/*  INITIALIZATION  */
DEBUG = FALSE;
IF DEBUG THEN PUT SKIP LIST ('RECORD#:', RECORD#);

IF FIRST THEN DO;

  /*  GET THE HEAD OF THE AVAILABLE LIST  */
  READ FILE (BINDEX) INTO (NODES(1)) KEY (0);
  SUBSTR(TMPNODE,1,2) = SUBSTR(NODES(1),1,2);
  HEAD = NO_KEYS;

  FIRST = FALSE;
  ALTERED = FALSE;
  RECNUM = 0;
  DO J = 1 TO NO_PAGES;
    IX(J) = J;
    END;
  END;  /*  INITIALIZATION  */

SELECT (FUNCTION);

  WHEN (1) CALL GETNODE;
  WHEN (2) CALL PUTNODE;
```

```
      WHEN (3) CALL FETCH;
      WHEN (4) CALL RELEASE;
      WHEN (5) CALL DUMPLAS1;

      OTHERWISE
        PUT EDIT ('INVALID FUNCTION IN INDEXIO: ', FUNCTION)
        ( SKIP(3), A, F(9));

      END;  /*  SELECT  */

    RETURN;

GETNODE: PROC;

   /*
   THIS PROCEDURE GETS AN INDEX NODE SPECIFIED BY "RECORD#".
FIRST, ALL THE NODES IN MEMORY ARE SEARCH.  IF THE REQUESTED
NODE IS IN NOT IN MEMORY, THEN IT IS READ IN, REPLACING THE
LEAST RECENTLY USED NODE.  EITHER WAY, ITS PLACE IN "IX" IS
UPDATED TO REFLECT ITS REFERENCE. "IX" IS A POINTER ARRAY
THAT KEEPS ALL THE NODES IN LOGICAL ORDER OF TIME SINCE LAST
REFERENCE.
  */


  /*  SEARCH FOR THE REQUESTED NODE  */
  DO J = 1 TO NO_PAGES WHILE (RECNUM(IX(J)) ¬= RECORD#);
    END;

  /*  REQUESTED NODE NOT FOUND  */
  IF J > NO_PAGES THEN DO;
    J = NO_PAGES;
    IF ALTERED(IX(J)) THEN DO;
      WRITE FILE (BINDEX) FROM (NODES(IX(J)))
        KEYFROM (RECNUM(IX(J)));
      /*  INCREMENT WRITE COUNTER  */
      STAT(2) = STAT(2) + 1;
      END;
    READ FILE (BINDEX) INTO (NODES(IX(J))) KEY (RECORD#);
    /*  INCREMENT READ COUNTER  */
    STAT(1) = STAT(1) + 1;
    RECNUM(IX(J)) = RECORD#;
    ALTERED(IX(J)) = FALSE;
    END;

  /*  PUT THE NODE AT THE TOP OF THE LIST  */
  I = IX(J);
  DO J = J TO 2 BY -1;
    IX(J) = IX(J-1);
    END;
  IX(1) = I;

  /*  ASSIGN THE PHYSICAL RECORD TO THE INPUT STRUCTURE.
     TMPNODE IS BASED ON "NODE"  */
```

```
  /* CSTG IS A BUILTIN FUNCTION THAT GIVES THE LENGTH OF
     ITS ARGUMENT IN BYTES  */
  LENGTH = CSTG(NODE);
  SUBSTR(TMPNODE, 1, LENGTH) = NODES(IX(1));


  IF DEBUG THEN PUT SKIP LIST ('GETNODE:', RECORD#, NODE);

  RETURN;
  END;  /*  GETNODE  */


PUTNODE: PROC;

   /*
   THIS PROCEDURE PUTS AN INDEX NODE INTO THE LIST OF NODES
IN MEMORY.  IF THE NODE IS ALREADY PRESENT IN MEMORY, THEN
THE NODE AND ITS POSITION IN "IX" ARE UPDATED.  IF THE
NODE IS NOT IN MEMORY, THEN THE LEAST RECENTLY USED NODE IS
REPLACED, WRITING IT THE FILE FIRST IF ITS "ALTERED" FLAG
IS SET.  THE "ALTERED" FLAG ON THE INPUT NODE IS SET.
  */

  IF DEBUG THEN PUT SKIP LIST ('PUTNODE:', RECORD#, NODE);

  /*  FIND THE NODE  */
  DO J = 1 TO NO_PAGES WHILE (RECNUM(IX(J)) ¬= RECORD#);
    END;

  /*  NODE NOT FOUND  */
  IF J > NO_PAGES THEN DO;
    J = NO_PAGES;
    IF ALTERED(IX(J)) THEN DO;
      WRITE FILE (BINDEX) FROM (NODES(IX(J)))
        KEYFROM (RECNUM(IX(J)));
      /*  INCREMENT WRITE COUNTER  */
      STAT(2) = STAT(2) + 1;
      END;
    RECNUM(IX(J)) = RECORD#;
    END;

  /*  MOVE THE OTHER MEMBERS OF THE LIST DOWN  */
  I = IX(J);
  DO J = J TO 2 BY -1;
    IX(J) = IX(J-1);
    END;
  IX(1) = I;
   ALTERED(IX(1)) = TRUE;

  /*  STORE THE INPUT STRUCTURE IN THE RECORD.
     TMPNODE IS BASED ON "NODE"  */
  /* CSTG IS A BUILTIN FUNCTION THAT GIVES THE LENGTH OF
     ITS ARGUMENT IN BYTES  */
  LENGTH = CSTG(NODE);
```

```
  NODES(IX(1)) = SUBSTR(TMPNODE, 1, LENGTH);

  RETURN;
  END;  /*  PUTNODE  */

FETCH: PROC;

  /*  THIS PROCEDURE GETS A NODE FROM THE AVAILABLE LIST
AND ADDS IT TO THE LIST OF NODES IN MEMORY.  */

  IF HEAD <= 0 THEN DO;
    PUT EDIT ('OUT OF NODE SPACE.  HEAD: ', HEAD)
      (SKIP(3), A, F(9));
    STOP;
    END;

  RECORD# = HEAD;
  CALL GETNODE;
  HEAD = NO_KEYS;

  IF DEBUG THEN PUT SKIP LIST ('FETCH:', RECORD#);

  RETURN;
  END;  /*   FETCH  */

RELEASE: PROC;

  /*  THIS PROCEDURE PUTS A NODE BACK ON THE AVAILABLE
LIST USING THE PAGING ROUTINES GETNODE AND PUTNODE  */

  NO_KEYS = HEAD;
  HEAD = RECORD#;
  CALL PUTNODE;

  IF DEBUG THEN PUT SKIP LIST ('RELEASE:', RECORD#);

  RETURN;
  END;  /*  RELEASE  */

DUMPLAST: PROC;

  /*
  THIS PROCEDURE IS USED TO DUMP INDEX NODES IN MEMORY
WHICH HAVE BEEN ALTERED BACK ONTO THE FILE.  IT SHOULD BE
USED AT THE END OF THE PROGRAM.
  */

  DO J = 1 TO NO_PAGES;
    IF ALTERED(J) THEN DO;
      WRITE FILE(BINDEX) FROM (NODES(J)) KEYFROM (RECNUM(J));
      STAT(2) = STAT(2) + 1;
      END;
    END;
```

```
/*  OUTPUT THE HEAD OF THE AVAILABLE LIST  */
NO_KEYS = HEAD;
SUBSTR(NODES(1),1,2) = SUBSTR(TMPNODE,1,2);
WRITE FILE (BINDEX) FROM (NODES(1)) KEYFROM (0);

ALTERED = FALSE;
FIRST = TRUE;

RETURN;
END;  /*  DUMPLAST  */

END;  /*  INDEXIO  */
```

```
/*                            GOFIND                              */
GOFIND: PROC (KEY, ROOT, KEYLENGTH, MAX_BRANCH, RECORD#,
  KEY#, POINTER, FOUND, STAT);

  /*

  AUTHOR:  ROBERT WEBSTER
           DEPARTMENT OF COMPUTER SCIENCE
           OKLAHOMA STATE UNIVERSITY
           1979

    THIS PROCEDURE SEARCHED THE B-TREE STRUCTURE FOR A GIVEN
   KEY.  THE RECORD NUMBER (RECORD#) AND THE POSITION OF THE
   KEY WITHIN THE RECORD (KEY#) ARE RETURNED, IF THE KEY IS
   FOUND.  IF THE KEY IS NOT FOUND, THE RECORD NUMBER AND KEY
   NUMBER ARE RETURNED FOR THE NEXT GREATER KEY IN THE TREE.
   THE FLAG "FOUND" INDICATES A SUCCESSFUL SEARCH WITH A VALUE
   OF '1'B.


 INPUT PARAMETERS:

   KEY - KEY TO BE SEARCHED FOR
   ROOT - ROOT NODE OF THE TREE TO BE SEARCHED
   KEYLENGTH - LENGTH OF KEYS IN THE TREE
   MAX_BRANCH - MAXIMUM BRANCHING FACTOR IN THE TREE

 OUTPUT PARAMETERS:

   RECORD# - RECORD NUMBER OF THE DESIRED (OR NEXT HIGHER)
             KEY.
   KEY# - NUMBER OF THE DESIRED (OR NEXT HIGHER) KEY WITHIN
   STAT - STAT(1) IS A COUNTER FOR THE NUMBER OF NODES READ.

 INTERNAL VARIABLES:

   1 NODE - INDEX NODE
     2 NO_KEYS - NUMBER OF KEYS
     2 KEYS - KEYS IN THE NODE
     2 PTRS - POINTERS IN THE NODE

   LEVEL - CURRENT LEVEL IN THE TREE
   LWB - LOWER BOUND FOR BINARY SEARCH
   NEXT - NEXT NODE TO BE SEARCHED
   UPB - UPPER BOUND FOR BINARY SEARCH

 */


DCL
KEY CHAR(*),
POINTER  FIXED BIN (31,0),
NEXT  FIXED BIN (31,0) STATIC,
(ROOT, KEYLENGTH, MAX_BRANCH, RECORD#, KEY#, STAT(*))
```

```
    FIXED BIN,
  FOUND BIT(*),
  (LEVEL, LWB, UP3)  FIXED BIN STATIC,
  1 NODE,
    2 NO_KEYS  FIXED BIN,
    2 KEYS(MAX_BRANCH-1)  CHAR(KEYLENGTH),
    2 PTRS(MAX_BRANCH)  FIXED BIN (31,0),

  TRUE  BIT(1)  STATIC  INIT('1'B),
  FALSE BIT(1)  STATIC  INIT('0'B),

  INDEXIO EXTERNAL ENTRY (FIXED BIN, 1, 2 FIXED BIN,
    2 (*) CHAR (*), 2 (*) FIXED BIN (31,0), FIXED BIN,
    FIXED BIN, FIXED BIN, (*) FIXED BIN);


  RECORD#, LEVEL = 0;
  KEY# = 1;
  NEXT = ROOT;

  /*  LOOP UNTIL THE BOTTOM LEVEL  */
  DO WHILE (NEXT > 0);
    LEVEL = LEVEL + 1;
    RECORD# = NEXT;
    /*  READ INDEX NODE  */
    CALL INDEXIO (1, NODE, RECORD#, KEYLENGTH, MAX_BRANCH,
      STAT);

    /*  DO BINARY SEARCH IN NODE TO FIND KEY'S POSITION  */
    LWB = 1;
    UPB = NO_KEYS;
    DO WHILE (LWB <= UPB);
      KEY# = (LWB + UPB) / 2;
      IF KEY < KEYS(KEY#) THEN UPB = KEY# - 1;
      ELSE IF KEY > KEYS(KEY#) THEN LWB = KEY# + 1;
      ELSE GO TO OUT;
      END;
    KEY# = LWB;
  OUT:;
    NEXT = PTRS(KEY#);
    END;

  /*  SET THE FOUND FLAG  */
  FOUND = FALSE;
  IF RECORD# > 0 THEN IF KEY# <= NO_KEYS THEN
    THEN IF KEY = KEYS(KEY#) THEN FOUND = TRUE;

  /*  IF THE KEY NUMBER IS TOO BIG, THEN REFER IT TO THE
    NEXT NODE  */
  IF KEY# > NO_KEYS THEN DO;
    KEY# = 1;
    RECORD# = -PTRS(1);
    CALL INDEXIO (1, NODE, RECORD#, KEYLENGTH, MAX_BRANCH,
      STAT);
```

```
    END;

POINTER = -PTRS(KEY# + 1);

RETURN;
END;  /*  GOFIND  */
```

```
/*                            TRAVEL                              */
TRAVEL: PROC (RECORD#, KEY#, POINTER, KEY, KEYLENGTH,
  MAX_BRANCH, STAT, EOF);

  /*

  AUTHOR:  ROBERT WEBSTER
           DEPARTMENT OF COMPUTER SCIENCE
           OKLAHOMA STATE UNIVERSITY
           1979

    THIS PROCEDURE RETURNS THE NEXT POINTER OF A TREE AND
INCREMENTS THE KET# AND POSSIBLY THE RECORD#.  EOF IS SET
AFTER THE RIGHTMOST POINTER HAS BEEN RETURNED AND ANOTHER IS
REQUESTED.

INPUT PARAMETERS:

  RECORD# - CURRENT RECORD NUMBER
  KEY# - NUMBER OF THE KEY WITHIN THE RECORD
  STAT - NODE READ COUNTER IS STAT(1)
  KEYLENGTH - LENGTH OF KEYS IN TREE
  MAX_BRANCH - MAXIMUM BRANCHING FACTOR IN THE TREE

OUTPUT PARAMETERS:

  RECORD# - RECORD NUMBER IS SET WHEN THE LAST KEY OF THE
            RECORD IS PROCESSED.
  KEY# - KEY# IS INCREMENTED BY ONE, OR SET TO ZERO FOR A
         NEW RECORD.
  POINTER - POINTER AT THE BOTTOM OF THE TREE ASSOCIATED
            WITH THE CURRENT KEY.
  KEY - THE KEY SPECIFIED BY KEY#.
  STAT - NODE READ COUNTER IS STAT(1).
  EOF - END OF FILE FLAG.

INTERNAL VARIABLES:

  1 NODE - INDEX NODE
    2 NO_KEYS - NUMBER OF KEYS IN THE NODE
    2 KEYS   KEYS IN THE NODE
    2 PTRS - POINTERS ASSOCIATED WITH THE KEYS.  PTRS(0)
             POINTS TO THE NEXT RECORD.  ALL PTRS ARE NEGATIVE
             OR ZERO.

*/

DCL
(KEYLENGTH, MAX_BRANCH, RECORD#, KEY#, STAT(*)) FIXED BIN,
POINTER  FIXED BIN (31,0),
KEY  CHAR(*),
EOF  BIT(*),

1 NODE,
```

```
    2 NO_KEYS  FIXED BIN,
    2 KEYS(MAX_BRANCH-1)  CHAR(KEYLENGTH),
    2 PTRS(MAX_BRANCH)  FIXED BIN (31,0),

TRUE  BIT(1) STATIC INIT('1'B),
FALSE BIT(1) STATIC INIT('0'B),

INDEXIO EXTERNAL ENTRY (FIXED BIN, 1, 2 FIXED BIN,
    2 (*) CHAR (*), 2 (*) FIXED BIN (31,0), FIXED BIN,
    FIXED BIN, FIXED BIN, (*) FIXED BIN);

/*  CHECK FOR END  */
IF RECORD# <= 0 THEN DO;
    EOF = TRUE;
    RETURN;
    END;
EOF = FALSE;

/*  GET NODE  */
CALL INDEXIO (1, NODE, RECORD#, KEYLENGTH, MAX_BRANCH,
    STAT);

/*  GET KEY AND POINTER FROM NODE  */
KEY = KEYS(KEY#);
POINTER = -PTRS(KEY# + 1);

/*  INCREMENT KEY#  */
KEY# = KEY# + 1;
/*  IF DONE WITH THIS NODE, RESET KEY# AND RECORD#  */
IF KEY# > NO_KEYS THEN DO;
    RECORD# = -PTRS(1);
    KEY# = 1;
    END;

RETURN;
END;  /*  TRAVEL  */
```

```
/*                              RANF                                    */
(NOFIXEDOVERFLOW):
RANF: PROC(NARG) RETURNS (FLOAT BINARY);
/*
THIS PROCEDURE GENERATES PSEUDO-RANDOM NUMBERS, UNIFORMILY
DISTRIBUTED ON (0,1).  THIS VERSION IS FOR THE IBM 360.
J.P. CHANDLER, COMPUTER SCIENCE DEPARTMENT.,
OKLAHOMA STATE UNIVERSITY.

METHOD: COMPOSITE OF THREE MULTIPLICATIVE CONGRUENTIAL
          GENERATORS,
          G. MARSAGLIA AND T. BRAY, COMM. ACM 11 (1968) 757.
IF RANF IS CALLED WITH NARG = 0, THE NEXT RANDOM NUMBER IS
RETURNED.
IF RANF IS CALLED WITH NARG ¬= 0, THE GENERATOR IS
RE-INITIALIZED USING IABS(2*NARG+1) AND THE FIRST RANDOM
NUMBER FROM THE NEW SEQUENCE IS RETURNED.

*/

DCL NARG FIXED BIN (31,0),
    J FIXED BIN (15,0) STATIC,
    (KLM, N(128), NDIV, NR) FIXED BIN (31,0) STATIC,
    (RAN, RDIV) FLOAT BIN STATIC,
    JRAN  FIXED BIN (31,0) BASED(ADDR(RAN)),
    NFIRST BIT(1) STATIC INIT('1'B),
    K FIXED BIN (31,0) STATIC INIT(7654321),
    L FIXED BIN (31,0) STATIC INIT(7654321),
    M FIXED BIN (31,0) STATIC INIT(7654321),
    MK FIXED BIN (31,0) STATIC INIT(282629),
    ML FIXED BIN (31,0) STATIC INIT(34821),
    MM FIXED BIN (31,0) STATIC INIT(65541);

  IF NARG ¬= 0 THEN DO;
    /*  RE-INITIALIZE THE GENERATOR  */
    KLM = ABS(2 * NARG + 1);
    K, L, M = KLM;
    END;
  ELSE IF ¬ NFIRST THEN GO TO SKIP;

  /* INITIALIZE THE ROUTINE  */
  NFIRST = '0'B;
  NDIV = 16777216;
  RDIV = 32768 * 65536;
  /* FILL THE TABLE  */
  DO J = 1 TO 128;
    K = K * MK;
    N(J) = K;
    END;

  /*  COMPUTE THE NEXT RANDOM NUMBER  */
SKIP:;
  L = L * ML;
  J = 1 + ABS(L) / NDIV;
```

```
M = M * MM;
NR = ABS(N(J) + L + M);
RAN = FLOAT(NR) / RDIV;

/*  FIX UP THE LEAST SIGNIFICANT BIT  */
IF J > 64 & RAN < 1 THEN JRAN = JRAN + 1;

/*  REFILL THE PLACE IN THE TABLE  */
K = K * MK;
N(J) = K;
RETURN (ABS(RAN));
END;  /*  RANF  */
```

APPENDIX D

PROCEDURES FOR RELATIONAL DATABASE STORAGE AND ACCESS

This appendix contains psuedo-code, or program design language descriptions for the procedures STORE, DEFINE, and ACCESS presented in chapter IV. These descriptions are not detailed. They are meant to aid the reader in understanding the use of the procedures.

```
STORE: PROC (TUPLE, RELATION, TID, OPERATION);
  /*  "TID" IS A TUPLE IDENTIFIER  */
  GET CATALOG INFORMATION ON RELATION;

  SELECT  OPERATION;

    WHEN INSERT CALL INSERT TUPLE (TUPLE, TID);
    WHEN DELETE CALL DELETE TUPLE (TUPLE, TID);
    WHEN UPDATE CALL UPDATE TUPLE (TUPLE, TID);
    END SELECT;


  INSERT TUPLE: PROC (TUPLE, TID);

    IF CLUSTERING ATTRIBUTE IS NULL THEN DO;
      SEARCH PAGE INDEX FOR PARTIALLY FULL PAGE;
      IF ALL PAGES ARE FULL THEN DO;
        GET PAGE FROM PAGE INDEX;
        UPDATE PAGE INDEX;
        END;
      INSERT TUPLE INTO PAGE;
      IF PAGE BECOMES FULL THEN UPDATE PAGE INDEX;
      END;

    ELSE DO;  /*  CLUSTERED RELATION  */
      EXTRACT ATTR_VALUE FROM TUPLE;
      SEARCH PAGE INDEX FOR FIRST ATTRIBUTE VALUE
        LESS THAN OR EQUAL TO ATTR_VALUE;
      IF PAGE IS NOT FULL THEN INSERT TUPLE INTO PAGE;
      ELSE DO;
        SORT TUPLES IN PAGE ON CLUSTERING ATTRIBUTE;
        PLACE UPPER 1/2 OF THE TUPLES INTO NEW PAGE;
        UPDATE TIDs OF RELOCATED TUPLES IN BINARY LINKS
          AND TUPLE INDEXES;
        UPDATES PAGE INDEX;
        INSERT THE TUPLE INTO THE APPROPRIATE PAGE;
        END;
      END;  /*  CLUSTERED RELATION  */

    /*  INSERT INTO TUPLE INDEXES  */
    DO J = 1 TO NUMBER OF TUPLE INDEXES;
      CALL TUPLE INDEX (TUPLE, TID, RELATION,
        TUPLE INDEX INFO(J), INSERT);
      END;

    /*  DELETE THE BINARY LINKS  */
    DO J = 1 TO NUMBER OF SETS OF BINARY LINKS;
      CALL BINARY LINKS (TUPLE, TID, RELATION,
        BINARY LINK INFO(J), INSERT);
      END;
    END INSERT TUPLE;

  DELETE TUPLE: PROC (TUPLE, TID);
    REMOVE TUPLE FROM PAGE;
```

```
  IF THE PAGE BECOMES EMPTY THEN DO;
    DELETE THE PAGE FROM THE PAGE INDEX;
    PLACE THE PAGE ONTO THE AVAILABLE LIST;
    END;

  DO J = 1 TO NUMBER OF TUPLE INDEXES;
    CALL TUPLE INDEX (TUPLE, TID, RELATION,
      TUPLE INDEX INFO(J), DELETE);
    END;

  DO J = 1 TO NUMBER OF SETS OF BINARY LINKS;
    CALL BINARY LINKS (TUPLE, TID, RELATION,
      BINARY LINK INFO(J), DELETE);
    END;
  END DELETE TUPLE;

UPDATE TUPLE: PROC (TUPLE, TID);
  GET OLD TUPLE FROM PAGE;
  IF CLUSTERING ATTRIBUTE IS NOT NULL AND
    OLD ATTR_VALUE ¬= NEW ATTR_VALUE THEN DO;
    CALL DELETE TUPLE (OLD TUPLE, TID);
    CALL INSERT TUPLE (NEW TUPLE, TID);
    RETURN;
    END;

  REPLACE OLD TUPLE IN PAGE WITH NEW TUPLE;
  DO J = 1 TO NUMBER OF TUPLE INDEXES;
    IF OLD AND NEW ATTR_VALUES FOR INDEXED ATTRIBUTE
      ARE NOT EQUAL THEN DO;
      CALL TUPLE INDEX (OLD TUPLE, TID, RELATION,
        TUPLE INDEX INFO(J), DELETE);
      CALL TUPLE INDEX (NEW TUPLE, TID, RELATION,
        TUPLE INDEX INFO(J), INSERT);
      END;
    END;

  DO J = 1 TO NUMBER OF SETS OF BINARY LINKS;
    IF OLD AND NEW ATTR_VALUES FOR LINKED ATTRIBUTES
      ARE NOT EQUAL THEN DO;
      CALL BINARY LINKS (OLD TUPLE, TID, RELATION,
        BINARY LINK INFO(J), DELETE);
      CALL BINARY LINKS (NEW TUPLE, TID, RELATION,
        BINARY LINK INFO(J), INSERT);
      END;
    END;

  END UPDATE TUPLE;

  END STORAGE;
```

```
DEFINE: PROC (RELATION, RELATION INFO, TUPLE INDEX INPUT,
  BINARY LINK INPUT, OPERATION);

  SELECT OPERATION;

    WHEN DEFINE RELATION DO;
      STORE RELATION INFO IN CATALOG;
      END;
    WHEN DELETE RELATION DO;
      GET CATALOG INFORMATION ON RELATION;
      TRAVERSE PAGE INDEX, DELETING PAGES AND INDEX NODES;
      DO J = 1 TO NUMBER OF TUPLE INDEXES;
        CALL DELETE TUPLE INDEX(TUPLE INDEX(J));
        END;
      DO J = 1 TO NUMBER OF SETS OF BINARY LINKS;
        CALL DELETE BINARY LINKS(BINARY LINK INDEX(J));
        END;
      END DELETE RELATION;
    WHEN DEFINE INDEX DO;
      UPDATE CATALOG INFORMATION ON RELATION;
      IF RELATION IS NOT EMPTY THEN DO;
        DO UNTIL END OF  RELATION IS REACHED;
          CALL NEXT TO GET NEXT TUPLE;
          CALL FETCH TO GET THE TUPLE;
          CALL TUPLE INDEX(TUPLE, TID, RELATION,
            TUPLE INDEX INPUT, INSERT);
          END;
        END;
      END DEFINE INDEX;
    WHEN DEFINE BINARY LINKS DO;
      UPDATE CATALOG INFORMATION ON RELATION;
      IF RELATION IS NOT EMPTY THEN DO;
        DO UNTIL END OF RELATION IS REACHED;
          CALL NEXT TO GET NEXT TUPLE;
          CALL FETCH TO GET THE TUPLE;
          CALL BINARY LINKS (TUPLE, TID, RELATION,
            BINARY LINK INDEX INPUT, INSERT);
          END;
        END;
      END DEFINE BINARY LINKS;
    WHEN DELETE TUPLE INDEX
      CALL DELETE TUPLE INDEX(TUPLE INDEX INPUT);
    WHEN DELETE BINARY LINKS
      CALL DELETE BINARY LINKS (BINARY LINK INDEX INPUT);
    END SELECT;


  DELETE TUPLE INDEX: PROC (INDEX);
    PERFORM POSTORDER TRAVERSAL ON INDEX,
      DELETING INDEX NODES;
    END DELETE TUPLE INDEX;
  DELETE BINARY LINK INDEX: PROC (INDEX);
    PERFORM POSTORDER TRAVERSAL ON "FROM" INDEX,
      DELETING EACH NODE;
```

```
  PERFORM POSTORDER TRAVERSAL ON "TO" INDEX,
    DELETING EACH NODE;
  END DELETE BINARY LINK INDEX;
END DEFINE;
```

```
ACCESS: PROC (TUPLE, TID, ATTRIBUTE, ATTR_VALUE, OPERATOR,
  LINK RELATION, OPERATION);
  /*  ATTR_VALUE IS THE VALUE OF AN INPUT ATTRIBUTE  */
  /*  OPERATOR IS A RELATIONAL OPERATOR TO BE USED FOR
      RESTRICTIONS WITH ATTR_VALUE                   */

  GET CATALOG INFORMATION ON RELATION;
  SELECT OPERATION;

    WHEN FETCH CALL FETCH (TUPLE, TID);
    WHEN RESTRICTION DO;
      CALL RESTRICT (ATTR_VALUE, OPERATOR);
      END RESTRICTION;
    WHEN NEXT TID DO;
      CALL TRAVERSE TO GET NEXT TID ON TUPLE INDEX;
      END;
    WHEN NEXT TUPLE DO;
      CALL NEXT TO GET NEXT TID;
      CALL FETCH TO GET THE TUPLE;
      END NEXT TUPLE;
    WHEN LINK DO;
      /* GET TIDS IN "LINK RELATION" THAT MATCH "TUPLE"   */
      IF THE CORRECT SET OF BINARY LINKS DOES NOT EXIST
        THEN DO;
        CALL FETCH TO GET TUPLE REFERRED TO BY TID;
        EXTRACT ATTR_VALUE FROM TUPLE;
        CALL RESTRICT (ATTR_VALUE, EQUAL);
        END;
      ELSE DO;  /* USE THE BINARY LINK INDEX  */
        CALL SEARCH TO FIND TID IN BINARY LINK INDEX;
        DO UNTIL TIDs DO NOT MATCH OR END OF RELATION;
          ADD TID FROM LINK RELATION TO LIST;
          CALL TRAVERSE TO GET THE NEXT TID;
          END;
        END;
      END LINK;
    END SELECT;


  RESTRICT: PROC (ATTR_VALUE, OPERATOR);
    IF TUPLE INDEX EXISTS ON ATTRIBUTE THE DO;
      CALL SEARCH TO GET A KEY AND TID;
      DO UNTIL KEYS DON'T MATCH OR END OF RELATION;
        ADD TID TO LIST;
        CALL TRAVERSE TO GET THE NEXT KEY AND TID;
        END;
      END;
    END RESTRICT;

  END ACCESS;
```

VITA

Robert Edward Webster

Candidate for the Degree of

Master of Science

Thesis: B^{+}-TREES

Major Field: Computing and Information Sciences

Biographical:

Personal data: Born in Tahlequah, Oklahoma, November 23, 1955, the son of Mr. and Mrs. Don Webster.

Education: Graduated from Pryor High School, Pryor, Oklahoma, in May, 1974; received Bachelor of Science degree in Computing and Information Sciences from Oklahoma State University in December, 1977; completed requirements for the Master of Science degree at Oklahoma State University in December, 1979.

Professional Experience: Graduate assistant, Oklahoma State University, Department of Computing and Information Sciences, Stillwater, Oklahoma, January, 1978 to December, 1979.