6-1989

The Implementation of a Generalized Database Directory

Claudia Helms-Vaccaro

Follow this and additional works at: http://scholarworks.wmich.edu/masters_theses

Part of the Computer Sciences Commons

Recommended Citation

http://scholarworks.wmich.edu/masters_theses/1097

This Masters Thesis-Open Access is brought to you for free and open access by the Graduate College at ScholarWorks at WMU. It has been accepted for inclusion in Master's Theses by an authorized administrator of ScholarWorks at WMU. For more information, please contact maira.bundza@wmich.edu.
THE IMPLEMENTATION OF A GENERALIZED DATABASE DIRECTORY

by

Claudia Helms-Vaccaro

A Thesis
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Master of Science
Department of Computer Science

Western Michigan University
Kalamazoo, Michigan
June 1989
THE IMPLEMENTATION OF A GENERALIZED DATABASE DIRECTORY

Claudia Helms-Vaccaro, M.S.
Western Michigan University, 1989

This paper presents the use of an index made up of a single B+ tree connected at its leaf nodes to a lower set of B+ trees—a multitree. It may be utilized to index dense or non-dense attributes of a file. An extra benefit of this structure is the ability to store a secondary attribute value for use when searching the index, often decreasing accesses and time required to complete the search.

Improved efficiency over current indices, such as conventional B+ trees or inverted files, is evaluated.
ACKNOWLEDGEMENTS

I would like to thank Dr. D. Motzkin for her guidance in the development of this thesis. I would, also, like to thank Dr. Williams and Dr. Boals for their valuable comments on improving the content of this thesis.

I would like to thank Gail Lewis for helping me master the PC, Jan VanOpstall for proofreading my papers, and Cathy Broaddus for all her encouragement.

I would like to thank my husband, Bob, without whom I would never have been able to complete this thesis.

Last of all, I would like to thank my family and friends for all the encouragement and support that they have shown me while I have been working on this.

Claudia Helms-Vaccaro
INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book. These are also available as one exposure on a standard 35mm slide or as a 17" x 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700  800/521-0600

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The implementation of a generalized database directory

Helms-Vaccaro, Claudia Dianne, M.S.
Western Michigan University, 1989
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS ................................ ii
LIST OF TABLES .................................... vi
LIST OF FIGURES ................................... vii

CHAPTER

I. INTRODUCTION ................................... 1
II. REVIEW OF LITERATURE ............................ 3
III. MULTITREE APPLICATIONS ....................... 6
   Example of Use of a Multitree ............ 6
   Formal Definition ............................ 8
   Dense/Non-Dense Attributes ............. 12

IV. OVERVIEW OF THE SEARCH, INSERT AND DELETE ALGORITHMS. .................. 14

Search ........................................ 14
   Search of File Based on Indexed Attribute .......... 14
   Example of Locating all Records With a Given Attribute Value........ 16
   Search for all Values of Secondary Attributes Within a Given Attribute Value .... 16
   Example of a Search for all Values of the Secondary Attribute Within a Given Attribute Value .... 17
   Search of File for a Secondary Attribute Within Attribute Value ........ 18
### Table of Contents—Continued

- Example of Locating all Records With a Given Secondary Attribute Value Within a Given Indexed Attribute Value .......................... 18
- Search in Relation to Insert or Delete ........................................... 19
- Search Algorithm ................................................................. 19
- Insert ................................................................. 22
  - Insertion of Entries With New Attribute Values ...................... 23
  - Insertion for Existing Attribute Values ............................... 25
  - Example of Insert .......................................................... 26
  - Insert and Related Algorithms ........................................... 29
- Delete ................................................................. 34
  - Overview of Delete ...................................................... 35
  - Redistribution ............................................................ 35
  - Combination .............................................................. 36
  - Redistribution Example .................................................. 37
  - Combination Example .................................................... 41
  - Delete and Related Algorithms ......................................... 44
- V. PERFORMANCE EVALUATION ................................................ 49
  - Search Evaluation ......................................................... 50
  - Dense Multitrees .......................................................... 50
Table of Contents—Continued

Non-Dense Multitrees ............... 50
Insert Evaluation ................. 52
Delete Evaluation ............... 53

VI. IMPLEMENTATION ................. 55
Run Instructions ................. 55
Program ........................................ 56
Real World Data ................. 83

VII. CONCLUSION ......................... 86

APPENDICES

A. Real World Example—Dense Multitree .... 87
B. Real World Example—
   Non-Dense Multitree ............... 95
C. Data After Program Runs .......... 104

BIBLIOGRAPHY .............................. 107
LIST OF TABLES

1. Initial Data Used in Test Runs of MULTI .... 83
2. Data After Program Runs ................... 105
LIST OF FIGURES

1. Overview of Multitree .................................. 2
2. Non-Dense Multitree ................................... 7
3. Node Layouts .......................................... 10
4. Dense Multitree ....................................... 13
5. Insertion Into Dense Multitree ....................... 23
6. Record Inserted Into File Prior to Modifying the Index ........................................... 26
7. Insertion of Tuple (10 512) Into Node 5 ............ 27
8. Node 5 Splits ......................................... 27
9. Node 14 Linked to Parent Node 4 ..................... 28
10. Non-leaf Split ....................................... 28
11. Non-dense Multitree to be Used for Deletion ... 38
12. Deletion of Record 7 From File and Tuple (10 102) From Index ................................. 39
13. Redistribution With Right Sibling .................. 40
14. Deletion of Record 5 From File and Tuple (10 340) From Index ................................. 42
15. Combination With Right Sibling ...................... 43
CHAPTER I

INTRODUCTION

This work is an implementation of algorithms and ideas suggested by Dr. D. Motzkin. This paper presents the use of an index made up of a single B+ tree, referred to as the upper tree, connected at its leaf nodes to a lower set of B+ trees, the lower trees. Each leaf node of the upper tree contains pointers to the root nodes of the lower trees. The leaf nodes in the lower trees, in turn, store the addresses of records within the file that is being referenced by the index. This structure shall be referred to as a 'multitree' and is pictured in Figure 1. The leaf nodes of the lower tree may also store a secondary attribute which in some cases may reduce the number of retrievals to the file.

It will be shown that multitrees can be efficiently used as indices for dense, as well as non-dense attributes, that they can serve as generalized directories to any of the database attributes.
Figure 1. Overview of Multitree.
CHAPTER II

REVIEW OF LITERATURE

Numerous articles have been written about the efficiency of B trees (Rosenburg & Synder, 1981; Wright, 1985). Wright provides a number of average performance measures for standard B trees, including number of splits per insertion, formula for the average storage utilization, expected height of the tree and the average search path lengths. Rosenberg showed that a B tree which is space optimal, i.e. one with a minimal number of nodes, is nearly time optimal. A compact tree, with its leaf nodes full, is an example of a space optimal tree. B trees become space pessimal as insertions and deletions are performed and need a compaction process to be performed to maintain their space optimality.

The use of B+ trees as indices for relational databases is discussed in Schkolnick and Tiberio (1985) and Van Gucht and Fischer (1988). Van Gucht and Fischer use the B tree index in his research of multilevel nested relations. Schkolnick and Tiberio studied the cost of updates, insertions and deletions for various methods in which the B+ tree index is accessed. He describes a primary/
secondary index method. Hatzopoulos and Kollias (1985) described B trees with a primary/secondary index method. However, neither of these is quite the same as what is proposed by the multitree.

One of the main deficiencies of conventional B trees is that they have not provided adequate directories for non-dense attributes, attributes which are not unique. Although much has been written about B trees, very little has been documented using B trees as the index structure for files with non-dense attributes.

Thus far, inverted files have been used to provide a more efficient means for indexing such files. However, inverted files have disadvantages in that they are very large, do not have an efficient updating method, and that it is hard to predict the length of the list prior to creating it (Davis, 1974). The length of a non-dense attribute index may change as the data within it changes. A generalized directory for non-dense attributes, based on inverted files which resolves some of these problems, has been described in Motzkin and Williams (1988). However, the update problems have not been resolved.

The problems pointed out by Schkolnick and Tiberio (1985) of using a B tree structure for this purpose, as well as the update problems associated with inverted files,
shall be rectified by the multitree. In addition to the improvement in update time, the multitree has the ability to index two related attributes at the same cost as a single attribute, in the case of a non-dense file.
CHAPTER III

MULTITREE APPLICATIONS

In this section, the application of a multitree as a generalized index for either dense or non-dense attributes will be discussed.

Example of Use of a Multitree

Figure 2 is an example of a multitree. The upper tree is indexed by the attribute values. The lower trees are indexed by the record addresses in the file. In the leaf node of the lower tree, these addresses act as pointers to the records in the file. The values of a secondary attribute are stored in the attribute field of these nodes. The use of a secondary attribute will be discussed in detail in the formal definition.

The file in Figure 2 is based on a manufacturing company which produces a product for many customers. The indexed attribute is product. The product may not be the same price for each customer. Thus, when the product is used as the indexed attribute value, there are multiple records with the same attribute value. This is an example of a multitree used to index a non-dense attribute. The secondary attribute in this index is customer.
Figure 2. Non-dense Multitree, $m = 3$. 
Formal Definition

A multitree is composed of an upper B+ tree linked to a lower set of B+ trees. Since the definition of a multitree builds upon that of a B+ tree, a definition of a B+ tree based on that of [Horowitz, 1976] is provided:

Let \( m = 2k + 1 \), where \( k \) is a positive integer.

Then an \( m \)-way B+ tree is as follows:

(a) the root of a B+ may contain between 1 and \( 2k \) attribute values.

(b) all nodes except the root contain between \( k \) and \( 2k \) attribute values.

(c) non-leaf nodes having \( k \) attribute values have \( k + 1 \) children.

(d) all attribute values in the subtree of the \( i \)-th attribute value are greater than or equal to the \( i \)-th attribute value but they are less than the \( (i-th + 1) \) attribute value.

(e) all leaf nodes are at the same level. Each leaf node has a pointer to its right sibling, with the rightmost having a null pointer.

Furthermore, multitrees must also comply with the following:
(f) the root of each tree contains a pointer to the left most leaf node in the tree and a flag specifying if the attribute being indexed is dense or non-dense.

(g) for dense attributes: each attribute value in a leaf node has a corresponding address pointer which points to the address of the corresponding record in the file. No lower tree exists, in this case.

(h) for non-dense attributes: each attribute value in a leaf node of the upper tree has a corresponding address which points to the root of a lower $B^+$ tree.

(i) lower trees are indexed on the address of the record in the file. In non-leaf nodes, these addresses are stored in the attribute value field. In leaf nodes, they are used as address pointers.

The upper tree will be described in detail, first, and then the lower tree. Layouts of these structures are shown in Figure 3.

The root of the upper tree has the structure $(DF, LP, n, addr0, (A_1, L_1), (A_2, L_2), ..., (A_n, L_n))$. DF is a density flag which denotes if the tree is dense or non-
Figure 3. Node Layouts, Based on Nodes From Figure 2.
dense. LP is a pointer to the left most leaf node of the tree. The rest of the root is comparable in structure to non-root nodes.

Each non-root node in the index has the structure

\((n, \text{addr}_0, (A_1, L_1), (A_2, L_2), \ldots, (A_n, L_n), \text{next}_\text{node})\).

\(n\) is number of \((A_i, L_i)\) tuples in the node. \text{Addr}_0\) is a pointer to the subtree which contains tuples with attribute values less than the attribute value in the first tuple in that node. A tuple is a pair of \((A_i, L_i)\). \(A_i\) contains the attribute values on which the file is being indexed. \(L_i\) of a non-leaf node contains address pointers to the subtree with attribute values greater than or equal to \(A_i\) but less than the next attribute value in that node (i.e. \(A_{i+1}\)). In leaf nodes, it contains the pointer to either the root of the lower tree in a non-dense tree or the actual record in the file in a dense tree. \text{Next}_\text{node}\) is the pointer to the leaf node's right sibling. It is not used on non-leaf nodes.

The lower tree's structure is much the same as the upper. It does vary in that the lower tree does not have a density flag on its root and that the tuple's fields are used differently. The non-leaf nodes index on the values stored in the \(A_i\) field, but the \(A_i\) field contains the address of the record in the file rather than an attribute value.
\( L_i \) is the same as in the upper tree for non-leaf nodes. In leaf nodes, however, it contains the address of the record in the file. As all the \( L_i \)'s in the leaf nodes of a lower tree are addresses of records with the same attribute value, the \( A_i \) field is available for other use. It is used to store a secondary attribute. The secondary attribute is another attribute of the record in address \( L_i \), other than the indexed attribute, that might be useful to have quick access to. Since the \( A_i \) field would be empty otherwise, the cost of the secondary attribute is free. The other fields in a lower tree node are the same as in an upper tree. The lower tree does not exist in a dense multitree.

**Dense/Non-Dense Attributes**

When using the multitree as a directory for dense attributes, it becomes a conventional B+ tree. The upper tree's non-leaf nodes contain the attribute values and its leaf nodes contain pointers to the records in the file. No lower tree is needed since each attribute value corresponds to exactly one address, see Figure 4.

When it is used to index non-dense attributes, the upper tree's non-leaf nodes appear the same as those of a dense index. The leaf nodes, as was explained above, contain pointers to the roots of the lower trees. The
lower trees use the address in the file of the record as the indexing value. Leaf nodes use this address as the pointer and a secondary attribute stored in the attribute's position. This structure is pictured in Figure 2.

In the following sections, multitrees that serve as directories to dense attributes will be referred to as dense trees, while multitrees that serve as directories for non-dense attributes will be referred to as non-dense trees.
CHAPTER IV

OVERVIEW OF THE SEARCH, INSERT AND DELETE ALGORITHMS

The algorithms used to create the multitree structure were based on those in Horowitz and Sahni (1976). Starting with the search algorithm, the three major algorithms: search, insert, and delete, will be described.

Search

There are four types of search that may be used with this type of index. They are: (1) the search of the file on an indexed attribute value, (2) the search for all secondary attribute values within an indexed attribute, (3) the search of the file for a specific secondary attribute value within an indexed attribute value, and (4) the process of searching that is required for an insert or a delete.

Search of File Based on Indexed Attribute

In a dense index, this would be basically the same as a search through a conventional B+ tree. Since the dense multitree is virtually the same as a B+ tree, most further discussion will center on multitrees for non-dense
attributes.

In a non-dense tree, the upper tree would be searched in the same way as a B+ tree until the leaf node containing the indexed attribute value is located. Then, the address pointer would be used to locate the root of the lower tree. The lower tree would be accessed by this root. The root of the lower tree stores a pointer to its left most leaf node. This node would be accessed. Each tuple in this node contains an address in the file of a record with the indexed attribute value and a secondary attribute value. Each of the records are accessed with the pointers stored in the leaf node's tuples. Once all the tuples in this node have been read, the next right node is accessed by the pointer to it stored in the sibling pointer. This process would continue until the right sibling pointer equals zero signifying that no more leaf nodes exist within that attribute value.

The number of retrievals of blocks of data from the file is minimized because the addresses of the records to be retrieved are stored in ascending order in the index. Thus, each block is processed fully before the next block is read and will not need to be retrieved again.
Example of Locating all Records With a Given Attribute Value

To locate all the records with an attribute value of 42 (e.g., to locate all records related to product 42 in the example in Figure 2), the root of the index and the attribute value 42 would be passed to SEARCH. Following the path to the attribute value 42, the leaf node 9 is reached. The address pointer related to the attribute value 42 is 13, which is the root of the desired lower tree. The leaf pointer in node 13 indicates that node 11 is the left most leaf for the lower tree corresponding to the attribute value 42. Node 11 is then accessed. Each tuple in node 11, gives an address of a record in the file with 42 as its attribute value (i.e., record 4). Node 11 has only one tuple so after that record has been accessed and processed, node 12 is accessed because it is the right sibling of node 11. It has two tuples which are processed sequentially (i.e., records 8 and 9, respectively). After these records are retrieved, the right sibling pointer is checked and found to be zero, denoting that all the leaf nodes in this lower tree have been processed.

Search for all Values of Secondary Attributes Within a Given Attribute Value

Searching for all secondary attributes of a given
attribute would proceed as above until the left most leaf node of the lower tree was reached. At this point, the desired data can be accessed from the index. Each tuple is processed as above except that the secondary attribute can be retrieved directly from that tuple. Storing the secondary attributes in the attribute fields saves retrievals to the file to access them in this type of search. Reads to the file to access them are eliminated, in this case.

Example of a Search for all Values of the Secondary Attribute Within a Given Attribute Value

To locate all the values of the secondary attribute within a given indexed attribute, (e.g., to locate all the customers for product 42) in Figure 2, the search would proceed as that in the example of locating all records with a given attribute value until the left most leaf of the lower tree was accessed. At this point, the tuple in node 11 would be processed and the value 076 retrieved. With no more tuples in node 11, processing would proceed to node 12. Here, secondary attribute values 512 and 340 would be accessed. Since there are no more leaf nodes in this subtree, the list of secondary attributes would be complete.
Search of File for a Secondary Attribute Within Attribute Value

In a non-dense tree, the search would proceed as above until the left most leaf node in the lower tree was reached. At this point, the A_i field in each tuple would be checked to see if it is the desired secondary attribute value. If it is, that record in the file would be accessed. If not, then the next tuple would be checked. Accessing sibling leaf nodes would proceed as above. The entire set of lower leaf nodes would need to be checked in case there are multiple records with the desired secondary attribute.

Example of Locating all Records With a Given Secondary Attribute Value Within a Given Indexed Attribute Value

To locate all records with the secondary attribute of 512 within the attribute value of 42 (e.g. all records related to customer 512 and product 42), the search would proceed as described in the example of locating all records with a given attribute value until the lower tree's leaf nodes were reached. At this point, if the secondary attribute of interest were customer 512, then the search through the tuples would be much the same as above. The exception being that the only records in the file that would be accessed would be those with a secondary attribute...
of 512. In this case, node 11 would be processed and found not to contain the desired secondary attribute. Node 12 would be read next. The first tuple does have 512 as a secondary attribute so record 8 would be retrieved. After the completion of any processing on this record, the search would continue to the next tuple, which fails to contain the desired secondary attribute. Since this is the end of the list for attribute 42, the search would end. This type of search would have saved two unnecessary retrievals over the above type of search.

Search in Relation to Insert or Delete

When SEARCH is called in relation to an INSERT or DELETE, both the upper and lower trees are searched in a similar manner to locate the positions where the insertions or deletions will occur. This will be described in detail in the INSERT and DELETE sections.

Search Algorithm

Raddr is the address of the root of the tree, either the upper tree or the lower tree depending on the type of search being conducted. Search_type specifies whether the search is: (1) based on indexed attribute, (2) for all values of a secondary attribute within a given attribute
value, (3) for all records with a specific secondary attribute within a given attribute value, (4) of the upper tree on INSERT or DELETE, or (5) of the lower tree on INSERT or DELETE.

If the search is of type 1 - 4, search beginning at raddr for the attribute value in x. If the search_type is type 5 then search beginning at raddr (root of the lower tree) for the record address stored in x.

Each node has the structure \((n, addr0, (A_1, L_1), \ldots, (A_i, L_i), \ldots, (A_n, L_n), \text{next_node})\) described above. \(P\) is the current node being processed. The \(L\) values in the leaf nodes of the upper tree point to the root of the lower tree. In the lower tree, the \(L\) values in the leaf nodes are the addresses of records in the file. For non-leaf nodes, next_node is not utilized. Next_node in the leaf nodes is used when doing a search through the leaf nodes, it points to the next right leaf node or is equal to zero if it is the end of the list.

\(P\), nodeaddr, found and \(i\) are returned globally. \(P\) is the current node. Nodeaddr is the address of \(P\). Found indicates if \(x\) was found or not. If found is true then \(i\) stores the position in the node in which \(x\) exists, else the position in which it would logically be inserted.
SEARCH(raddr, search_type, x, s_value);
begin;
nodeaddr := raddr;
found := false;
dense_tree := false;
i := 1;
if (nodeaddr = 0) then (* if searching new tree*)
at_leaf := true
else
at_leaf := false;
while (not(at leaf node)) do;(* search to leaf node *)
read node P located at nodeaddr in multitree;
when the root node of the upper tree
(search_type = 1) is read,
set dense_tree to true if P.DF = 'D'
P.A_{p+1} := +infinity;
Let i be such that P.A_i <= x < P.A_{i+1};
save current node in list
(* list of parent nodes needed during INSERT & DELETE *)
if (x < P.A_i) then (* get search addr *)
nodeaddr := P.addr0; (* use addr0 *)
else
nodeaddr := P.I_i;
if (P.addr0 = 0) then
at_leaf := true;
end; (* while loop *)
if parent list not empty, remove current node from list of parent nodes
if (dense_tree) then
do;
if (x = P.A_i) then
  case (search_type) of
    4 : found := true;
    1 : read record from file at address I_i
  end; (* end case *)
  else
    set i to point to next tuple in node
  return;
end; (* end dense tree *)
if ((x = P.A_i & search_type not= 5) or
(x = P.I_i & search_type = 5)) then
  (* attribute found on leaf node *)
  found := true;
  case (search_type) of
    4, 5 : return;
  else (* case else *)
doi;
read lower trees root, get left most leaf pointer;
while (P.next_node <> 0) do;
    read leaf node
    case search_type of
        1 : read records \( L_k \) in file and list
        2 : list all \( A_k \) from tuples in this node
        3 : read records from file when
            \( A_k = s_{value} \) (* desired secondary attribute*)
        end; (* end inner case *)
    end; (* end while *)
    end; (* end else in outer case *)
end; (* end outer case *)
else
    set i to point to next tuple in node
    (* i needed for INSERT *)
end;
(* procedure SEARCH *)

Insert

In a dense tree, insertion is the same as conventional insertion into a B+ tree. Figure 5 shows the result of an insertion into the file and index illustrated in Figure 4. The discussion will concentrate on non-dense trees which will include a brief description of insertion as it pertains to conventional B+ trees.

When a new record is inserted into the file, a corresponding entry must be inserted into the multitree. If the value of the indexed attribute already exists in the file then only the lower tree will be modified by the insertion. When the value of the indexed attribute is a
new value then both the upper and lower tree must be modified. The insertion of a new attribute value will be discussed first.

**Insertion of Entries With New Attribute Values**

To insert an entry into the index, **SEARCH** is first called to locate the leaf node of the upper tree into which the new entry should be inserted. In the case in which the attribute value does not already exist, the values returned by **SEARCH** would signify that the attribute value was not contained in the index. **INSERT** would be called with the address of the leaf node. The tuples in the record would be shifted to the right to make a position for the new attribute to be entered such that the new attribute is
greater than its left neighbor and less than its right.

If the number of tuples in the record is less than the order of the tree, then the insertion into the upper tree is complete. However, if it is greater than or equal to the order of the tree then the node must be split into two. In leaf nodes, all the attribute values appear so the nodes are split into two nodes of \((m - \text{round-up}(m/2))\) and \((\text{round-up}(m/2))\) tuples, respectively. The address of the node to be split acts as the address of the first node and a new address is used for the second node. The new node retains the value of the next_node pointer before the split. The first node's next_node is set to point to the new node, thus maintaining the linked list of leaf nodes.

In non-leaf nodes, addr0 points to the left most subtree of that node. An attribute value is not maintained in non-leaf nodes for addr0 because attributes in its subtree are known to be less than \(A_1\). So when a non-leaf node split occurs, the two resulting nodes contain \((\text{round-up}(m/2) - 1)\) and \((m - \text{round-up}(m/2))\) tuples, respectively. This pointer is not used.

A new tuple is added to the parent node to point to the new node. The parent is then checked to see if it requires splitting. This process continues until either splitting is not required or the root of the tree is split. If the
root is split, then a new root is created which contains pointers to both nodes resulting from the split of the root. The pointer to the leftmost leaf node is maintained in the new root node.

A new node would be created to serve as the root of the lower tree. At this point, the only tuple in this node will be the one pointing to the record which was just inserted into the file. The address of this node would be stored in the address pointer of the tuple entry just created in the upper tree.

**Insertion for Existing Attribute Values**

The process of inserting an entry into the index in which the attribute value already exists in the upper tree is much the same. Of course insertion into the upper tree is not needed, but a similar process is followed for the lower tree. SEARCH would be called to find the attribute value in the leaf of the upper tree. The address of the lower tree would be retrieved from this tuple. SEARCH would be called again with that address. If the address to be inserted into the lower tree does not exist in the lower tree, then the insertion process can continue. If upon insertion into the lower tree, a split propagated up to the root of the lower tree, then after the new root was
created, the address pointer in the upper tree would be altered to reflect the change in the root of the lower tree.

This type of insertion does not apply to dense trees as each attribute value exists, at most, once in the file.

Example of Insert

Putting this in terms of the example, a record with an attribute value of 10 and secondary attribute of 512 shall be inserted into the file at address 12 (Figures 6-10). SEARCH would be called to locate the root of the lower tree for attribute value 10, and again for address 12 in the

```
  1
   0.1.0.(00,.4),

  2
   7.2.(9,.1),

  3
   1.5.(00,15),

  4
   2.6.(111,.0),

  5
   4.0.(111,.0),

  6
   1.5.(00,15),

  7
   0.1.0.(00,.4),

  8
   7.2.(9,.1),

  9
   1.5.(00,15),

 10
   2.6.(111,.0),

 11
   4.0.(111,.0),

 12
   1.5.(00,15),
```

Figure 6. Record Inserted Into File Prior to Modifying the Index, m = 3.
This figure is a subset of Figure 2.
Figure 7. Insertion of Tuple (10,512) Into Node 5.

Figure 8. Node 5 Splits, Creating Node 14.
Figure 9. Node 14 Linked to Parent Node 4

Figure 10. Non-leaf Split, Creating New Root.
lower tree. The node for insertion would be determined to be node 5 and a new tuple 3 created. Upon insertion into the node 5, there would be too many tuples in node 5 since the tree is of order 3. Hence, a split would be necessitated. The node would be split. The entry made into its parent pointing to the new node would force a split in the parent node as well. The address of the new root node would replace the current value in the leaf node of the upper tree.

Insert and Related Algorithms

Split Algorithm

SPLIT is a procedure called by INSERT to handle splitting of nodes. Nodeaddr is the address of the current node being processed. P is the current node being split. New_A is the variable which will be stored in the parent's A field. New_L is the address of the new node created in the split. New_A and new_L are output from SPLIT. Upper_tree defines if the current insert is being conducted on the upper tree or the lower tree. Dense_tree is passed globally from SEARCH and designates whether the multitree is dense or non-dense. m is passed globally and is the order of the tree. P' is the new node created by the split.
Due to the lack of proper character representation, [] is being used to represent the round up characters hence \([m/2]\) means the least integer which is greater than or equal to \(m/2\).

**SPLIT**(nodeaddr, P, new_A, new_L, upper_tree);

begin;
set new_L to next node address in index which is available for insertion
if (leaf_node) then
  do;
    \(P := (m-[m/2], addr_0, (A_1 L_1), \ldots, (A_{[m/2]-1} L_{[m/2]-1}), \text{new}_L)\);
    \(P' := ([m/2]!, 0, (A_{[m/2]} L_{[m/2]}), \ldots, (A_m L_m), \text{next}_node)\);
    if (upper_tree or dense_tree) then
      new_A := P.A_{m-[m/2]+1})
    else
      new_A := P.L_{m-[m/2]+1})
  end;
else (* non-leaf node *)
  do;
    \(P := ([m/2]-1, addr_0, (A_1 L_1), \ldots, (A_{[m/2]-1} L_{[m/2]-1}), 0)\);
    \(P' := ([m/2], L_{[m/2]}, (A_{[m/2]+1} L_{[m/2]+1}), \ldots, (A_m L_m), 0)\);
    new_A := P.A_{m/2};
  end;
write P to nodeaddr, P' to new_L
last_nodeaddr := nodeaddr;
set nodeaddr to address of the parent of P from parent address list
end; (* procedure SPLIT *)

**New-Root-Needed Algorithm**

Last_nodeaddr is the address of the last node that is modified or the root address if no node was split. New_A is the variable to be inserted into the A field of the
tuple. New_L is the address to the subtree related to A (or 0 if inserting into a leaf node) which is to be inserted into the L field. Nodeaddr is the address of the current node. Upon the calling of this routine it is equal to 0. Leaf_node designates if this is an insertion into a leaf node. If it is then this is the creation of a new tree. S_value is the value of the secondary attribute which is needed if this is the creation of a new lower tree. Rootaddr, LP, and dense_tree are passed globally. Rootaddr is the address of the root of the upper tree and is passed from the main program. LP and dense_tree are passed from SEARCH. LP stores the address of the left most leaf node of the tree. Dense_tree designates whether the multitree is dense or non-dense.

```
NEW_ROOT_NEEDED(last_nodeaddr,new_A,new_L,nodeaddr,
                   leaf_node,s_value);

begin;
  set nodeaddr to next node address which is available for insertion in the index
  if (upper_tree) then
    do;
      R := (dense_tree,LP,1,last_nodeaddr,
            (new_A,new_L));
      rootaddr := nodeaddr; (* new upper tree root address *)
      write R at rootaddr in index;
    end;
  else (* lower tree *)
    do;
      if (leaf_node) then
        R := (nodeaddr,1,(s_value,new_A));
      else
        R := (LP,1,last_nodeaddr,(new_A,new_L));
      write R at nodeaddr in index;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
reset L in upper tree for attribute to nodeaddr and write it;
end; (* else *)
end; (* procedure NEW_ROOT_NEEDED *)

Insert Algorithm

x is the value to be inserted into the tree. If the insertion is into the upper tree, x is an attribute value. If the insertion is into the lower tree, x is the address of the record in the file. INSERT is called twice with two different values of search_type for a non-dense tree and once for a dense tree with a single value for search_type. Search_type is '4' on the first call to INSERT for insertion into the upper tree or a dense tree. It is '5' on the second call for a non-dense tree for insertion into the lower tree. S_value is the value of the secondary attribute when the insertion is into a lower tree. Faddr is the address in the file of the record being inserted into the index. Raddr is the address of the root of the tree being inserted into. On the initial call to INSERT, raddr is the root of the upper tree. On the second call, hold_raddr is sent as the value of raddr. Hold_raddr was derived in the initial call to INSERT, it is the address of the lower tree and is passed globally. P, nodeaddr, i, and found are returned globally by SEARCH. P is the node in which the insertion will occur, nodeaddr is the leaf node
address into which x will be inserted, i is the position in the node where the insertion will occur. If found in SEARCH is returned as false, it signifies that x does not exist in the tree. If found is returned as true, INSERT would not be called as x already exists in the index.

```
INSERT(x,search_type,s_value,faddr,raddr);
begin;
    CALL SEARCH(raddr,search_type,x,S_value);
    if (search_type = 4) then
        hold_raddr := nodeaddr; (* used as raddr in second call of INSERT *)
    if (found) then (* x already exists in tree: attribute in upper tree or record in lower tree *)
        if (dense_tree or search_type = 5) then
            writeln ('error - record already exists in index - insert impossible');
            return;
        end;
    case search_type of
        5 : do; (* lower tree *)
            new_A := s_value;
            new_L := faddr;
            upper_tree := false;
        end;
    4 : do; (* upper tree *)
        upper_tree := true;
        if (dense_tree) then
            do;
                new_a := x;
                new_L := faddr;
            end;
        else
            do;
                new_A := x;
                new_L := 0;
            end;
        end; (* end upper tree *)
    else return; (* any other search type*)
    end; (* end case *)

last_nodeaddr := nodeaddr;
```
leaf_node := true;
insert_done := false;
while (nodeaddr <> 0 & not(insert_done)) do;
  (* while node exists and insertion not done *)
  insert (new_A,new_L) into appropriate position
  in P and increment P.n
  if (P.n < m) then  (* node size is within
  limits *)
    do;
      rewrite (index,nodeaddr) from (P);
      insert_done := true;
    end;
  else  (* P must be split, node to large *)
    CALL SPLIT(nodeaddr,P,new_A,new_L,upper_tree);
    if (nodeaddr <> 0) then
      leaf_node := false;
    end;
  if (nodeaddr = 0) then  (* if nodeaddr = 0 *)
    (* if new tree or *)
    CALL NEW_ROOT_NEEDED;  (* if root split *)
  end;
end;

Delete

Of the three main routines, DELETE is the most complicated due to the possibility of needing to redistribute or combine nodes. Since in a B+ tree all attributes are stored in the leaf nodes, deletion need only take place at this level. The attributes, which are also located in non-leaf nodes and act as path directors, only need to be modified to reflect combinations and redistributions. Most deletions in non-dense trees appear in the lower trees, with the upper tree only being modified if the lower tree has been completely eliminated for that attribute value or during tree compression. Deletion from
the upper tree is handled in the same way as that of the lower tree which is described below.

Deletions in dense trees are virtually the same as in a conventional B+ tree and will not be discussed here.

Overview of Delete

As with an insertion, to do a deletion it is first necessary to call SEARCH. If the attribute is not found, then the deletion is not possible. Once the entry to be deleted has been located, DELETE is called with that node's address. The other tuples in the node are shifted to the left to implement the deletion and n is decremented to reflect the deletion.

If n is greater than or equal to \([m/2]\) then the deletion is complete. However, if not, then either the attributes must be redistributed or a combination of records must take place. The sibling of first choice is the right one if it exists else the left may be used. The algorithm will be described using the right sibling.

Redistribution

To implement a redistribution, the right sibling node is read and must have at least \([m/2]\) tuples in its record. The right sibling node is the first choice, but if one does
not exist the left may be used. For a leaf node redistribution, the first tuple of the sibling is transferred to the last entry of the deletion node. The tuples in the sibling are shifted to the left to fill the first tuple's position. The parent node is modified to point to the attribute in the new first tuple in the sibling. If the redistribution were in a non-leaf node, the attribute in the parent's tuple, which corresponds to addr0 in the deletion node's sibling, would be placed as the last attribute entry in the deletion node and the sibling's address pointer (addr0) to its left most child would be the address pointer for that attribute. The first attribute in the sibling would move up to fill the spot left in the parent and its corresponding address pointer would move to the left most address pointer in the sibling. The sibling would be shifted to the left as with a leaf to fill the empty space of the first tuple, the one that was used in the redistribution. When a redistribution takes place, no further redistributions or combinations need to be implemented because the number of nodes does not go below the limit imposed by the B tree definition.

Combination

A combination, on the other hand, may propagate up the
entire height of the tree. A combination takes place when there are not enough tuples in the deletion node and its right sibling, or left if no right sibling exists, to make two separate nodes, hence they must be combined into one. On a non-leaf node, the attribute value in the parent which points to the sibling node, and the left most address pointer of that sibling, combine to form a new tuple at the end of the tuple list in the deletion node. In a leaf node, this part is not necessary because all attribute values are contained in the node. That is the difference between combining leaf and non-leaf nodes. Other than that, the sibling's tuples are just added after the final entry in the deletion node. The parent is modified to delete the tuple which pointed to the sibling. This may necessitate another combination or a redistribution.

Redistribution Example

The multitree illustrated by Figure 11 is of order \( m=5 \). The deletion of the tuple \((10, 102)\) (Figures 12-13) provides an example of redistribution.

After record 7 has been deleted from the file, the corresponding tuple \((10, 102)\) is located in the index in node 5 and deleted from there. The remaining number of tuples in node 5 is one, which is less than \([m/2]\), the
Figure 11. Non-dense Multitree to Be Used for Delete, m = 5.
Figure 12. Deletion of Record 7 From File and Tuple (10 102) From Index.
Figure 13. Redistribution With Right Sibling.
lower limit. Its right neighbor, however, has greater than \( \lceil m/2 \rceil + 1 \) and so can give up a tuple and still be in the allowable range of \( n \). The tuple \((365, 8)\) is added to node 5. The remaining tuples in node 6 are shifted to the left to reflect the removal of the first tuple. The tuple in node 4 which points to node 6 must be modified to reflect that 9 is the lowest address in node 6. That would complete the redistribution processing caused by the deletion.

**Combination Example**

If attribute 340 were deleted from the file at record 5, the corresponding deletion from the index would leave node 3 deficient in tuples (Figures 14-15). Upon checking its right neighbor, it would be found not to contain enough tuples to give one up, so a combination of nodes would occur. The tuples in node 5 would be added to node 3 and \( n \) in node 3 set to reflect the new number of tuples. The next_node pointer in node 3 would be what was contained in node 5's next_node, 6. Node 4 would reflect the change by removing the tuple \((6,5)\) from its list and decrementing its \( n \). Thus, the combination caused by the deletion would be completed.
Figure 14. Deletion of Record 5 From File and Tuple (10, 340) From Index.
Figure 15. Combination With Right Sibling.
Delete and Related Algorithms

Combine Algorithm

COMBINE is called by DELETE to combine two nodes when the current node being processed does not contain enough tuples to stand alone. Leaf_node designates whether the combination is happening at leaf level or not. P, Y, and Z are nodes involved in the combination. P is the node from which the deletion is made. Y is its sibling. Z is the parent of P and Y. Paddr, yaddr, and zaddr are their addresses in the index, respectively. j is the position of the tuple in Z which points to P.

COMBINE(leaf_node,P,Y,Z,paddr,yaddr,zaddr,j);
begin;
  if (j < Z.n) then (* combine with right sibling*)
    do;
      (* P, Z.A_{j+1}, & Y *)
      if not(leaf_node) then
        do;
          P.A_{p.n+1} := Z.A_{j+1};
          P.L_{p.n+1} := Y.addr0;
        end;
      add all of Y's tuples at the end of P node;
      P.n := P.n + Y.n;
      P.next_node := Y.next_node;
      return yaddr to list of available nodes;
    write P from paddr;
    shift Z tuples (A_k, L_k) for k >= j+2 to the left 1 tuple to fill in Y's position;
  end; (* combine right sibling *)
else
  do; (* combine with left sibling *)
    (* combine P, Z.A_{j-1}, & Y *)
    if not(leaf_node) then
      do;
        Y.A_{y.n} := Z.A_{j-1};
        Y.L_{y.n} := P.addr0;
      end;
add all of P's tuples at the end of Y node;
Y.n := Y.n + P.n;
Y.next_node := P.next_node;
return paddr to list of available nodes;
write Y from yaddr;
end;  (* combine left sibling *)
Z.n := Z.n - 1;
P := Z;
paddr := zaddr;
end;  (* procedure COMBINE *)

Redistribute Algorithm

REDISTRIBUTE is called by DELETE to handle the situation when P, the deletion node, does not have enough tuples to stand alone, but its sibling node has more than enough so a tuple can be taken from it and given to P. Leaf_node designates whether the redistribution is happening at leaf level or not. P, Y, and Z are nodes involved in the redistribution. P is the deletion node. Y is P's sibling. Z is the parent of P and Y. Paddr, yaddr, and zaddr are their addresses in the index, respectively. j is the position of the tuple in Z which points to P. Delete_completed is passed to DELETE to signify the deletion is completed.

REDISTRIBUTE(leaf_node,P,Y,Z,paddr,yaddr,zaddr,j,
   delete_completed);
begin;
P.n := P.n + 1;
if (j < Z.n) then
   do;  (* redistribute using right sibling *)
      if (leaf_node) then
         do;
shift \( Y_1 \) tuple to \( P_{p,n} \) tuple;
if (search_type = '4') then
  \( Z.A_{j+1} := Y.A_2; \)  (* upper tree *)
else
  \( Z.A_{j+1} := Y.L_2; \)  (* lower tree *)
end;
else  (* non-leaf *)
do;
P.A_{p,n} := Z.A_{j+1};
P.L_{p,n} := Y.addr0;
Z.A_{j+1} := Y.A_1;
Y.addr0 := Y.L_1;
end;
shift \( Y \) to left 1;
else  (* redistribute using left sibling *)
do;
  if (leaf_node) then
do;
shift \( Y_{Y,n} \) tuple to \( P_1 \) tuple
  \( Z.A_j := Y.L_{Y,n}; \)
end;
else  (* non-leaf *)
do;
P.A_l := Z.A_{j-1};
P.L_l := P.addr0;
Z.A_j := Y.A_{Y,n};
end;
end;
Y.n := Y.n - 1;
rewrite (P to paddr, Z to zaddr, & Y to yaddr);
delete_completed := true;
return;
end;  (* procedure REDISTRIBUTE *)

Delete Algorithm

Raddr is the root of the tree from which the deletion is to be made. Search_type designates whether it is from the upper or a lower tree. \( x \) is the value to be deleted from the tree. If the deletion is in the upper tree it is an attribute value. If the deletion is in a lower tree, \( x \)
is the record address in the file. SEARCH returns nodeaddr, P, i, and found globally to DELETE. Nodeaddr is the address of the node to be deleted from. P is that node. i is the position of the tuple to be deleted within P. The initial time through the loop i is sent from SEARCH. If further processing is necessary, it is then derived by checking the node. fnx designates whether x was found or not. j is the position of the tuple in P's parent (Z) which points to P. DELETE is called twice, once for the upper tree and again for the lower tree (in a non-dense tree).

```pascal
DELETE(raddr,search_type,x);
begin;
  CALL SEARCH(raddr,search_type,x);
  if (not(found) then
    do;
      writeln('record does not exists -
      deletion impossible');
      return;
  end;
  paddr := nodeaddr;
  delete_completed := false;
  shift (A, L) k > i to left one position &
  return L to list of available addresses
  P.n := P.n - 1;
  while ((P.n < ([m/2]-1)) & (paddr <> rootaddr)) do;
    (* P has too few attributes and
    must be merged with a sibling *)
    set Z and zaddr equal to P's parent and
    its address from parent list
    set j equal to the position of the tuple in
    Z which points to P;
    if (j < Z.n) then (* P has right sibling *)
      yaddr := Z.L_i+1;
    else (* P must have a left sibling *)
      yaddr := Z.L_{j-1};
    read P's sibling Y from yaddr;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
if (\(Y.n \geq \lceil m/2 \rceil\)) then (* redistribute attributes *)
    CALL REDISTRIBUTE(leaf_node, P, Y, Z,
                        paddr, yaddr, zaddr, j, delete_completed);
else (* not enough tuples to redistribute *)
    CALL COMBINE(leaf_node, P, Y, Z,
                 paddr, yaddr, zaddr, j);
end; (* end while *)
if (not(delete_completed)) then
    write P from paddr;
end; (* procedure DELETE *)
CHAPTER V

PERFORMANCE EVALUATION

The space requirements for a multitree are similar to those of a conventional B+ tree. The maximum number of nodes for a conventional B+ tree of order $m$ is $(m^H - 1)$, where $H$ is the height of the tree. For a multitree both the upper ($H_U$) and lower trees ($H_L$) heights must be taken into consideration. So the maximum number of nodes in a multitree is $(m^{H_U} - 1) + (m^{H_L} - 1)$. The minimum number of nodes is $2\lceil m/2 \rceil^{H_U} + 2\lceil m/2 \rceil^{H_L}$.

The time requirements depend upon whether the multitree is dense or non-dense. The dense tree time requirements are the same as those of a B+ tree. The non-dense tree's attribute modifications are achieved with the efficiency of a B+ tree, while the search is achieved with slightly better efficiency than that of an inverted file.

Updating a multitree is more efficient than the standard structure used with non-dense files, the inverted file, because you can go directly to the place in the index that is affected by the update.
Search Evaluation

Dense Multitrees

In a dense multitree, the number of disk accesses is equal to $H_U + 1$ on a standard search to locate a specific record. To create a sorted list the number of accesses would be $1 + (\text{number of leaf nodes}) + (\text{number of tuples})$. The root node must be read to locate the left most leaf node. After that, each leaf node is read and each record that has a tuple entry is read, resulting in the above formula.

Non-Dense Multitrees

A non-dense multitree is more complex partially due to there being three possibilities for types of searches.

Search of File Based on Indexed Attribute

The number of disk accesses for this type of search is equal to the height of the upper tree, $H_U$, plus the same formula as a dense tree for its lower tree. Therefore, in order to find an entry in the index, the number of disk accesses required would be $H_U + (1 + \text{number of leaf nodes in lower tree} + \text{number of tuples in lower tree})$, where $H_U$ is the height of the upper tree and the rest of the formula relates to the lower tree.
Search of File for all Values of Secondary Attributes Within a Given Attribute Value

The ability to store the secondary attribute in the attribute field yields a decrease in the number of accesses required with this type of search. The number of accesses required is $HU + (1 + \text{number of leaf nodes in lower tree})$. Thus, the savings related to having the secondary attribute is the total number of tuples in the leaf nodes of the lower tree of the specified attribute value.

Search of File for a Specific Secondary Attribute Within Attribute Value

The number of accesses in this case are $HU + (1 + \text{number of leaf nodes in lower tree} + \text{number of tuples with desired secondary attribute in lower tree})$.

The best case for this type of search would be if the desired secondary attribute were in only one record. In this instance, the accesses required would be $HU + (2 + \text{number of leaf nodes in lower tree})$. The height of the upper tree, the root of the lower tree and all its leaf nodes would still need to be accessed, but there would only be one access made to the file.

The worst case would be if the desired secondary attribute was the same in all the records. This would yield the same number of accesses as in the search of the
file based on the indexed attribute.

Insert Evaluation

To insert an attribute which already has an entry in the upper tree, SEARCH must be called twice resulting in accessing the upper tree HU times and the lower HL times. It has been assumed that there is enough space in memory to store each parent node. When no split is necessary, there would be one additional access to rewrite the updated node, H + 1. If s is the number of levels through which the split is propagated, then the number of accesses as a result of the split is 2s + 1. One access is needed to rewrite each modified node, one to write each newly created node and a final one to write the last parent modified by the split of its children.

The worst case would be if the split went all the way to the root in which case the formula would break down to be 2H + 1 where H is the height of the tree in which the split occurs. An additional access would be needed to rewrite the leaf node in the upper tree due to the change in the root of the lower tree. The total accesses, including those from SEARCH, would be (HU + HL) reads + (2s + 2) writes for a split in the lower tree that required the creation of a new root for the lower tree.
The upper tree is affected by splitting when a new attribute is added to it in which case the formula would change to $HU + (2s + 1) + 2$. The majority of splitting generally occurs in the lower tree.

When an insertion is made into a dense multitree the access requirements are similar to those in a conventional B+ tree. The number of accesses is $HU + (2s + 1)$.

Delete Evaluation

The number of accesses required to complete a simple delete where no redistribution or combining of nodes is necessary is $(HU + HL)$ reads + 1 (write), since both the upper and lower trees must be read to their leaf nodes to accomplish a delete and the lower leaf node rewritten.

When a redistribution must be implemented the number of accesses is $(HU + HL + 1)$ reads + (3) writes. Both trees must be processed to get to the leaf node for the delete. An additional read of the sibling node is needed for the redistribution. Then three writes are necessary for the two nodes in which the redistribution took place plus one for their parent due to the change in its attribute caused by the redistribution.

The number of accesses for a combination is $(HU + HL + c + 1)$ reads and $(c + 2)$ writes, where $c$ is the number of
levels through which the combination is propagated. This requires \( HU + HL + (2c + 3) \) accesses. A sibling must be read at each level that requires a combination and a write at each level affected.
CHAPTER VI

IMPLEMENTATION

Included in this section are the run instructions for the program, a copy of the program and the data used prior to testing.

Run Instructions

The name of the program is MULTI. MULTI provides menus or prompts for each step. To begin there is a menu provided:

A. create
B. search
C. insert
D. delete
E. exit

One would begin by choosing "A" to create the index. At this point the choice of dense or non-dense is given. After the index has been created, any of the other selections may be made.

Choosing "B" to search the index prompts the user for the type of search desired. For a dense tree the chooses given are for a list of the entire file or for a specific attribute. Searching a non-dense tree prompts the user with the chooses of a list of all attributes, a list of
data related to a specific secondary attribute, or a list of all secondary attributes.

Inserting into the file prompts the user for the data needed to insert the record into the pbkfile.

The prompts given on a delete depend on whether the delete is from a dense or non-dense index. If it is from a dense index, then the user is only prompted for the pricebook. If the delete is from a non-dense index, then the user is prompted for the pricebook and the record address.

Error messages are provided for invalid chooses or for chooses which are not possible. An example of a choose which is not possible is to try to delete a record which does not exist.

Program

The following is the program that was used to implement the principles put further by this paper.

```
program multi;
{${v-}
const m : integer = 5;
  max_pp = 10;
  top = 5;
type string10 = string[10];

  keyaddr = record
    key : string10;
    addr : integer
  end;
```
list100 = array [1..100] of integer;

filerec = record
    df : char;
    n : integer;
    addr0 : integer;
    tuple : array [0..top] of keyaddr;
    next_node : integer
end;

saverec = record
    node_ptr : integer;
    df : char;
    n : integer;
    addr0 : integer;
    tuple : array [0..top] of keyaddr;
    next_node : integer
end;

holdrec = record
    hold_i : integer;
    node_ptr : integer;
    df : char;
    n : integer;
    addr0 : integer;
    tuple : array [0..top] of keyaddr;
    next_node : integer
end;

pbkrec = record
    delete_flag : char;
    pbk : packed array [1..2] of char;
    sku : packed array [1..10] of char;
    uom : packed array [1..2] of char;
    price : integer;
    cost : integer
end;

var
    index : file of filerec;
    pbkfile : file of pbkrec;
    x : string10;
    num : real;
    last_write, insert_processed, first_time : boolean;
    i, j, s, save_i, run_type, save_node : integer;
    left_most_leaf, paddr, rootaddr, nodeaddr : integer;
    pbkaddr, delete_addr, subroot, recs_read : integer;
    recs_written, seq_index, seq_pbkfile: integer;
    parent_path : array [1..max_pp] of saverec;
hold_parent: holdrec;
index_avail, pbk_avail: list100;
ans, command, display, hold, density_flag: char;
pbk_rec: pbkrec;
T, P, R, Q, Y, Z: filerec;

procedure display_node(F: filerec);
var s: integer;  { display node to screen }
begin;
  writeln ('(',F.df,' ',*F.n,' , ',F.addrO,' );
  for s := 1 to F.n do
    write(' (',F.tuple[s].key,' ,
          F.tuple[s].addr,' ),');
  writeln(F.next_node,')
  writeln(' ');
end;  { display_node procedure }

procedure zero_out(s: integer; var G: filerec);
begin;  { zero out tuple }
  G.tuple[s].key := '00';
  G.tuple[s].addr := 0;
end;  { zero_out procedure }

procedure read_index(naddr: integer; var F: filerec);
var s: integer;  { read entry in index }
begin;
  for s := 1 to m do  { init F.tuple }
    zero_out(s,F);
  seek(index,naddr);  { read F from index file }
  read(index,F);
  recs_read := recs_read + 1;
  if (display = 'y') then
    begin;
      writeln ('in read: node # ',naddr);
      display_node(F);
    end;
end;  { read_index procedure }

procedure write_index(naddr: integer; F: filerec);
var s: integer;  { write F to index }
begin;
  if (naddr = rootaddr) then
    F.df := density_flag;
  seek(index,naddr);
  write(index,F);
recs_written := recs_written + 1;
if (display = 'y') then
  begin;
    writeln('in write: node # ', naddr);
    display_node(F);
  end;
end;  
  { write_index procedure }

function find_i(G: filerec; search_addr: integer) : integer;
var i : integer;  
  { find position of search_addr in G }
begin;
  find_i := 0;
  if (search_addr = G.addr0) then
    find_i := 0
  else
    begin;
      i := 1;
      while ((i < G.n) and
        (search_addr <> G.tuple[i].addr))  do
        i := i + 1;
      if (i > G.n) then
        writeln('** error: find_i out of range')
      else
        find_i := i;
    end;
end;  
  { find_i procedure }

procedure save_upper_leaf(i: integer; addr:integer;
F : filerec);
var j : integer;  
  { save upper tree's leaf node }
begin;
  hold_parent.node_ptr := addr;  
    { save F's address }
  hold_parent.hold_i := i;  
    { save position }
  hold_parent.n := F.n;  
    { save F }
  hold_parent.addr0 := F.addr0;
  hold_parent.next_node := F.next_node;
  j := 1;
  while (j < m) do
    begin;
      hold_parent.tuple[j].key := F.tuple[j].key;
      hold_parent.tuple[j].addr := F.tuple[j].addr;
    end;
procedure save_parent(paddr: integer);
  var i, j: integer;  
  begin;
    i := 1;
    while ((i <= max_pp) and
      (parent_path[i].node_ptr <> 0)) do
      begin;
        i := i + 1;
        (find next spot in list )
      end;
    if (i <= 10) then
      begin;  
        (save P and P's address )
        parent_path[i].node_ptr := paddr;
        parent_path[i].n := P.n;
        parent_path[i].addr0 := P.addr0;
        parent_path[i].next_node := P.next_node;
        j := 1;
        while (j < m) do
          begin;
            parent_path[i].tuple[j].key := P.tuple[j].key;
            parent_path[i].tuple[j].addr :=
              P.tuple[j].addr;
            j := j + 1;
            end;
      end;
    else  
      (else give message )
      writeln ('parent_path array out of bounds');
  end;  
  { save_parent procedure }

procedure parent(var paddr : integer; var F : filerec);
  var i, j: integer;  
  begin;
    i := 1;
    while ((i <= max_pp) and
      (parent_path[i].node_ptr <> 0)) do
      begin;
        i := i + 1;
        if (i = 1) then
          paddr := 0
        else
          begin;
            i := i - 1;
            paddr := parent_path[i].node_ptr;
            F.n := parent_path[i].n;
          end;
  end;  
  { return parent if exists }
F.addr0 := parent_path[i].addr0;
F.next_node := parent_path[i].next_node;
j := 1;
while (j < m) do
begin;
    F.tuple[j].key := parent_path[i].tuple[j].key;
    F.tuple[j].addr := parent_path[i].tuple[j].addr;
j := j + 1;
end;
parent_path[i].node_ptr := 0;
end;
end; { end of parent }

procedure search(localroot:integer;run_type:integer;
x:string10);
{ search for x in tree with root localroot }
var at_leaf, root_read : boolean;
temp : string10;
begin;
nodeaddr := localroot;
j := 0;
while (j < max_pp) do
begin;
    { initialize parent_path to empty }
    j := j + 1;
    parent_path[j].node_ptr := 0;
end;
j := 0;
if (nodeaddr = 0) then
begin;
    left_most_leaf := 0;
at_leaf := true;
end
else
begin;
    root_read := true;
at_leaf := false;
end;
while (not(at_leaf)) do
begin;
    read_index(nodeaddr,P);
    if (root_read) then
    begin;
        if (run_type = 1) then
            density_flag := P.df;
    end;
end;
left_most_leaf := P.next_node;
root_read := false;
end;
i := 1;
while ((i < P.n) and (x > P.tuple[i].key)) do
begin; { increment i until x > key within P.n }
i := i + 1;
end;
save_parent(nodeaddr); { save P & P's address }
if ((x < P.tuple[i].key) and (i = 1)) then
nodeaddr := P.addrO { get next node address }
else
if (x < P.tuple[i].key) then
nodeaddr := P.tuple[i - 1].addr
else
nodeaddr := P.tuple[i].addr; { match found }
if (P.addrO = 0) then
at_leaf := true
else
at_leaf := false;
end; { end of outer while }
parent(nodeaddr,P);
if (nodeaddr = 0) then { new lower tree }
i := 1
else
if (run_type = 1) then { upper tree leaf }
begin;
if (x = P.tuple[i].key) then { key found }
j := 1
else
if (x > P.tuple[i].key) then
i := i + 1;
end
else { lower tree leaf }
begin;
i := 1;
str(P.tuple[i].addr:10,temp);
while ((i < P.n) and (x > temp)) do
begin;
i := i + 1;
str(P.tuple[i].addr:10,temp);
end;
if (x = temp) then
j := 1
else
if (x > temp) then
i := i + 1;
end;
procedure seq_search(ssaddr: integer);
var i, search_type: integer;
more_nodes, record_found, valid_search: boolean;
reply: char;
begin;
if (density_flag = 'd') then
  search_type := 4
else
  begin;
    writeln('Do you wish to search for:');
    writeln('1. All records in pricebook ', pbk_rec.pbk);
    writeln('2. All secondary attributes in pricebook ');
    writeln(pbk_rec.pbk);
    writeln('3. All records with a specific secondary ');
    writeln('attribute in pricebook ', pbk_rec.pbk);
    writeln('Enter 1, 2, or 3');
    readln(search_type);
    read_index(ssaddr,P) ;
    if (P.next_node <> ssaddr) then
      read_index(P.next_node,P);
    end; { density <> d; non_dense tree search } 
more_nodes := true;
valid_search := true;
case search_type of
1 : begin;
  writeln('data for ',pbk_rec.pbk,' is:');
  writeln('sku uom');
  writeln('price cost');
  while (more_nodes) do
    begin;
      i := 1;
      while (i <= P.n) do
        begin;
          seek (pbkfile,P.tuple[i].addr);
          read (pbkfile,pbk_rec);
          write(pbk_rec.sku,' ',pbk_rec.price);
          write(pbk_rec.uom,' ',pbk_rec.cost);
        i := i + 1;
      end;
    end;
end; { search procedure }
if ((P.next_node = ssaddr) or (P.next_node = 0)) then
  more_nodes := false
else
  read_index(P.next_node,P);
end;

end;  { search_type = 1 }

2 : begin;
  writeln('secondary attributes for
    pbk_rec.pbk,' is/are:');
  while (more_nodes) do
  begin;
    i := 1;
    while (i <= P.n) do
    begin;
      writeln(',P.tuple[i].key);
      i := i + 1;
    end;
    if ((P.next_node = ssaddr) or (P.next_node = 0)) then
      more_nodes := false
    else
      read_index(P.next_node,P);
  end;
end;  { search_type = 2 }

3 : begin;
  writeln('What product do you want?');
  readln(pbk_rec.sku);
  record_found := false;
  writeln('data for
    pbk_rec.pbk, '/pbk_rec.sku, ' is:');
  writeln('uom price cost');
  while (more_nodes) do
  begin;
    i := 1;
    while (i <= P.n) do
    begin;
      if (P.tuple[i].key = pbk_rec.sku) then
      begin;
        record_found := true;
        seek (pbkfile,P.tuple[i].addr);
        read (pbkfile,pbk_rec);
        write(' ',pbk_rec.uom,
          ',pbk_rec.price);
        writeln(' ',pbk_rec.cost);
        i := i + 1;
      end;
    end;
  end;
if ((P.next_node = ssaddr) or (P.next_node = 0)) then
  more_nodes := false
else
  read_index(P.next_node,P);
end;   { end while more_nodes }
if not(record_found) then
  writeln('no records found with secondary attribute',pbk_rec.sku);
end;  { end search_type = 3 }

4 : begin;
  reply := ' '
  while (reply <> 'a') and (reply <> 'b') do
    begin;
      writeln ('which type of search would you like: ');
      writeln ('  a. Search for a specific item or');
      writeln ('  b. Search entire file producing file list');
      readln(reply);
      if (reply <> 'a') and (reply <> 'b') then
        writeln ('reply "a" or "b"');
      end; ( reply <> a or b )
    if (reply = ' b ' ) then
      begin;
        read_index(ssaddr,P);
        if (P.next_node <> ssaddr) then
          read_index(P.next_node,P);
        writeln('data from file is:');
        writeln('pbk sku uom price cost');
        while (more_nodes) do
          begin;
            i := 1;
            while (i <= P.n) do
              begin;
                seek (pbkfile,P.tuple[i].addr);
                read (pbkfile,pbk_rec);
                write(pbk_rec.pbk,' ',
                      pbk_rec.sku,' ',
                      pbk_rec.uom,' ',
                      pbk_rec.price);
                writeln(' ',pbk_rec.cost);
                i := i + 1;
              end;
            if ((P.next_node = ssaddr) or (P.next_node = 0)) then
              more_nodes := false
else
  read_index(P.next_node,P);
end; { reply = 'b'
else { reply = 'a'
begin;
writeln('What pricebook(00-99):');
readln(pbk_rec.pbk);
x := pbk_rec.pbk;
hold_parent.node_ptr := -1;
search(rootaddr,1,x);
if (j = 0) then
  writeln ('pricebook does not exists')
else
  begin;
    writeln ('data for ',
    pbk_rec.pbk,'is:'));
    write ('sku uom ');
    writeln ('price cost');
    seek (pbkfile,P.tuple[i].addr);
    read (pbkfile,pbk_rec);
    write (pbk_rec.sku,' ');
    write (pbk_rec.uom,' ');
    write (pbk_rec.price,' ');
    writeln(pbk_rec.cost);
  end; { j <> 0 }
end; { reply = 'a' }
end; { search_type = 4 }
else
  valid_search := false;
end; { end case }
if ((valid_search) and (record_found)) then
  writeln('all data displayed for search attribute')
else
  if not(valid_search) then
    writeln('invalid search choice');
end; { seq_search procedure }

function next_avail_rec(var avail_lst : list100;
  var seq_avail : integer) : integer;
var i : integer; { gives next avail record number }

begin;
i := 1;
next_avail_rec := 0; { initialize to 0 }
while ((i < 100) and (avail_lst[i] <> 0)) do begin;
i := i + 1;
end;
if (i >= 100) then
  writeln('avail_lst limit exceeded, increase size')
else
  if (i = 1) then
    begin;  { no record numbers out of sequential
      order available so use next sequential
      record number }
      seq_avail := seq_avail + 1;
      next_avail_rec := seq_avail;
    end
  else  { use address from avail_lst }
    begin;
      next_avail_rec := avail_lst[i - 1];
      if (not(last_write)) then
        writeln('using avail node # ',
          avail_lst[i - 1]);
      avail_lst[i - 1] := 0;
    end;
  end;  { next_avail_rec function }

procedure return_avail(return_addr : integer;
  var avail_lst : list100);
var  i : integer;  { returns avail record number to
  avail_lst }
begin;
  writeln('address being returned = # ',return_addr);
  i := 1;
  while ((i < 100) and (avail_lst[i] <> 0)) do
    i := i + 1;
  if (i = 100) then
    writeln('avail_lst exceeded it limit, expand it')
  else
    avail_lst[i] := return_addr;
end;  { return_avail procedure }

procedure insert_into_pbkfile;
  { inserts record into pbkfile }
var acceptable : boolean;
  ans : char;
begin;
acceptable := false;
while (not(acceptable)) do begin;
  writeln('you must now enter data
         for the pbkfile insert');
  writeln(' ');
  writeln('what is the unit of measure');
  readln(pbk_rec.uom);
  writeln(' ');
  write('the next two fields are numeric
         with implied decimals,');  
  writeln(' ');
  writeln('do not input decimals');
  writeln(' ');
  writeln('what is the price?');
  readln(pbk_rec.price);
  writeln(' ');
  writeln('what is the cost?');
  readln(pbk_rec.cost);
  writeln(' ');
  writeln('the entry for the pbkfile is as follows:');
  writeln(' ');
  write('  pbk: ',pbk_rec.pbk);
  writeln(' ');
  write('  sku: ',pbk_rec.sku);
  writeln(' ');
  write('  uom: ',pbk_rec.uom);
  writeln(' ');
  write('  price: ',pbk_rec.price);
  writeln(' ');
  write('  cost: ',pbk_rec.cost);
  writeln(' ');
  writeln('are these values acceptable(y/n)?');
  readln(ans);
  if (ans = 'y') then
    acceptable := true;
end;

 writeln('record inserted into pbk file at address ', pbkaddr);
 seek(pbkfile,pbkaddr);
 write(pbkfile,pbk_rec); 
 insert_processed := true;
end;  ( insert_into_pbk procedure )

procedure split(run_type,localroot:integer;
    var new_root_needed,leaf_node:boolean;
    var last_nodeaddr,new_A: integer;
    var new_L:string10);  
    { processes splits on insert }
var P_prime : filerec;
 s, first_prime : integer;
begin;

if ((paddr = rootaddr) and (run_type = 1)) then
  new_root_needed := true;
for s := 1 to m do { init P_prime.tuple }
  zero_out(s,P_prime);

new_A := next_avail_rec(index_avail,seq_index);
{ get record number for new node created }
P.df := ' ';
P_prime.df := ' ';
if (leaf_node) then { leaf node: set variables }
  begin;
    first_prime := round (m/2 + 0.49);
    P.n := m - first_prime;
    P_prime.n := first_prime;
    P_prime.addr0 := 0;
    if (P.next_node = localroot) then
      P_prime.next_node := 0
    else
      P_prime.next_node := P.next_node;
    P.next_node := new_A;
    if (run_type = 1) then
      new_L := P.tuple[P.n + l].key
    else
      str(P.tuple[P.n + 1].addr:10,new_L);
  end;
else { non leave node: set variables }
  begin;
    first_prime := round (m/2 + 0.49) + 1;
    P.n := first_prime - 2;
    P_prime.n := m - (first_prime - 1);
    P_prime.addr0 := P.tuple[P.n + 1].addr;
    P_prime.next_node := 0;
    new_L := P.tuple[P.n + 1].key;
    P.tuple[P.n + 1].key := '00';
    P.tuple[P.n + 1].addr := 0;
  end;

j := 0;
s := first_prime;
while (s <= m) do { set up P_prime, split node }
  begin;
    j := j + 1;
    P_prime.tuple[j].key := P.tuple[s].key;
    P_prime.tuple[j].addr := P.tuple[s].addr;
    zero_out(s,P);
    s := s + 1;
  end;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
if (run_type = 1) and (leaf_node) then
    if (i > P.n) then
        save_upper_leaf(i-P.n,new_A,P_prime)
    else
        save_upper_leaf(i,nodeaddr,P);
write_index(nodeaddr,P);
write_index(new_A,P_prime); parent(nodeaddr,P); { get parent of P } if (nodeaddr <> 0) then
    last_nodeaddr := nodeaddr;
leaf_node := false;
end; { split procedure }

procedure need_new_root(new_root_needed:boolean;
    last_nodeaddr,new_A,run_type:integer;
    new_L:string10);
{ new root node needed }
begin;
    for s := 1 to m do
        zero_out(s,R);
    R.n := 1;
    R.addr0 := last_nodeaddr;
    R.tuple[1].key := new_L;
    R.tuple[1].addr := new_A;
if (run_type = 1) then
    R.df := density_flag;
nodeaddr := next_avail_rec(index_avail,seq_index);
if (new_root_needed) then
    rootaddr := nodeaddr;
if (left_most_leaf = 0) then { only entry in tree }
    R.next_node := nodeaddr
else
    R.next_node := left_most_leaf;
write_index(nodeaddr,R);
if (run_type = 2) then { prod insert: must change }
begin;
    { addr in upper tree }
    { get hold_parent }
    P.n := hold_parent.n;
    P.addr0 := hold_parent.addr0;
    P.next_node := hold_parent.next_node;
j := 1;
while (j < m) do
    begin;
        P.tuple[j].key := hold_parent.tuple[j].key;
        P.tuple[j].addr := hold_parent.tuple[j].addr;
j := j + 1;
}
end;               { reset addr }
P.tuple[hold_parent.hold_i].addr := nodeaddr;
write_index(hold_parent.node_ptr,P);
end;                { end upper tree addr modification }

end;                { need_new_root procedure }

procedure insert(localroot: integer; run_type : integer);

{ insert record into index }
var  addr, j, l, new_A, paddress : integer;
new_L : string10;
last_nodeaddr : integer;
leaf_node, insert_done,new_root_needed : boolean;

begin;

insert_done := false;   { initialize variables }
leaf_node := true;
nodeaddr := localroot;
last_nodeaddr := localroot;
if (run_type = 2) then
begin;
  new_root_needed := false;
  new_L := pbk_rec.sku;
  new_A := pbkaddr;
end
else
begin;
  new_root_needed := true;
  new_L := x;
  if (density_flag = 'd') then
    new_A := pbkaddr
  else
    new_A := 0;
end;
while ((nodeaddr <> 0) and (not(insert_done))) do
begin;
  { insert into exiting node }
  P.n := P.n + 1;
  l := P.n;
  if ((run_type = 2) and (leaf_node)) then
    begin;
      { shift right until at correct spot }
      { to insert new_L }
      P.tuple[l-1].key := P.tuple[l-1].key;
      P.tuple[l].addr := P.tuple[l-1].addr;
  end
  end
end;
l := l - 1;
end \quad \{ \text{end inner while} \}
else
while ((l > 1) and (P.tuple[l-1].key > x)) do
begin; \{ shift right until at correct spot \}
\quad \{ \text{to insert new_L} \}
P.tuple[l].key := P.tuple[l-1].key;
P.tuple[l].addr := P.tuple[l-1].addr;
l := l - 1;
end; \quad \{ \text{end inner while} \}
P.tuple[l].key := new_L; \quad \{ \text{insert new_L} \}
P.tuple[l].addr := new_A;
end; \quad \{ \text{end while nodeaddr <> 0} \}

if (P.n < m) then \{ P.n within limit of m \}
begin;
\text{new_root_needed := false;}
write_index(nodeaddr,P);
insert_done := true;
end
else \{ P.n exceeds m; must be split \}
split(run_type,localroot,new_root_needed,
leaf_node,last_nodeaddr,new_A,new_L);
end; \quad \{ \text{end insert procedure} \}

procedure redistribute(right_sibling,leaf_node:boolean;
l,run_type,yaddr,zaddr:integer;
var paddr:integer;
var P,Y,Z:filerec;
var delete_completed:boolean);
\{ tuples must be redistributed \}
begin;
if (right_sibling) then \{ P has a right sibling \}
begin;
P.n := P.n + 1;
Y.n := Y.n - 1;
if (leaf_node) then
begin; \{ redistribution between leafnodes \}
P.tuple[P.n].key := Y.tuple[1].key;
P.tuple[P.n].addr := Y.tuple[1].addr;
if (run_type = 1) then
Z.tuple[1].key := Y.tuple[2].key
else
str(Y.tuple[2].addr:10,Z.tuple[1].key);
i := 0;
while (i < Y.n) do
begin;
i := i + 1;
Y.tuple[i].key := Y.tuple[i + 1].key;
Y.tuple[i].addr := Y.tuple[i + 1].addr;
end;
zero_out(Y.n + 1, Y);
end { leafnode delete }
else { non-leafnode delete }
begin;
{ determine what to do }
P.tuple[P.n].key := Z.tuple[1].key;
P.tuple[P.n].addr := Y.addr0;
Z.tuple[1].key := Y.tuple[1].key;
Y.addr0 := Y.tuple[1].addr;
i := 0;
while (i < Y.n) do
begin;
i := i + 1;
Y.tuple[i].key := Y.tuple[i + 1].key;
Y.tuple[i].addr := Y.tuple[i + 1].addr;
end;
end; { non-leafnode deletion }
zero_out(Y.n + 1, Y);
end { Y.n >= m/2 }
else
begin; { P must have a left sibling }
i := 0;
while (i < P.n) do
begin;
i := i + 1;
P.tuple[i + 1].addr := P.tuple[i].addr;
P.tuple[i + 1].key := P.tuple[i].key;
end;
if (leaf_node) then
begin;
P.tuple[1].key := Y.tuple[Y.n].key;
P.tuple[1].addr := Y.tuple[Y.n].addr;
if (run_type = 1) then
    Z.tuple[l + 1].key := Y.tuple[Y.n].key
else
    str(Y.tuple[Y.n].addr:10,
        Z.tuple[l + 1].key);
end
else
begin; { non-leaf node }
P.tuple[1].key := Z.tuple[1].key;
P.tuple[1].addr := P.addr0;
P.addr0 := Y.tuple[Y.n].addr;
Z.tuple[1 + 1].key := Y.tuple[Y.n].key;
end;
zero_out(Y.n,Y);
P.n := P.n + 1;
Y.n := Y.n - 1;
end;
write_index(yaddr,Y);
write_index(paddr,P);
write_index(zaddr,Z);
delete_completed := true;
end;  { redistribute procedure }

procedure combine(right_sibling,leaf_node:boolean;
    l,yaddr,zaddr:integer;
    var paddr:integer;
    var P,Y,Z:filerec);
    { sibling nodes must be combined }
var s : integer;
begin;
    if (right_sibling) then  { P has a right sibling } 
        begin;
            s := P.n;
P.n := P.n + Y.n;
            if (not(leaf_node)) then 
                begin;
                    s := s + 1;
P.tuple[s].key := Z.tuple[l].key;
P.tuple[s].addr := Y.addr0;
P.n := P.n + 1;
                end;
        i := 0;
        while (i < Y.n) do 
            begin;
                s := s + 1;
i := i + 1;
P.tuple[s].key := Y.tuple[i].key;
P.tuple[s].addr := Y.tuple[i].addr;
            end;
P.next_node := Y.next_node;
        return_avail (yaddr,index_avail);
write_index(paddr,P);
Z.n := Z.n - 1;
i := 1;
while (i <= Z.n) do
begin;
    Z.tuple[i].key := Z.tuple[i + 1].key;
    Z.tuple[i].addr := Z.tuple[i + 1].addr;
    i := i + 1;
end;
zero_out(Z.n + 1,Z);
P := Z;
paddr := zaddr;
end
else
begin;  \{ P must have a left sibling \}
    s := Y.n;
    Y.n := Y.n + P.n;
    if \( \text{not(leaf\_node)} \) then
    begin;
        s := s + 1;
        Y.n := Y.n + 1;
        Y.tuple[s].key := Z.tuple[1 + l].key;
        Y.tuple[s].addr := P.addr0;
    end;
    i := 0;
while (i < P.n) do
begin;
    s := s + 1;
    i := i + 1;
    Y.tuple[s].key := P.tuple[i].key;
    Y.tuple[s].addr := P.tuple[i].addr;
end;
Y.next_node := P.next_node;
return_avail(paddr,index_avail);
write_index(yaddr,Y);
Z.n := Z.n - 1;
i := 1 + 1;
while (i <= Z.n) do
begin;
    Z.tuple[i].key := Z.tuple[i + 1].key;
    Z.tuple[i].addr := Z.tuple[i + 1].addr;
    i := i + 1;
end;
zero_out(Z.n + 1,Z);
P := Z;
paddr := zaddr;
end;  \{ combine procedure \}

procedure delete(localroot,run_type: integer);
    \{ delete tuple from node \}
var i, j, k, l, s : integer;
    paddr, qaddr, yaddr, zaddr, temp : integer;
delete_completed, leaf_node, right_sibling : boolean;

begin;

    paddr := localroot;
    leaf_node := true;
delete_completed := false;
l := 99;
P.n := P.n - 1;
l := save_i;
return_avail(P.tuple[l].addr,pbk_avail);
while (1 <= P.n) do { delete i-th element }
begin;
P.tuple[l].addr := P.tuple[l + 1].addr;
P.tuple[l].key := P.tuple[l + 1].key;
l := l + 1;
end;
zero_out(P.n + 1,P);

    temp := round(m/2 + 0.49) - 1;
while ((P.n < (round(m/2 + 0.49) -1)) and
    (paddr <> subroot)) do
begin; { must combine or redistribute P }
parent(zaddr,Z); { get P's parent - Z }
l := find_i(Z,paddr);
if (l < Z.n) then { P has a right sibling (Y) }
begin;
    right_sibling := true;
l := l + 1;
yaddr := Z.tuple[l].addr;
end
else
begin; { P must have a left sibling (Y) }
    right_sibling := false;
if (1 <> l) then
    yaddr := Z.tuple[l - 1].addr
else
    yaddr := Z.addr0;
l := l - 1;
end; { P left sibling }
read_index(yaddr,Y);
if (Y.n >= round(m/2 + 0.49)) then
    { redistribute P, Y, Z }
redistribute(right_sibling,leaf_node,l,
    run_type,yaddr,zaddr,paddr,
P,Y,Z,delete_completed)
else  
  { combine P & Y }  
  combine(right_sibling,leaf_node,1,  
        yaddr,zaddr,paddr,P,Y,Z);  

leaf_node := false;  
end;  { while }  

if (not(delete_completed)) then  
  write_index(paddr,P);  
end;  { delete procedure }  

procedure read_pbk;  
  { read record from pbkfile }  
begin;  
  seek (pbkfile,pbkaddr);  
  read (pbkfile,pbk_rec);  
  while ((not eof(pbkfile)) and  
          (pbk_rec.delete_flag = 'd')) do  
  begin; 
    return_avail(pbkaddr,pbk_avail);  
    pbkaddr := pbkaddr + 1;  
    seek (pbkfile,pbkaddr);  
    read (pbkfile,pbk_rec);  
  end;  
end;  { read_pbk }  

procedure process_a;  
  { create tree }  
begin;  
  while ((density_flag <> 'd') and  
          (density_flag <> 'n')) do  
  begin; 
    writeln('Is the index dense or non-dense  
            (d/n respectively)');  
    readln (density_flag);  
    if (density_flag <> 'd') and  
       (density_flag <> 'n') then  
       begin;  
         writeln ('');  
         writeln ('respond "d" for dense or  
                   "n" for non-dense');  
         writeln ('');  
       end;  
  end;  { while density_flag <> d or n }  
  writeln('index being created, please wait...');  
  rootaddr := 0;  
  pbkaddr := 0;
nodeaddr := 0;
seq_index := 0;
read_pbk;
while (not eof(pbkfile)) do
begin;
    hold_parent.node_ptr := -1;
    x := pbk_rec.pbk;
    search(rootaddr,1,x);
    if (j = 0) then
    begin;
        insert(nodeaddr,1);
        if (density_flag = 'n') then
        begin;
            if ((hold_parent.node_ptr < 0) and
                (not(first_time))) then
                save_upper_leaf(i,nodeaddr,P)
            else
                if (first_time) then
                    save_upper_leaf(i,nodeaddr,R);
                first_time := false;
                str(pbkaddr:10,x);
                left_most_leaf := 0;
                insert(0,2);
        end; { non-dense, lower tree insert }
    end
    else { j = 1, pbk already exist }
    begin;
        if (density_flag = 'd') then
        begin;
            write ('error - attribute already exists');
            writeln (' pbk = ',pbk_rec.pbk);
        end
        else
        begin;
            if (hold_parent.node_ptr < 0) then
                save_upper_leaf(i,nodeaddr,P);
            str(pbkaddr:10,x);
            search(P.tuple[i].addr,2,x);
            if (j = 0) then
                insert(nodeaddr,2)
            else
                begin;
                    write('pbk/sku already exist for ');
                    writeln(pbk_rec.pbk,'/',
                        pbk_rec.sku);
                end;
        end; { end density_flag <> 'd' }
    end;
end; { end j = 1 }
procedure process_a;   { process_a procedure }

begin;
  pbkaddr := pbkaddr + 1;
  read_pbk;
  seq_pbkfile := pbkaddr - 1;
  writeln('index has been created');
end;  {end of while }

procedure process_b;   { search tree }

begin;
  if (density_flag = 'd') then
    seq_search(rootaddr)
  else
    begin;
      writeln('What pricebook(00-99):');
      readln(pbk_rec.pbk);
      x := pbk_rec.pbk;
      hold_parent.node_ptr := -1;
      search(rootaddr,I,x);
      if (j = 0) then
        writeln('price book does not exist')
      else
        seq_search(P.tuple[i].addr)
    end;  {density_flag <> d; non-dense }
end;  {process_b }

procedure process_c;   { insert into tree }

begin;
  writeln('What pricebook(00-99):');
  readln(pbk_rec.pbk);
  x := pbk_rec.pbk;
  hold_parent.node_ptr := -1;
  search(rootaddr,I,x);
  if (j = 0) then
    begin;
      pbkaddr := next_avail_rec(pbk_avail,seq_pbkfile);
      writeln('What product number(10 digit):');
      readln(pbk_rec.sku);
      insert_into_pbkfile;
      insert(nodeaddr,1);
      if (density_flag = 'n') then
        begin;  { non-dense tree insert }
          if (hold_parent.node_ptr < 0) then
            save_upper_leaf(i,nodeaddr,P);
          str(pbkaddr:10,x);
        end;  {non-dense tree insert }
    end;  {process_c}
left_most_leaf := 0;
insert(0,2);
writeln('Insertion to both trees completed');
end;  { non_dense tree insertion }
else
if (density_flag = 'd') then
begin;
writeln ('error - insertion not possible;
already exists');
end
else
begin;
writeln('What product number(10 character):');
readln(pbk_rec.sku);
next_avail_rec(pbk_avail,seq_pbkfile);
str(pbkaddr:10,x);
subroot := P.tuple[i].addr;
if (hold_parent.node_ptr < 0) then
save_upper_leaf(i,nodeaddr,P);
search(P.tuple[i].addr,2,x);
save_i := i;
insert_into_pbkfile;
in(nodaddr,2);
end;  { non-dense insertion into lower tree }
end;  { process_c procedure }

procedure process_d;  { delete from tree }
begin;
writeln('What pricebook(00-99):');
readln(pbk_rec.pbk);
x := pbk_rec.pbk;
hold_parent.node_ptr := -1;
search(rootaddr,l,x);
if (j = 0) then
writeln('price book does not exist,
action not possible')
else  { j = 1, pbk found }
if (density_flag = 'd') then
begin;
save_i := i;
delete(nodeaddr,1);
seek (pbkfile,nodeaddr);
read (pbkfile,pbk_rec);
pbk_rec.delete_flag := 'd';
seek (pbkfile,nodeaddr);
write (pbkfile,pbk_rec);
end
else
begin;
writeln('What record address:');
readln(delete_addr);
str(delete_addr:10,x);
subroot := P.tuple[i].addr;
if (hold_parent.node_ptr < 0) then
  save_upper_leaf(i,nodeaddr,P);
search(P.tuple[i].addr,2,x);
save_i := i;
if (j = 0) then
  writeln('record does not exist, action not possible')
else
begin; { set delete flag on pbk record
  seek (pbkfile,delete_addr);
  read (pbkfile,pbk_rec);
  pbk_rec.delete_flag := 'd';
  seek (pbkfile,delete_addr);
  write (pbkfile,pbk_rec);
  delete(nodeaddr,2);
end;
end; { j = 1 }
end; { process_d procedure }

begin; { main program }
assign(pbkfile,'A:recs.dat');
(assign(index,'A:index.dat');
for i := 1 to 100 do
  begin
    index_avail[i] := 0;
    pbk_avail[i] := 0;
  end;
first_time := true;
last_write := false;
insert_processed := false;
recs_read := 0;
recs_written := 0;
density_flag := ' 1;
command := '1;
while (command <> 'E') do
begin
  writeln('what do you wish to do: ');
  writeln('  A. create');
  writeln('  B. search');
  writeln('  C. insert');
  writeln('  D. delete');
  writeln('  E. exit');
  readln(command);
  if (command = 'E') or (command = 'e') then
    command := 'E' { exit from program }
  else { check for a,b,c,d - legal commands }
  begin;
    writeln('Do you want the node displayed?(y/n) ')
    readln(display);
    if (command = 'A') or (command = 'a') then
      process_a { create tree }
    else
      if (command = 'b') or (command = 'B') then
        process_b { search tree }
      else
        if (command = 'c') or (command = 'C') then
          process_c { insert into tree }
        else
          if (command = 'd') or (command = 'D') then
            process_d { delete from tree }
          else
            begin;
              writeln('Invalid response: 
              A, B, C, D or E only');
              writeln('Please retry.');
            end;
        end; { check for a,b,c,d }
    end;
  if (command <> 'E') then
  begin
    write('it took ', recs_read,' read(s) and ');
    writeln(recs_written,' write(s) 
    to accomplish your task.');
    writeln(' ');
    recs_read := 0;
    recs_written := 0;
  end
end
end; { while command <> 'e'}

if (insert_processed) then { force last write }
begin;
  last_write := true;
  pbkaddr := next_avail_rec(pbk_avail,seq_pbkfile);
  seek (pbkfile,pbkaddr);
  write(pbkfile,pbk_rec);
end
end.

Real World Data

Table 1 is the initial data that were used in the implementation of the program MULTI.

The description of the fields in Table 1 are as follows. Address is the address of the record in the pbkfile. Delete_flag designates whether or not the record has been marked for deletion from the file. Pbk is the pricebook field. Sku is the product number. Uom is the unit of measure. Price is the price the customer is charged. Cost is the cost to the manufacturer to buy and manufacture the product.

TABLE 1

Initial Data Used in Test Runs of MULTI

<table>
<thead>
<tr>
<th>DELETE</th>
<th>ADDRESS</th>
<th>FLAG</th>
<th>PBK</th>
<th>SKU</th>
<th>UOM</th>
<th>PRICE</th>
<th>COST</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>47</td>
<td>01249141</td>
<td>DZ</td>
<td></td>
<td>912</td>
<td>544</td>
</tr>
<tr>
<td>DELETE ADDRESS</td>
<td>FLAG</td>
<td>PBK</td>
<td>SKU</td>
<td>UOM</td>
<td>PRICE</td>
<td>COST</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-----</td>
<td>-------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>47</td>
<td>06649141</td>
<td>DZ</td>
<td>960</td>
<td>604</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>77</td>
<td>99911111</td>
<td>EA</td>
<td>321</td>
<td>123</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>47</td>
<td>50578741</td>
<td>DZ</td>
<td>1156</td>
<td>843</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>47</td>
<td>01249141</td>
<td>DZ</td>
<td>963</td>
<td>544</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>47</td>
<td>60772141</td>
<td>DZ</td>
<td>2148</td>
<td>1304</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>47</td>
<td>01620177</td>
<td>DZ</td>
<td>440</td>
<td>144</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>90</td>
<td>77711111</td>
<td>EA</td>
<td>543</td>
<td>432</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>47</td>
<td>89119588</td>
<td>EA</td>
<td>9863</td>
<td>6462</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>47</td>
<td>98444788</td>
<td>EA</td>
<td>1611</td>
<td>1251</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>32</td>
<td>01249141</td>
<td>DZ</td>
<td>1109</td>
<td>544</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>32</td>
<td>06649141</td>
<td>DZ</td>
<td>979</td>
<td>604</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>32</td>
<td>41187277</td>
<td>DZ</td>
<td>506</td>
<td>212</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>32</td>
<td>50578741</td>
<td>DZ</td>
<td>1097</td>
<td>843</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>32</td>
<td>56878741</td>
<td>DZ</td>
<td>926</td>
<td>573</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>32</td>
<td>60772141</td>
<td>DZ</td>
<td>2502</td>
<td>1304</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>32</td>
<td>01620177</td>
<td>DZ</td>
<td>473</td>
<td>144</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>32</td>
<td>71101188</td>
<td>DZ</td>
<td>4185</td>
<td>3148</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>32</td>
<td>89119588</td>
<td>EA</td>
<td>9863</td>
<td>6462</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>51</td>
<td>01620177</td>
<td>DZ</td>
<td>399</td>
<td>144</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>51</td>
<td>01249141</td>
<td>DZ</td>
<td>1056</td>
<td>544</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>99</td>
<td>01249141</td>
<td>DZ</td>
<td>963</td>
<td>544</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>99</td>
<td>06649141</td>
<td>DZ</td>
<td>955</td>
<td>604</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>99</td>
<td>41187277</td>
<td>DZ</td>
<td>402</td>
<td>212</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>99</td>
<td>50578741</td>
<td>DZ</td>
<td>1087</td>
<td>843</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>86</td>
<td>56878741</td>
<td>DZ</td>
<td>809</td>
<td>573</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>86</td>
<td>60772141</td>
<td>DZ</td>
<td>1999</td>
<td>1304</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>86</td>
<td>01620177</td>
<td>DZ</td>
<td>356</td>
<td>144</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>86</td>
<td>71101188</td>
<td>DZ</td>
<td>4387</td>
<td>3148</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>86</td>
<td>89119588</td>
<td>EA</td>
<td>9863</td>
<td>6462</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>86</td>
<td>98444788</td>
<td>EA</td>
<td>1670</td>
<td>1251</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>27</td>
<td>06649141</td>
<td>DZ</td>
<td>1017</td>
<td>604</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>27</td>
<td>01249141</td>
<td>DZ</td>
<td>1138</td>
<td>544</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>27</td>
<td>50578741</td>
<td>DZ</td>
<td>1203</td>
<td>843</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>27</td>
<td>41187277</td>
<td>DZ</td>
<td>517</td>
<td>212</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>59</td>
<td>56878741</td>
<td>DZ</td>
<td>945</td>
<td>573</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>59</td>
<td>60772141</td>
<td>DZ</td>
<td>2191</td>
<td>1304</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>59</td>
<td>01620177</td>
<td>DZ</td>
<td>475</td>
<td>144</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>59</td>
<td>71101188</td>
<td>EA</td>
<td>4232</td>
<td>3148</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>12</td>
<td>89119588</td>
<td>EA</td>
<td>9915</td>
<td>6462</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>12</td>
<td>01620177</td>
<td>DZ</td>
<td>425</td>
<td>144</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELETE ADDRESS FLAG</td>
<td>PBK</td>
<td>SKU</td>
<td>UOM</td>
<td>PRICE</td>
<td>COST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>-------</td>
<td>-------------</td>
<td>-----</td>
<td>-------</td>
<td>------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>12</td>
<td>01249141</td>
<td>DZ</td>
<td>1229</td>
<td>514</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>01</td>
<td>06878741</td>
<td>DZ</td>
<td>770</td>
<td>499</td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>01</td>
<td>57772141</td>
<td>DZ</td>
<td>1333</td>
<td>965</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>01</td>
<td>01620177</td>
<td>DZ</td>
<td>307</td>
<td>144</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>01</td>
<td>89119588</td>
<td>EA</td>
<td>9863</td>
<td>6462</td>
<td></td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>01</td>
<td>71101188</td>
<td>EA</td>
<td>4076</td>
<td>3148</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>01</td>
<td>98444788</td>
<td>EA</td>
<td>1579</td>
<td>1251</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>79</td>
<td>57772141</td>
<td>DZ</td>
<td>1289</td>
<td>965</td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>79</td>
<td>41187277</td>
<td>DZ</td>
<td>476</td>
<td>212</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>79</td>
<td>50578741</td>
<td>DZ</td>
<td>1197</td>
<td>843</td>
<td></td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>79</td>
<td>06878741</td>
<td>DZ</td>
<td>738</td>
<td>499</td>
<td></td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>79</td>
<td>56878741</td>
<td>DZ</td>
<td>884</td>
<td>573</td>
<td></td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>79</td>
<td>60772141</td>
<td>DZ</td>
<td>1943</td>
<td>1304</td>
<td></td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>79</td>
<td>01620177</td>
<td>DZ</td>
<td>387</td>
<td>144</td>
<td></td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>45</td>
<td>66611111</td>
<td>EA</td>
<td>4321</td>
<td>3213</td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>45</td>
<td>99911111</td>
<td>EA</td>
<td>321</td>
<td>231</td>
<td></td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>42</td>
<td>99911111</td>
<td>EA</td>
<td>5432</td>
<td>4321</td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>42</td>
<td>77711111</td>
<td>EA</td>
<td>6543</td>
<td>4321</td>
<td></td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>42</td>
<td>67711111</td>
<td>EA</td>
<td>543</td>
<td>432</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>42</td>
<td>44411111</td>
<td>EA</td>
<td>432</td>
<td>333</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER VII

CONCLUSION

In this work, a generalized index to a relational database, called multitree, has been described.

For dense attributes the multitree acts the same as a B+ tree and for non-dense attributes it is composed of a set of linked B+ trees. The access time is at least as fast or faster than an inverted file for non-dense attributes and a B tree for dense attributes. The update time is of the same complexity as that of a B tree for any kind of attribute and an improvement over inverted files.

The multitree allows for a secondary attribute to be stored in the lower tree which can improve search time and decrease the number of total accesses needed when a search is being conducted within the attribute value for the secondary value as described in the evaluation of SEARCH.

Overall, the multitree provides numerous improvements and enhancements over current structures.
Appendix A

Real World Example--Dense Multitree
The error messages generated during the create are caused because a non-dense file is being used to create the dense index. If a dense file were used then there would be no such messages. The user's responses are in bold print.

A:\>multi

what do you wish to do:
A. create
B. search
C. insert
D. delete
E. exit

a

Is the index dense or non-dense (d/n) respectively?

index is being created, please wait...
error - attribute already exists pbk = 47
error - attribute already exists pbk = 47
error - attribute already exists pbk = 47
error - attribute already exists pbk = 47
error - attribute already exists pbk = 47
error - attribute already exists pbk = 47
error - attribute already exists pbk = 47
error - attribute already exists pbk = 32
error - attribute already exists pbk = 32
error - attribute already exists pbk = 32
error - attribute already exists pbk = 32
error - attribute already exists pbk = 32
error - attribute already exists pbk = 32
error - attribute already exists pbk = 32
error - attribute already exists pbk = 32
error - attribute already exists pbk = 32
error - attribute already exists pbk = 32
error - attribute already exists pbk = 32
error - attribute already exists pbk = 32
error - attribute already exists pbk = 32
error - attribute already exists pbk = 32
error - attribute already exists pbk = 32
error - attribute already exists pbk = 27
error - attribute already exists pbk = 27

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
error - attribute already exists  pbk = 27
error - attribute already exists  pbk = 59
error - attribute already exists  pbk = 59
error - attribute already exists  pbk = 59
error - attribute already exists  pbk = 51
error - attribute already exists  pbk = 12
error - attribute already exists  pbk = 12
error - attribute already exists  pbk = 01
error - attribute already exists  pbk = 01
error - attribute already exists  pbk = 01
error - attribute already exists  pbk = 01
error - attribute already exists  pbk = 01
error - attribute already exists  pbk = 01
error - attribute already exists  pbk = 79
error - attribute already exists  pbk = 79
error - attribute already exists  pbk = 79
error - attribute already exists  pbk = 79
error - attribute already exists  pbk = 79
error - attribute already exists  pbk = 79
error - attribute already exists  pbk = 79
error - attribute already exists  pbk = 45
error - attribute already exists  pbk = 42
error - attribute already exists  pbk = 42
error - attribute already exists  pbk = 42
index has been created
it took 110 read(s) and 22 write(s) to accomplish your task

what do you wish to do:
   A. create
   B. search
   C. insert
   D. delete
   E. exit

b

Which type of search would you like:
   a. Search for a specific item, of
   b. Search entire file producing file list

a

data for pbk 99 is:
   sku   uom  price  cost
01249141  dz    963   544

it took 4 read(s) and 0 write(s) to accomplish your task
what do you wish to do:
A. create
B. search
C. insert
D. delete
E. exit

Which type of search would you like:
   a. Search for a specific item, of
   b. Search entire file producing file list

data from the file is:

<table>
<thead>
<tr>
<th>pbk</th>
<th>sku</th>
<th>uom</th>
<th>price</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>06878741</td>
<td>dz</td>
<td>770</td>
<td>499</td>
</tr>
<tr>
<td>12</td>
<td>89119588</td>
<td>ea</td>
<td>9915</td>
<td>6462</td>
</tr>
<tr>
<td>27</td>
<td>06649141</td>
<td>dz</td>
<td>1017</td>
<td>604</td>
</tr>
<tr>
<td>32</td>
<td>01249141</td>
<td>dz</td>
<td>1109</td>
<td>544</td>
</tr>
<tr>
<td>42</td>
<td>66611111</td>
<td>ea</td>
<td>4321</td>
<td>3213</td>
</tr>
<tr>
<td>47</td>
<td>01249141</td>
<td>dz</td>
<td>912</td>
<td>544</td>
</tr>
<tr>
<td>51</td>
<td>01620177</td>
<td>dz</td>
<td>399</td>
<td>144</td>
</tr>
<tr>
<td>59</td>
<td>56878741</td>
<td>dz</td>
<td>945</td>
<td>573</td>
</tr>
<tr>
<td>77</td>
<td>99911111</td>
<td>ea</td>
<td>321</td>
<td>123</td>
</tr>
<tr>
<td>79</td>
<td>57772141</td>
<td>dz</td>
<td>1289</td>
<td>965</td>
</tr>
<tr>
<td>86</td>
<td>56878741</td>
<td>dz</td>
<td>809</td>
<td>573</td>
</tr>
<tr>
<td>90</td>
<td>77711111</td>
<td>ea</td>
<td>543</td>
<td>432</td>
</tr>
<tr>
<td>99</td>
<td>01249141</td>
<td>dz</td>
<td>963</td>
<td>544</td>
</tr>
</tbody>
</table>

it took 6 read(s) and 0 write(s) to accomplish your task.

what do you wish to do:
A. create
B. search
C. insert
D. delete
E. exit

What pricebook (00-99):
86
error - insertion not possible; already exists
it took 2 read(s) and 0 write(s) to accomplish your task.
what do you wish to do:
A. create  
B. search  
C. insert  
D. delete  
E. exit

c

What pricebook (00-99):
55
What product number (10 digit):
99911111
you must now enter data for the pbkfile insert

What is the unit of measure?
EA

the next two fields are numeric with implied decimals, do not input decimals

What is the price:
5544
What is the cost?
3322

The entry for the pbkfile is as follows:
pbk: 55  sku: 99911111
uom: ea  price: 5544  cost: 3322

Are these values acceptable (y/n)?
y
record inserted into pbkfile at address 61
it took 2 read(s) and 1 write(s) to accomplish your task

what do you wish to do:
A. create  
B. search  
C. insert  
D. delete  
E. exit

b

Which type of search would you like:
  a. Search for a specific item, of
  b. Search entire file producing file list

b
data from the file is:

<table>
<thead>
<tr>
<th>pbk</th>
<th>sku</th>
<th>uom</th>
<th>price</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>06878741</td>
<td>dz</td>
<td>770</td>
<td>499</td>
</tr>
<tr>
<td>12</td>
<td>89119588</td>
<td>ea</td>
<td>9915</td>
<td>6462</td>
</tr>
<tr>
<td>27</td>
<td>06649141</td>
<td>dz</td>
<td>1017</td>
<td>604</td>
</tr>
<tr>
<td>32</td>
<td>01249141</td>
<td>dz</td>
<td>1109</td>
<td>544</td>
</tr>
<tr>
<td>42</td>
<td>66611111</td>
<td>ea</td>
<td>4321</td>
<td>3213</td>
</tr>
<tr>
<td>47</td>
<td>01249141</td>
<td>dz</td>
<td>912</td>
<td>544</td>
</tr>
<tr>
<td>51</td>
<td>01620177</td>
<td>dz</td>
<td>399</td>
<td>144</td>
</tr>
<tr>
<td>55</td>
<td>99911111</td>
<td>ea</td>
<td>5544</td>
<td>3322</td>
</tr>
<tr>
<td>59</td>
<td>56878741</td>
<td>dz</td>
<td>945</td>
<td>573</td>
</tr>
<tr>
<td>77</td>
<td>99911111</td>
<td>ea</td>
<td>321</td>
<td>123</td>
</tr>
<tr>
<td>79</td>
<td>57772141</td>
<td>dz</td>
<td>1289</td>
<td>965</td>
</tr>
<tr>
<td>86</td>
<td>56878741</td>
<td>dz</td>
<td>809</td>
<td>573</td>
</tr>
<tr>
<td>90</td>
<td>77711111</td>
<td>ea</td>
<td>543</td>
<td>432</td>
</tr>
<tr>
<td>99</td>
<td>01249141</td>
<td>dz</td>
<td>963</td>
<td>544</td>
</tr>
</tbody>
</table>

it took 6 read(s) and 0 write(s) to accomplish your task.

what do you wish to do:
A. create
B. search
C. insert
D. delete
E. exit
d

What pricebook (00-99):
32
address being returned = 10
it took 2 read(s) and 1 write(s) to accomplish your task

what do you wish to do:
A. create
B. search
C. insert
D. delete
E. exit
b

Which type of search would you like:
  a. Search for a specific item, of
  b. Search entire file producing file list
b

data from the file is:

<table>
<thead>
<tr>
<th>pbk</th>
<th>sku</th>
<th>uom</th>
<th>price</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>06878741</td>
<td>dz</td>
<td>770</td>
<td>499</td>
</tr>
<tr>
<td>12</td>
<td>89119588</td>
<td>ea</td>
<td>9915</td>
<td>6462</td>
</tr>
<tr>
<td>27</td>
<td>06649141</td>
<td>dz</td>
<td>1017</td>
<td>604</td>
</tr>
</tbody>
</table>
it took 6 read(s) and 0 write(s) to accomplish your task.

what do you wish to do:
A. create
B. search
C. insert
D. delete
E. exit

What pricebook (00-99):
98
What product number (10 digit):
33311111
you must now enter data for the pbkfile insert

What is the unit of measure?
DZ

the next two fields are numeric with implied decimals, do not input decimals

What is the price:
4321
What is the cost?
2341

The entry for the pbkfile is as follows:
  pbk: 98  sku: 33311111
  uom: dz  price: 4321  cost: 2341

Are these values acceptable (y/n)?
y
record inserted into pbkfile at address 10
it took 2 read(s) and 1 write(s) to accomplish your task
what do you wish to do:
A. create
B. search
C. insert
D. delete
E. exit
Appendix B

Real World Example—Non-Dense Multitree
A:\>multi
what do you wish to do:
A. create
B. search
C. insert
D. delete
E. exit

a

Is the index dense or non-dense (d/n) respectively?

n

index is being created, please wait...
index has been created
it took 168 read(s) and 127 write(s) to accomplish your task.

what do you wish to do:
A. create
B. search
C. insert
D. delete
E. exit

b

What pricebook (00-99):

47

Do you wish to search for:
1. All records in pricebook 47
2. All secondary attributes in pricebook 47
3. All records with a specific secondary attribute in pricebook 47

Enter 1, 2, or 3

1

Data for pbk 47 is:

<table>
<thead>
<tr>
<th>sku</th>
<th>uom</th>
<th>price</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>01249141</td>
<td>dz</td>
<td>912</td>
<td>544</td>
</tr>
<tr>
<td>06649141</td>
<td>dz</td>
<td>960</td>
<td>604</td>
</tr>
<tr>
<td>50578741</td>
<td>dz</td>
<td>1156</td>
<td>843</td>
</tr>
<tr>
<td>60772141</td>
<td>dz</td>
<td>2148</td>
<td>1304</td>
</tr>
<tr>
<td>01620177</td>
<td>dz</td>
<td>440</td>
<td>144</td>
</tr>
<tr>
<td>89119588</td>
<td>ea</td>
<td>9863</td>
<td>6462</td>
</tr>
<tr>
<td>98444788</td>
<td>ea</td>
<td>1611</td>
<td>1251</td>
</tr>
</tbody>
</table>

it took 6 read(s) and 0 write(s) to accomplish your task
what do you wish to do:
   A. create
   B. search
   C. insert
   D. delete
   E. exit
d
What pricebook (00-99): 47
What record address: 6
address being returned = 6
it took 4 read(s) and 1 write(s) to accomplish your task.
what do you wish to do:
   A. create
   B. search
   C. insert
   D. delete
   E. exit
b
What pricebook (00-99): 47
Do you wish to search for:
   1. All records in pricebook 47
   2. All secondary attributes in pricebook 47
   3. All records with a specific secondary attribute in pricebook 47
Enter 1, 2, or 3
2
Secondary attribute(s) for pbk 47 is/are:
  01249141
  06649141
  50578741
  60772141
  89119588
  98444788
it took 6 read(s) and 0 write(s) to accomplish your task
what do you wish to do:
   A. create
   B. search
   C. insert
   D. delete
   E. exit

c
What pricebook (00-99):
   47
What product number (10 digit):
   50578741
you must now enter data for the pbkfile insert

What is the unit of measure?
   EA

the next two fields are numeric with implied decimals, do not input decimals

What is the price:
   99
What is the cost?
   75

The entry for the pbkfile is as follows:
   pbk: 47  sku: 50578741
   uom: ea  price: 99  cost: 75

Are these values acceptable (y/n)?
   y
record inserted into pbkfile at address 6
insertion of product is complete
it took 5 read(s) and 3 write(s) to accomplish your task

what do you wish to do:
   A. create
   B. search
   C. insert
   D. delete
   E. exit

b
What pricebook (00-99):
   47
Do you wish to search for:
1. All records in pricebook 47
2. All secondary attributes in pricebook 47
3. All records with a specific secondary attribute in pricebook 47

Enter 1, 2, or 3
3

What product do you want?
50578741

Data for 47/ 50578741 is:
   uom  price  cost
   dz   1156  843
   ea    99   75

it took 7 read(s) and 0 write(s) to accomplish your task

what do you wish to do:
   A. create
   B. search
   C. insert
   D. delete
   E. exit
d

What pricebook (00-99):
12
What record address:
40

address being returned = 40
it took 4 read(s) and 1 write(s) to accomplish your task.

what do you wish to do:
   A. create
   B. search
   C. insert
   D. delete
   E. exit
b

What pricebook (00-99):
12
Do you wish to search for:
1. All records in pricebook 12
2. All secondary attributes in pricebook 12
3. All records with a specific secondary attribute in pricebook 12

Enter 1, 2, or 3
1

Data for pbk 12 is:

<table>
<thead>
<tr>
<th>sku</th>
<th>uom</th>
<th>price</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>89119588</td>
<td>ea</td>
<td>9863</td>
<td>6462</td>
</tr>
<tr>
<td>01249141</td>
<td>dz</td>
<td>912</td>
<td>544</td>
</tr>
</tbody>
</table>

it took 4 read(s) and 0 write(s) to accomplish your task.

what do you wish to do:
A. create
B. search
C. insert
D. delete
E. exit
d

What pricebook (00-99):
12
What record address:
41

address being returned = 41
it took 4 read(s) and 1 write(s) to accomplish your task.

what do you wish to do:
A. create
B. search
C. insert
D. delete
E. exit
b

What pricebook (00-99):
12

Do you wish to search for:
1. All records in pricebook 12
2. All secondary attributes in pricebook 12
3. All records with a specific secondary attribute in pricebook 12
Enter 1, 2, or 3
1

Data for pbk 12 is:

<table>
<thead>
<tr>
<th>sku</th>
<th>uom</th>
<th>price</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>89119588</td>
<td>ea</td>
<td>9863</td>
<td>6462</td>
</tr>
</tbody>
</table>

it took 4 read(s) and 0 write(s) to accomplish your task

what do you wish to do:
A. create
B. search
C. insert
D. delete
E. exit

c

What pricebook (00-99):
12

What product number (10 digit):
88811111

you must now enter data for the pbkfile insert

What is the unit of measure?
EA

the next two fields are numeric with implied decimals, do not input decimals

What is the price:
321
What is the cost?
123

The entry for the pbkfile is as follows:


Are these values acceptable (y/n)?
y
record inserted into pbkfile at address 41
insertion of product is complete
it took 4 read(s) and 1 write(s) to accomplish your task
what do you wish to do:  
A. create  
B. search  
C. insert  
D. delete  
E. exit  

c  

What pricebook (00-99):  
77  

What product number (10 digit):  
33311111  
you must now enter data for the pbkfile insert  

What is the unit of measure?  
EA  

the next two fields are numeric with implied decimals,  
do not input decimals  

What is the price:  
321  
What is the cost?  
111  

The entry for the pbkfile is as follows:  

pbk: 77  
sku: 33311111  
uom: ea  
price: 321  
cost: 111  

Are these values acceptable (y/n)?  

y  

record inserted into pbkfile at address 40  
insertion of product is complete  
it took 4 read(s) and 1 write(s) to accomplish your task  

what do you wish to do:  
A. create  
B. search  
C. insert  
D. delete  
E. exit  

c  

What pricebook (00-99):  
77  

What product number (10 digit):  
44411111  
you must now enter data for the pbkfile insert
What is the unit of measure?
KG

the next two fields are numeric with implied decimals,
do not input decimals

What is the price:
5432
What is the cost?
3249

The entry for the pbkfile is as follows:
 pbk: 77  sku: 44411111
   uom: kg price: 5432 cost: 3249

Are these values acceptable (y/n)?
y
record inserted into pbkfile at address 62
insertion of product is complete
it took 4 read(s) and 1 write(s) to accomplish your task

what do you wish to do:
 A. create
 B. search
 C. insert
 D. delete
 E. exit
 e
Appendix C

Data After Program Runs
TABLE 2

Data After Program Runs

<table>
<thead>
<tr>
<th>DELETE ADDRESS</th>
<th>PBK</th>
<th>SKU</th>
<th>UOM</th>
<th>PRICE</th>
<th>COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>47</td>
<td>01249141</td>
<td>DZ</td>
<td>912</td>
<td>544</td>
</tr>
<tr>
<td>1</td>
<td>47</td>
<td>06649141</td>
<td>DZ</td>
<td>960</td>
<td>604</td>
</tr>
<tr>
<td>2</td>
<td>77</td>
<td>99911111</td>
<td>EA</td>
<td>321</td>
<td>123</td>
</tr>
<tr>
<td>3</td>
<td>47</td>
<td>50578741</td>
<td>DZ</td>
<td>1156</td>
<td>843</td>
</tr>
<tr>
<td>4</td>
<td>47</td>
<td>01249141</td>
<td>DZ</td>
<td>963</td>
<td>544</td>
</tr>
<tr>
<td>5</td>
<td>47</td>
<td>60772141</td>
<td>DZ</td>
<td>2148</td>
<td>1304</td>
</tr>
<tr>
<td>6</td>
<td>47</td>
<td>50578741</td>
<td>EA</td>
<td>99</td>
<td>75</td>
</tr>
<tr>
<td>7</td>
<td>90</td>
<td>77711111</td>
<td>EA</td>
<td>543</td>
<td>432</td>
</tr>
<tr>
<td>8</td>
<td>47</td>
<td>89119588</td>
<td>EA</td>
<td>9863</td>
<td>6462</td>
</tr>
<tr>
<td>9</td>
<td>47</td>
<td>98444788</td>
<td>EA</td>
<td>1611</td>
<td>1251</td>
</tr>
<tr>
<td>10</td>
<td>98</td>
<td>33311111</td>
<td>DZ</td>
<td>4321</td>
<td>2341</td>
</tr>
<tr>
<td>11</td>
<td>32</td>
<td>06649141</td>
<td>DZ</td>
<td>979</td>
<td>604</td>
</tr>
<tr>
<td>12</td>
<td>32</td>
<td>41187277</td>
<td>DZ</td>
<td>506</td>
<td>212</td>
</tr>
<tr>
<td>13</td>
<td>32</td>
<td>50578741</td>
<td>DZ</td>
<td>1097</td>
<td>843</td>
</tr>
<tr>
<td>14</td>
<td>32</td>
<td>56878741</td>
<td>DZ</td>
<td>926</td>
<td>573</td>
</tr>
<tr>
<td>15</td>
<td>32</td>
<td>60772141</td>
<td>DZ</td>
<td>2502</td>
<td>1304</td>
</tr>
<tr>
<td>16</td>
<td>32</td>
<td>01620177</td>
<td>DZ</td>
<td>473</td>
<td>144</td>
</tr>
<tr>
<td>17</td>
<td>32</td>
<td>71101188</td>
<td>DZ</td>
<td>4185</td>
<td>3148</td>
</tr>
<tr>
<td>18</td>
<td>32</td>
<td>89119588</td>
<td>EA</td>
<td>9863</td>
<td>6462</td>
</tr>
<tr>
<td>19</td>
<td>51</td>
<td>01620177</td>
<td>DZ</td>
<td>399</td>
<td>144</td>
</tr>
<tr>
<td>20</td>
<td>51</td>
<td>01249141</td>
<td>DZ</td>
<td>1056</td>
<td>544</td>
</tr>
<tr>
<td>21</td>
<td>99</td>
<td>01249141</td>
<td>DZ</td>
<td>963</td>
<td>544</td>
</tr>
<tr>
<td>22</td>
<td>99</td>
<td>06649141</td>
<td>DZ</td>
<td>955</td>
<td>604</td>
</tr>
<tr>
<td>23</td>
<td>99</td>
<td>41187277</td>
<td>DZ</td>
<td>402</td>
<td>212</td>
</tr>
<tr>
<td>24</td>
<td>99</td>
<td>50578741</td>
<td>DZ</td>
<td>1087</td>
<td>843</td>
</tr>
<tr>
<td>25</td>
<td>86</td>
<td>56878741</td>
<td>DZ</td>
<td>809</td>
<td>573</td>
</tr>
<tr>
<td>26</td>
<td>86</td>
<td>60772141</td>
<td>DZ</td>
<td>1999</td>
<td>1304</td>
</tr>
<tr>
<td>27</td>
<td>86</td>
<td>01620177</td>
<td>DZ</td>
<td>356</td>
<td>144</td>
</tr>
<tr>
<td>28</td>
<td>86</td>
<td>71101188</td>
<td>EA</td>
<td>4387</td>
<td>3148</td>
</tr>
<tr>
<td>29</td>
<td>86</td>
<td>89119588</td>
<td>EA</td>
<td>9863</td>
<td>6462</td>
</tr>
<tr>
<td>30</td>
<td>86</td>
<td>98444788</td>
<td>EA</td>
<td>1670</td>
<td>1251</td>
</tr>
<tr>
<td>31</td>
<td>27</td>
<td>06649141</td>
<td>DZ</td>
<td>1017</td>
<td>604</td>
</tr>
<tr>
<td>32</td>
<td>27</td>
<td>01249141</td>
<td>DZ</td>
<td>1138</td>
<td>544</td>
</tr>
<tr>
<td>33</td>
<td>27</td>
<td>50578741</td>
<td>DZ</td>
<td>1203</td>
<td>843</td>
</tr>
<tr>
<td>34</td>
<td>27</td>
<td>41187277</td>
<td>DZ</td>
<td>517</td>
<td>212</td>
</tr>
<tr>
<td>35</td>
<td>59</td>
<td>56878741</td>
<td>DZ</td>
<td>945</td>
<td>573</td>
</tr>
<tr>
<td>36</td>
<td>59</td>
<td>60772141</td>
<td>DZ</td>
<td>2191</td>
<td>1304</td>
</tr>
<tr>
<td>37</td>
<td>59</td>
<td>01620177</td>
<td>DZ</td>
<td>475</td>
<td>144</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
<table>
<thead>
<tr>
<th>DELETE ADDRESS FLAG</th>
<th>PBK</th>
<th>SKU</th>
<th>UOM</th>
<th>PRICE</th>
<th>COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>59</td>
<td>71101188</td>
<td>EA</td>
<td>4232</td>
<td>3148</td>
</tr>
<tr>
<td>39</td>
<td>12</td>
<td>89119588</td>
<td>EA</td>
<td>9915</td>
<td>6462</td>
</tr>
<tr>
<td>40</td>
<td>77</td>
<td>33311111</td>
<td>EA</td>
<td>321</td>
<td>111</td>
</tr>
<tr>
<td>41</td>
<td>12</td>
<td>88811111</td>
<td>EA</td>
<td>321</td>
<td>123</td>
</tr>
<tr>
<td>42</td>
<td>01</td>
<td>06878741</td>
<td>DZ</td>
<td>770</td>
<td>499</td>
</tr>
<tr>
<td>43</td>
<td>01</td>
<td>57772141</td>
<td>DZ</td>
<td>1333</td>
<td>965</td>
</tr>
<tr>
<td>44</td>
<td>01</td>
<td>01620177</td>
<td>DZ</td>
<td>307</td>
<td>144</td>
</tr>
<tr>
<td>45</td>
<td>01</td>
<td>89119588</td>
<td>EA</td>
<td>9863</td>
<td>6462</td>
</tr>
<tr>
<td>46</td>
<td>01</td>
<td>71101188</td>
<td>EA</td>
<td>4076</td>
<td>3148</td>
</tr>
<tr>
<td>47</td>
<td>01</td>
<td>98444788</td>
<td>EA</td>
<td>1579</td>
<td>1251</td>
</tr>
<tr>
<td>48</td>
<td>79</td>
<td>57772141</td>
<td>DZ</td>
<td>1289</td>
<td>965</td>
</tr>
<tr>
<td>49</td>
<td>79</td>
<td>41187277</td>
<td>DZ</td>
<td>476</td>
<td>212</td>
</tr>
<tr>
<td>50</td>
<td>79</td>
<td>50578741</td>
<td>DZ</td>
<td>1197</td>
<td>843</td>
</tr>
<tr>
<td>51</td>
<td>79</td>
<td>06878741</td>
<td>DZ</td>
<td>738</td>
<td>499</td>
</tr>
<tr>
<td>52</td>
<td>79</td>
<td>56878741</td>
<td>DZ</td>
<td>884</td>
<td>573</td>
</tr>
<tr>
<td>53</td>
<td>79</td>
<td>60772141</td>
<td>DZ</td>
<td>1943</td>
<td>1304</td>
</tr>
<tr>
<td>54</td>
<td>79</td>
<td>01620177</td>
<td>DZ</td>
<td>387</td>
<td>144</td>
</tr>
<tr>
<td>55</td>
<td>45</td>
<td>66611111</td>
<td>EA</td>
<td>4321</td>
<td>3213</td>
</tr>
<tr>
<td>56</td>
<td>45</td>
<td>99911111</td>
<td>EA</td>
<td>321</td>
<td>231</td>
</tr>
<tr>
<td>57</td>
<td>42</td>
<td>99911111</td>
<td>EA</td>
<td>5432</td>
<td>4321</td>
</tr>
<tr>
<td>58</td>
<td>42</td>
<td>77711111</td>
<td>EA</td>
<td>6543</td>
<td>4321</td>
</tr>
<tr>
<td>59</td>
<td>42</td>
<td>67711111</td>
<td>EA</td>
<td>543</td>
<td>432</td>
</tr>
<tr>
<td>60</td>
<td>42</td>
<td>44411111</td>
<td>EA</td>
<td>432</td>
<td>333</td>
</tr>
<tr>
<td>61</td>
<td>55</td>
<td>99911111</td>
<td>EA</td>
<td>5544</td>
<td>3322</td>
</tr>
<tr>
<td>62</td>
<td>77</td>
<td>44411111</td>
<td>KG</td>
<td>5432</td>
<td>3249</td>
</tr>
</tbody>
</table>
BIBLIOGRAPHY


107


