A Uniform System for Data Base Directories

Chang

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

Part of the Computer Sciences Commons

Recommended Citation

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 wmu-scholarworks@wmich.edu.
The objective of this study is to provide an efficient uniform system for data base directories, so that every file in the data base will have the same type of efficient directory. Data base directories will then be uniformly generated, updated and maintained.

The uniform system which is developed here can handle not only discrete attributes, but also range attributes. For the range attributes, it provides the uniform ranges (i.e., a set of ranges such that the records are distributed uniformly between ranges.)

In this thesis, the design and implementation of the hybrid B-plus tree and inverted files which is called here A Uniform System for Data Base Directories is presented. A set of algorithms along with a number of examples are given to illustrate the construction and performance of the uniform system.
ACKNOWLEDGEMENTS

I am greatly indebted to my Thesis Advisor, Dr. Dalia Hotzkin, for her advice and encouragement during the design and implementation of the Uniform System for Data Base Directories. I am also grateful to Dr. David Johnson for his comments and suggestions on the thesis. A special thanks goes to Mr. David Hendrix for his enthusiasm in correcting my English.

My wife, Amy, deserves special recognition for her words of encouragement during the rough months of implementation.

Karl Chang
INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of “sectioning” the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.
CHANG, KARL

A UNIFORM SYSTEM FOR DATA BASE DIRECTORIES

WESTERN MICHIGAN UNIVERSITY M.S. 1983
# TABLE OF CONTENTS

**ACKNOWLEDGEMENTS** ................................................................. ii

**LIST OF TABLES** ........................................................................... v

**LIST OF FIGURES** ......................................................................... vi

**Chapter**

I. **INTRODUCTION** ................................................................. 1

II. **DESIGN OF THE SYSTEM** .................................................. 4

   Structure of the Hybrid Directory............................................. 4

   Example of the Hybrid Directory and corresponding file........... 5

   Structure of the Master B-plus Tree........................................... 5

   Node Record Description of the Master B-plus tree..................... 9

   Structure of an Inverted B-plus Tree........................................ 10

   Node Record Description of the Inverted B-plus Tree.................. 11

   Generation of the B-plus Tree ................................................. 12

   Description of the Generation Procedure................................. 12

III. **OPERATIONS OF THE SYSTEM** .......................................... 14

   Search Operation................................................................. 14

   Insertion Operation............................................................. 14

   Deletion Operation.............................................................. 17

   Performance and Analysis..................................................... 19

IV. **RUNNING PROCEDURE** ...................................................... 21

V. **CONCLUSION** ................................................................. 23
TABLE OF CONTENTS (CONTINUED)

APPENDIX A. PROGRAM DOCUMENTATION.........................25
APPENDIX B. DESCRIPTION OF ALGORITHMS..........................32
APPENDIX C. PROGRAM OUTLINES..................................37
APPENDIX D. SAMPLE FILES........................................61
APPENDIX E. SAMPLE RUNS.........................................71
BIBLIOGRAPHY.......................................................113
LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The Original Source File</td>
<td>6</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>4</td>
</tr>
<tr>
<td>2.</td>
<td>7</td>
</tr>
<tr>
<td>3.</td>
<td>9</td>
</tr>
<tr>
<td>4.</td>
<td>11</td>
</tr>
<tr>
<td>5.</td>
<td>16</td>
</tr>
<tr>
<td>6.</td>
<td>18</td>
</tr>
</tbody>
</table>

General Structure of the Hybrid Directory
Tree Diagram of a Hybrid Directory
Node Formats of the Master B-plus Tree
Node Formats of the Inverted B-plus Tree
Tree Diagram of Insertion Examples
Tree Diagram of Deletion Examples
CHAPTER I
INTRODUCTION

The basic features desired of a data base system that stores large amounts of data are fast access for retrieval, convenient update, and economy of storage. In order to achieve these goals, the files in the data base (DB) need to be associated with a directory. This directory is used for rapid accesses to any record in the files. Many techniques for organizing a directory have been implemented today, some of them are both efficient and elegant and have gained wide acceptance among all the directories.

Many hybrid tree organizations are proposed in the literature that combine some of the best qualities of hashing, binary trees, B-trees, and the many variants of these. Among them are hash trees (Coffman and Eve, 1970; Severance, 1974), extendible hashing (Fagin et al., 1979), virtual B-trees (Bayer, 1972; Knuth, 1973), binary B-tree (Bayer and McCreight, 1972), and 2-3 trees (Yao, 1978). The reader is directed to the original papers for details.

However, none of these implementations are acceptable for both nondense and dense attributes. A dense (or discrete) attribute is defined to be an attribute that takes on discrete values from a given list such as color = (blue, red, ..) or major = (cs, math, ...). A nondense (or range) attribute is an attribute that takes ranges of values such as age = (20 - 25, 25 - 40, ..) or salary = (0 - 1000, 1001 - 2000, ..).

Since we have to keep different types of directories in the data
base, maintenance becomes difficult. If there is only one type of
directory (suited for both dense and nondense attributes), searching
will be faster, the database will be easier to maintain, and
uniformity can be achieved without loss of efficiency. The purpose
of this study is to address these needs. In this thesis, the design
and implementation of the hybrid B-plus tree and inverted files are
called A Uniform System for Data Base Directory.

We chose to develop the hybrid structure of B-plus tree with
inverted files because it has several advantages over existing file
organizations.

For example, in the B-tree structure, each node has to contain
certain distinct keys. If there are too many identical keys, the
size of the node is unpredictable, which, in the B-tree, must be
predetermined. In this case, one may end up creating many linked
lists out of one node. This requires many additional disk accesses
to retrieve a specific record (even using the best search method.)
The hybrid structure collects all of the records which have the same
key value to construct a B-plus tree so that retrieval for nondense
attributes will no longer be a problem.

Also, in the pure inverted file structure, insertion of data
between the intervals causes unnecessary moving of the data.
However, using the B-plus tree structure reduces the number of
accesses needed for insertions. The uniform inverted file routines
added to the B-plus tree will enable the user to obtain uniform
ranges (i.e. a set of ranges such that the records are distributed
uniformly between ranges.) This will improve the time and space
required for processing range attributes.

Similarly, if a user wishes to retrieve the sorted list of all the key values, he simply retrieves to the leaf level of the B-plus tree since the leaf level of the master B-plus tree is always ordered. One can also use this feature to retrieve a sorted list of records from the leaf level of inverted B-plus tree.
CHAPTER II

DESCRIPTION OF THE STRUCTURE

Structure of the Directory

The hybrid directory of B-plus tree and inverted files has two parts (see figure 1):

(1) The master B-plus tree: The leaf level of the tree contains all of the key attribute values (or attribute value ranges for range attributes) and pointers to their corresponding inverted B-plus trees.

(2) The inverted B-plus trees: Each inverted tree is associated with one specific key attribute value (or an attribute range for range attribute) in the leaf nodes of the master tree. The leaf level of an inverted tree contains the record numbers (and associated attribute values for range attributes) of all the records which have the same attribute values.

![Diagram of the Hybrid Directory Structure](image)

**Figure 1** General Structure of the Hybrid Directory
The entries of the leaf are organized in ascending order while all the leaf nodes of a tree are linked together from left to right in ascending order. Thus, the leaf nodes of the master and inverted B-plus trees are referred to as the sequence set in which sequential processing can be applied. Also, the internal and leaf nodes of the tree have different formats. These formats are discussed under the section entitled structure of the master B-plus tree.

Example of Creating the Hybrid Directory

Assume an employee file has 40 records. Each record in the file contains department name, social security number, employee name, and salary field. First, store all of the record numbers and their corresponding discrete attributes (i.e. department) in ascending order according to the attribute value. This file is listed in Table 1.

We then, create the hybrid directory for the discrete attribute following the construction steps described in the last section of this chapter. After this, the Hybrid Directory is generated for this particular file. This Hybrid Directory is shown in Figure 2.
<table>
<thead>
<tr>
<th>Record No.</th>
<th>Attribute Value (i.e. Dept)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>MATH</td>
</tr>
<tr>
<td>5</td>
<td>GEOG</td>
</tr>
<tr>
<td>6</td>
<td>ENGL</td>
</tr>
<tr>
<td>7</td>
<td>ECON</td>
</tr>
<tr>
<td>9</td>
<td>ECON</td>
</tr>
<tr>
<td>10</td>
<td>ENGL</td>
</tr>
<tr>
<td>14</td>
<td>MATH</td>
</tr>
<tr>
<td>18</td>
<td>BEAS</td>
</tr>
<tr>
<td>19</td>
<td>ENGL</td>
</tr>
<tr>
<td>20</td>
<td>ENGL</td>
</tr>
<tr>
<td>24</td>
<td>CS</td>
</tr>
<tr>
<td>25</td>
<td>MATH</td>
</tr>
<tr>
<td>29</td>
<td>ECON</td>
</tr>
<tr>
<td>30</td>
<td>BEAS</td>
</tr>
<tr>
<td>31</td>
<td>ENGL</td>
</tr>
<tr>
<td>35</td>
<td>ENGL</td>
</tr>
<tr>
<td>36</td>
<td>CS</td>
</tr>
<tr>
<td>38</td>
<td>GEOG</td>
</tr>
<tr>
<td>47</td>
<td>EE</td>
</tr>
<tr>
<td>49</td>
<td>LING</td>
</tr>
<tr>
<td>52</td>
<td>ENGL</td>
</tr>
<tr>
<td>60</td>
<td>LING</td>
</tr>
<tr>
<td>62</td>
<td>MGMT</td>
</tr>
<tr>
<td>63</td>
<td>ENGL</td>
</tr>
<tr>
<td>66</td>
<td>MGMT</td>
</tr>
<tr>
<td>71</td>
<td>MGMT</td>
</tr>
<tr>
<td>78</td>
<td>ENGL</td>
</tr>
<tr>
<td>79</td>
<td>ENGL</td>
</tr>
<tr>
<td>80</td>
<td>BEAS</td>
</tr>
<tr>
<td>81</td>
<td>ME</td>
</tr>
<tr>
<td>84</td>
<td>EE</td>
</tr>
<tr>
<td>85</td>
<td>BEAS</td>
</tr>
<tr>
<td>86</td>
<td>IE</td>
</tr>
<tr>
<td>89</td>
<td>SOC</td>
</tr>
<tr>
<td>90</td>
<td>EE</td>
</tr>
<tr>
<td>94</td>
<td>ME</td>
</tr>
<tr>
<td>95</td>
<td>SOC</td>
</tr>
<tr>
<td>96</td>
<td>IE</td>
</tr>
<tr>
<td>97</td>
<td>BEAS</td>
</tr>
<tr>
<td>106</td>
<td>IE</td>
</tr>
</tbody>
</table>
Figure 2 Tree Diagram of a Hybrid Directory
Structure of the Master B-plus Tree

The Master B-plus Tree is a B-tree structure with the following additional properties:

1. All of the key values reside in the leaf nodes.

2. The logical structure of the tree consists of the attribute key value parts.

3. Leaf nodes are linked together from left to right.

4. A right child node of an entry contains the key value "greater than or equal to" its parent's entry, rather than just "greater than" as in regular B-tree structures.

In the Master B-plus tree, all keys reside in the leaves. The upper levels which are organized as a B-tree consist of the proper attribute values as the separators. The leaf nodes are linked together from left to right as a linked list and are referred to as the sequence set. It is good for the sequential process to retrieve all the key values of the pile file. The pointer which is associated with each key value in the master B-plus tree points to all records which have the same key as this key (these pointers are some times referred to as record numbers.) Each entry of a node in the internal levels of the master tree contains the attribute value and pointer to its child node.
Node Description of the Master B-plus Tree

The general node formats of the master B-plus tree are described by

**Internal node structure**

| N | LC | A1 | RC1 | A2 | RC2 | ... | An | RCn |

**Leaf node structure**

| N | K1 | P1 | K2 | P2 | ... | Kn | Pn | LINK |

Figure 3 node description of master B-plus tree

where

- \( N \) — number of entries in the node of master tree
- \( LC \) — pointer to the left child node of master tree
- \( Ai \) — an \( i \)th entry (an attribute value) in an internal node of master tree
- \( RC_i \) — pointer to the right child node of the \( i \)th entry in an internal node of master tree
- \( Ki \) — the \( i \)th entry (for the discrete attributes, the \( i \)th key value; for the range attributes, the lowest and highest attribute value in \( i \)th range)
- \( Pi \) — the pointer to the corresponding inverted tree for the \( i \)th entry of the master leaf
- \( LINK \) — the link pointer to the right leaf node of master tree
Structure of Inverted B-plus Trees

An inverted B-plus tree is a B-tree structure with the following additional properties:

1. The numbers of all records which have an identical key value reside in the leaf nodes.

2. Leaf nodes are linked together from left to right.

3. The internal nodes are handled in the same manner as any B-plus tree. A right child node of an entry contains the record value "greater than or equal to" its parent's entry, rather than just "greater than" as in regular B-tree structures.

Thus, in an inverted tree, all the record numbers reside in the leaves. The upper levels consist of the proper record numbers as the separators. The leaf nodes are linked together from left to right as a linked list and are referred to as the sequence set. The inverted tree can be utilized when a sequential process to retrieve all the records which have a specific key value. Each entry (i.e., record no.) in the leaf nodes is composed of only one value; the record number which points to a data record in the file. Each entry of a node in the internal levels of the tree contains a record number and pointer to its child node.
Node Description of Inverted E-plus Trees

The general node formats of the inverted E-plus tree are described by

**Internal node structure**

| N | LC | I_1 | RC_1 | I_2 | RC_2 | ... | In | RC_n |

**Leaf node structure**

| N | R_1 | V_1 | R_2 | V_2 | ... | R_n | V_n | LINK |

Figure 4  node description of inverted E-plus trees

where

N — number of entries in the node of inverted tree
LC — pointer to the left child node of inverted tree
I_i — the ith entry (a record number) in the internal node of inverted tree
RC_i — pointer to the right child node of the ith entry in an internal node of inverted tree
R_i — the ith record number in an inverted leaf which points to the corresponding record in the original file
V_i — the ith attribute value in an inverted leaf which associates with a record number in the original file
LINK — the link pointer to the right leaf node of inverted tree
Generation of the B-plus Tree

Using the technique of 'bottom-up' approach to construct the master B-plus tree and inverted directory, the steps to build a complete B-plus tree are as follows:

(1) Create all the leaf nodes. In this implementation we choose to put a minimum number of entries to facilitate updates. Thus, each leaf has a minimum number of entries and a link to the right leaf node.

(2) If there is more than one node on the current level of the tree, construct the next higher level of the tree (based on the number of nodes on this level) until there is only one node residing in the current level. This node becomes the root. Both master and inverted B-plus trees are constructed in the same manner.

Description of the Generation Procedure

The construction of the whole directory is as follows:

(1) Build all the inverted B-plus trees for the inverted directory itself, one by one, each corresponding to one specific key attribute value (i.e., we first read the sorted file to collect a group of identical attribute values.) We then use the record numbers which correspond to these attribute values to create an inverted B-plus tree. After this, we store the information (which includes this specific key attribute value and the pointer to its inverted B-plus tree) in a key information file. We continue this process until the end of the sorted file is reached.
(2) Based on all the different key attribute values, a B-plus tree for the master directory is built by reading the key information file (each record in this file contains a specific key attribute value and pointer to its corresponding inverted tree), and by using this information to create a master B-plus tree.
CHAPTER III

OPERATIONS OF THE DIRECTORY

Search Operation

Search operations for the master B-plus tree begin from the root and proceed through each level of the tree down to an appropriate leaf node. Since all the keys reside in the leaf nodes, it does not matter what values are encountered in the internal level as long as the path leads to the appropriate leaf. When this process reaches a leaf node, a check is made to see if a sought key exists in the current leaf. If the key is not found, an appropriate message is printed. Otherwise, a pointer to its corresponding inverted tree is obtained. The entire bottom level of this tree provides pointers to all records possessing this key (attribute) value.

Insertion Operation

Insert operations for the master and inverted B-plus trees of the directory are processed as for any B-plus tree. That is, whenever a leaf splits, the procedure prompts a copy of this entry as a separator in the internal level, retaining the actual key in the right leaf. This insertion in the parent node may in turn cause it to split, and so on up to the root. The tree may become one level deeper when the root itself splits. For range attributes, the same process is performed. Only difference is when the number of records
within a particular range have more than the maximum number of records per range. In order to maintain the uniform ranges, the splitting procedure is applied to divide this range into two new ranges; Thus, two new inverted trees which correspond to the new ranges on the master leaf are formed. The details are given in Appendix C (Program Outlines).

Assume a file which contains records need to be inserted to the employee file (which is listed in Table 1). These records are shown in Figure 5. Because all of the records have the identical attribute value, the master tree is maintained the same, all of the changes occur on an inverted tree which corresponds to this attribute value. The example for insertion process are shown in Figure 5.
1. Add record #15

```
  13
 /   \
\ 31
6 10 15
```

2. Add record #87

```
  13
 /   \
\ 31
6 10 15
```

3. Add record #57

```
  13
 /   \
\ 31
6 10 15
```

4. Add record #67

```
  13
 /   \
\ 31
6 10 15
```

5. Add record #70

```
  13
 /   \
\ 31
6 10 15
```

6. Add record #82

```
  13
 /   \
\ 31
6 10 15
```

7. Add record #73

```
  13
 /   \
\ 31
6 10 15
```

Figure 5 Tree Diagram of Insertion examples
Delete Operation

Deletion in the master and inverted B-plus trees of the directory is somewhat simplified as compared to the B-tree, due to its ability to leave non-key values (for the master tree) or non-record numbers (for the inverted tree) as separators on the internal levels. The rest of the deletion procedures are almost identical to the deletion procedures in a B-tree. Of course, if an underflow condition arises, the redistribution or concatenation procedures may require adjusting values in the internal level as well as in the leaf nodes. For range attributes, the same process is performed. The only difference is when the number of records within a particular range have less than the minimum number of records per range. In order to maintain the uniform ranges, the redistribution procedure is applied to distribute this range with the previous or next range; Thus, two inverted trees which correspond to the new ranges on the master leaf are involved to be reorganized. The details are given in Appendix C (Program Outlines).

Assume a file which contains the records need to be deleted from the employee file (which is listed in Table 1). These records are shown in Figure 6. Because all the records have the identical attribute value, the master tree is maintained the same, all of the changes occur on an inverted tree which corresponds to this attribute value. The example for deletion process are shown in Figure 6.
Follow the data after Insertion

1. Delete record #79

2. Delete record #73

3. Delete record #31

4. Delete record #63

underflow occurs, borrow entry from near sibling leaf

underflow occurs, combine with parent and a sibling node

underflow occurs, borrow entry from near sibling node

underflow occurs, combine with parent and a sibling node

Figure 6 Tree Diagram of Deletion examples
Consider a master B-plus tree of order m with n keys. The number of nodes at depth 0, 1, 2, ..., h must be at least $2^0, 2^1, 2^{(m/2)}, 2^{(m/2)}, \ldots, 2^{(m/2)^h}$, where h is the height of a tree.

Total number of nodes $= \frac{((m/2)^h - 1)}{((m/2) - 1)}$

and

Total number of keys $= \frac{((m/2)^h - 1)}{((m/2) - 1)}$

The height of a B-plus tree with N keys is therefore constrained by the following relation,

$\frac{(m/2)^h - 1}{(m/2)^h} \leq N$

$\frac{(m/2)^h - 1}{(m/2)^h} \leq N + 1$

$h \leq \log_d(N+1)$, where $d = \frac{m}{2}$

The tree then needs to have only $\log N$ level. This also applies to the inverted tree. If each inverted tree has order x with y record numbers, it will have the same number of nodes and levels as computed above. The search time is $O(l)$ where l is the number of levels. Insert or delete operations may need additional secondary storage accesses. The height of the tree still dominates the expression for these costs.

One successful application of the B-plus tree is IBM's general purpose B-plus based method, VSAM. VSAM is designed to support sequential searching as well as logarithmic cost of insert, delete and find operations. Compared to the conventional index-sequential
organization, the B-tree offers the following advantages:

(1) Dynamic allocation and release of storage.

(2) Guaranteed storage utilization of 50%

(3) No need for periodic "reorganization" of the entire file.

(4) Keeps a sorted list of keys at all times with no extra cost.

These advantages, which traditionally were applied only to dense attributes, are now applied to nondense attributes by using the hybrid structure.
CHAPTER IV

RUNNING PROCEDURE

A user runs the Uniform Data Base Directory using the command "RUN DBDIR1" for discrete attributes, or the command "RUN DBDIR2" for range attributes. When the program starts the execution process, it gives the user the following options:

ENTER ONE OF THE FOLLOWING FUNCTION TYPES:

(1) ENTER C for creation
(2) ENTER S for search
(3) ENTER D for deletion
(4) ENTER I for insertion
(5) ENTER E for exit from system

After the user chooses a valid option, the program will perform the indicated task. After this, the message "Do you want to see the current directory display on tty?" is printed to ask user's response. Two example runs are given here. The runs were generated using JOBLOG to show each step in the program execution.

The first example is for the discrete attributes. To start execution of this system, the user types "RUN DBDIR1" or "EXECUTE/C68 MAIN1.CBL,SORTED.CBL,CREATE.CBL,SEARCH.CBL,ADDDIR.CBL,DELDIR.CBL,PRINTS.CBL" from monitor mode. The JOBLOG run of this system is shown in page 72 - 89.

The second example is for the range attributes. Before this system starts to run, the user must enter "EXECUTE RANGES.PAS" in order to create the uniform ranges from the original file.
After this, the user types "RUN DBDIR2" or "EXECUTE/C68
MAIN2.CBL,RRANGE.CBL,CRANGE.CBL,SRANGE.CBL,ARANGE.CBL,DRANGE.CBL,
PRANGE.CBL" from monitor mode to start execution of this system.
The JOBLOG run of this example is shown in page 90 - 112.
CHAPTER V

CONCLUSION

This thesis has demonstrated the design and implementation of the uniform system for data base directories. First, a description of the structure was presented. Then, the performance of the operations and a set of algorithms were described to give the user an understanding of the theoretical aspects of the system. Finally, an actual run of the system using test files for both discrete and range attributes was demonstrated to achieve the tasks of creation, search, and update.

This implementation demonstrates the efficiency of the hybrid directory compared with other file organizations. For example, among the existing file organizations, the B-tree is both efficient and elegant and has gained wide acceptance. Unfortunately, the B-tree is deficient when non-dense attributes are involved. The hybrid directory makes up for this deficiency and provides fast retrieval of data. Also, the usual drawback of an inverted file which is cumbursome to update is overcome.

This approach has combined the best qualities of B-tree structure and inverted file structure to obtain the best directory. Especially to handle the non-dense attributes because it provides the uniform ranges no matter how the large and small clusters of data values are given. It has been shown that this directory is also adequate for dense attributes. It is important to keep one single type of directory for all files to facilitate efficient updating. Maintenance
of the data base is minimized when there is only one type of directory which needs to be created, updated, and maintained. Therefore, no matter what files exist in the data base, only one type of directory is needed, i.e. the hybrid B-plus tree and inverted file.

This directory facilitates fast searching, efficient updating and utilizes a minimum of 50% of the available storage. Thus the data base is easier to maintain and uniformity is achieved without loss of efficiency.
APPENDIX A

PROGRAM DOCUMENTATION
APPENDIX A  PROGRAM DOCUMENTATION

This project is designed to demonstrate how to create and update the hybrid B-plus tree with inverted files which we call the uniform system for data base directories. For the discrete attributes, the construction of the project consists of the main program and six subprograms (which are called directly from the main program.) The same design processes are applied to the range attributes, plus an additional process to manipulate the uniform ranges from the original file. The process flow of the system is illustrated in the following two diagrams:

Diagram 1: range attribute

Diagram 2: discrete attribute
The following is a listing of the program descriptions and the files which apply to the proper programs. In order to process the hybrid directory for the discrete attributes, the following programs and files are involved:

PROGRAM NAME: MAIN1.CBL

This is the main program for discrete attributes. When the system starts to run (i.e. after the user enters 'RUN DBDIR1' command from monitor level), four options are provided to the user through menu selection. These options are creation, search, insertion, and deletion of data in the hybrid directory. The constraint here is that the user should create the hybrid directory using the existing pile file before process updating. After the user selects a valid option, the program will call the appropriate subprogram to achieve the desired action. These subprograms are as follows:

SUBPROGRAM NAME: SORTED.CBL

This subprogram converts the original pile file (called ORIGIN.FIL) into a random file (called RANDOM.FIL). It then creates a file called SORTED.FIL which contains the record number and the corresponding discrete attribute pairs in ascending order by attribute value from the original file ORIGIN.FIL.
SUBPROGRAM NAME: CREATE.CBL

This subprogram creates the hybrid directory (called BPTREE.FIL) using the file SORTED.FIL.

SUBPROGRAM NAME: SEARCH.CBL

This subprogram searches an existing hybrid directory in BPTREE.FIL for the pointer (record number) to the original random file RANDOM.FIL. All records which satisfy the user request are then output to the TTY.

SUBPROGRAM NAME: ADDDIR.CBL

This subprogram inserts new records into the original file RANDOM.FIL and updates the existing hybrid directory BPTREE.FIL. The user can enter the records either interactively or through the use of an insertion file called INSERT.FIL.

SUBPROGRAM NAME: DELDIR.CBL

This subprogram deletes the records from the original file RANDOM.FIL and updates the existing hybrid directory BPTREE.FIL. The user can enter the record numbers either interactively or through the use of a deletion file called DELETE.FIL.
SUBPROGRAM NAME: PRINTS.CBL

This subprogram displays all the node records of the current hybrid directory BPTREE.FIL as a tree structure on the TTY.

In order to process the hybrid directory for the range attributes, the following programs and files are involved:

PROGRAM NAME: RANGES.PAS

This program must be executed (i.e. types 'EXECUTE RANGES.PAS' from monitor level) before using the processes for range attributes. The program creates the uniform ranges for the range attributes of the original file called ORANGE.FIL and stores the low and high attribute values of each range and their corresponding record numbers into an output file called INVITED.FIL.

PROGRAM NAME: MAIN2.CBL

This is the main program for range attributes. When the system starts to run (i.e. after the user enters 'RUN DBDIR2' command from monitor level ), four options are provided to the user through menu selection. These options are creation, search, insertion, and deletion of data in the hybrid directory. The constraint here is that the user
should create the hybrid directory using the existing inverted file INVTED.FIL before process updating. After the user selects a valid option, the program will call the appropriate subprogram to achieve the desired action. These subprograms are as follows:

SUBPROGRAM NAME: RRANGE.CBL

This subprogram reads the original pile file ORANGE.FIL and converts it to the random file (called RRANGE.FIL).

SUBPROGRAM NAME: CRANGE.CBL

This subprogram creates the hybrid directory (called BPTREE.FIL) using the file INVTED.FIL

SUBPROGRAM NAME: SRANGE.CBL

This subprogram searches an existing hybrid directory in BPTREE.FIL for the pointer (record number) to the original random file RRANGE.FIL. All records in the file RRANGE.FIL which satisfy the user request are then output to the TTY.
SUBPROGRAM NAME: ARANGE.CBL

This subprogram inserts new records into the original file RRANGE.FIL and updates the existing hybrid directory BPTREE.FIL. The user can enter the records either interactively or through the use of an insertion file called IRANGE.FIL.

SUBPROGRAM NAME: DRANGE.CBL

This subprogram deletes the records from the original file RRANGE.FIL and updates the existing hybrid directory BPTREE.FIL. The user can enter the record numbers either interactively or through the use of a deletion file called DRANGE.FIL.

SUBPROGRAM NAME: PRANGE.CBL

This subprogram displays all the node records of the current hybrid directory BPTREE.FIL as a tree structure on the TTY.
APPENDIX B

DESCRIPTION OF ALGORITHMS
Algorithm to Create the Hybrid Directory

PROGRAM CREATE_THE_HYBRID_DIRECTORY;

(* read the sorted file which contains the record no. and corresponding attribute value pairs, then create an inverted b-plus tree for each specific attribute value until the end of sorted file *)
read first record from the sorted file
attrval <- atrval_on_current_record
REPEAT
IF attrval # atrval_on_current_record
THEN
(* check if there are records which have not been written to leaf node yet, combine them to write last leaf node for proceed inverted tree *)
CALL CREATE_INTERNAL_NODES_OF_PROCEEDING_INVTD_TREE
attrval <- atrval_on_current_record
END IF
CALL STORE_NEW_REC_TO_CREATE_A_LEAF_OF_CUR_INV_TREE
read the next record from the sorted file
UNTIL end of the sorted file
CALL CREATE_INTERNAL_NODES_OF_LAST_INVTD_TREE

(* read the key information file which contains each key attribute value and pointer to its corresponding inverted tree. then create the master b-plus tree *)
CALL CREATE_MASTER_TREE_LEAF_LEVEL
# nodes on previous level <- # key values in key information file
WHILE # of nodes on previous level > 1
# nodes on curr level <- # nodes on prev level / min# entries
CALL CREATE_A_MASTER_TREE_LEVEL
# nodes on prev level <- # nodes on curr level
END WHILE
master root node <- file-key
rewrite the master tree information for the b-plus tree file
END OF PROGRAM;
Algorithm to Insert Records to Hybrid Directory

PROGRAM INSERT_TO_HYBRID_DIRECTORY;

(* first, search the master tree from the root level by level, until we reach the proper leaf of master tree. If new key value has not existed in this master leaf, insert it to proper entry position of current leaf.*)

WHILE not end of insert file DO

input a new record need to be inserted
CALL SEARCH_DOWNTO_A_MASTER-LEAF_FROM_ROOT

(* check if current master leaf node contains this key value; if current leaf does not contain this key, insert key value into the proper position in current leaf.*)

found <--- false
i <--- 0
WHILE not found OR i < # of entries in current master leaf DO
i <--- i + 1
if entry value of a master leaf(i) = attrval of a new record
THEN found <--- true
END WHILE

IF not found
THEN
# of entries on cur leaf <--- # of entries on cur leaf + 1
CALL INSERT_KEYVAL_TO_PROPER_ENTRY_IN_CUR_MASTER_LEAF
CALL CHECK_IF_A_NODE_ON_CUR_LEVEL_IS_OVERFLOW
END IF

(* finally, insert record number which is associated with this new key value to its corresponding inverted tree followed the same process as the master tree.*)

CALL SEARCH_DOWNTO_AN_INVTED_TREE_LEAF_LEVEL
CALL INSERT_RECORD_TO_PROPER_ENTRY_IN_CUR_INVTED_LEAF
CALL CHECK_IF_A_NODE_ON_CUR_LEVEL_IS_OVERFLOW

END WHILE

END OF PROGRAM;

SUBPROGRAM CHECK_IF_A_NODE_ON_CUR_LEVEL_IS_OVERFLOW;

WHILE # entries of cur node > max # entries AND not done DO
CALL SPLIT_A_NODE_INTO_TWON_CUR_LEVEL
read their parent node in one upper level
IF current node = root node of the tree
THEN done <--- true
END WHILE

if done
THEN CREATE_A_NEW_ROOT_NODE_FOR_CUR_TREE

END OF SUBPROGRAM;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Algorithm to Delete Records from Hybrid Directory

PROGRAM DELETE_FROM_HYBRID_DIRECTORY;

(* first, search master tree from the root, level by level, until the proper leaf of master tree is reached; find the proper key entry, then, search the inverted tree which corresponds to this key to find a specific record # and delete it from this inverted tree *)

WHILE not end of delete file DO

    read a record from the delete file
    CALL SEARCH_DOWNTO_A_MASTEC_LEAF
    read the record of current master_leaf_node
    found <-- false
    i <-- 0
    WHILE i < # entries in a master leaf OR not found DO
        i <-- i + 1
        IF entry_value(i) = atrval_of_a_deleted_record
           THEN found <-- true
    END WHILE
    IF found THEN
        CALL SEARCH_DOWNTO_AN_INVITED_LEAF
        CALL DELETE_AN_PROPER_ENTRY_FROM_CUR_INVITED_LEAF
        (* check if all of the nodes in current inverted tree are being deleted, then we continue to delete a key entry of a master leaf which the current inverted tree corresponds to *)
        IF # of nodes in current inverted tree = 0
           THEN
               CALL DELETE_FROM_MASTER_TREE (*same process as invted tree
           ELSE
               CALL CHECK_IF_A_NODE_ON_CUR_LEVEL IS_UNDERFLOW
        END IF
    END IF
END WHILE
END OF PROGRAM;

SUBPROGRAM CHECK_IF_A_NODE_ON_CUR_LEVEL IS_UNDERFLOW;

WHILE # entries in a node on cur level < min # of entries DO
    CALL CHECK_IF_NEAREST_SIBLING_NODE HAS ENOUGH ENTRIES
    IF not done
        THEN
            CALL COMBINE_WITH_NEAREST_SIBLING_AND_PARENT_NODE
        ELSE
            CALL TRANSFORM_AN_ENTRY_FROM_A_SIBLING_TO_CUR_NODE
    END IF
END WHILE
END OF SUBPROGRAM;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
PROGRAM UNIFORM_ATTRIBUTE_RANGES_PROCESS;

CALL GET_INPUT_FROM_FILE;
CALL SORT_ATTRIBUTE_VALUE;

// PROCESS TO SELECT INTERVALS OF MASTER DIRECTORY FOR INVERTED FILE //

WHILE USER_OPTION <> 5

// INITIALIZE THE MASTER DIRECTORY & LIST OF ATTRIBUTE VALUES //

FOR I <-- 1 TO 50
    FOR J <-- 1 TO 100
        MASDIR[I,J] <-- 0;
        TOTAL_ATOM_INTERVAL[I] <-- 0;

    IF (USER_OPTION > 0) AND (USER_OPTION < 5) THEN
        CASE USER_OPTION OF
            1 : CALL USER_CHOOSE_INTERVALS;
            2 : CALL PROGRAM_EQUALLY_DISTIBUTE_INTERVALS;
            3 : CALL PROGRAM MAKES_UNIFORM_INTERVALS;
            4 : CALL PROGRAM MAKES_UNIFORM_INTERVALS;
        END CASE

CALL PRINT_MASTER_DIRECTORY_TABLE;
END WHILE LOOP

END OF MAIN PROGRAM;
APPENDIX C

PROGRAM OUTLINES
PROGRAM UNIFORM_ATTRIBUTE_RANGES_PROCESS
CALL GET_INPUT_FROM_FILE;
CALL SORT_ATTRIBUTE_VALUE;
// PROCESS TO SELECT INTERVALS OF MASTER DIRECTORY FOR INVERTED FILE //
WHILE USER OPTION != 5
  // INITIALIZE THE MASTER DIRECTORY & LIST OF ATTRIBUTE VALUES //
  FOR I = 1 TO 50
    FOR J = 1 TO 100
      MASDIR[I,J] = 0;
      TOTAL_ATR_OF_INTVAL[I] = 0;
    IF (USER OPTION > 0) AND (USER OPTION < 5) THEN
      CASE USER OPTION OF
        1 : CALL USER_CHOOSE_INTERVALS;
        2 : CALL PROGRAM_EQUALLY_DISTRIBUTE_INTERVALS;
        3 : CALL PROGRAM_MAKES_UNIFORM_INTERVALS;
        4 : CALL PROGRAM_MAKES_UNIFORM_INTERVALS;
      END CASE
    CALL PRINT_MASTER_DIRECTORY_TABLE;
  END WHILE LOOP
END OF PROGRAM;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%% PROCEDURE USER_CHOOSE_INTERVALS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// THIS PROCEDURE IS TO LET USER TO SELECT THE RANGE OF EACH INTERVAL FOR MASTER DIRECTORY //
N = 0;
I = 0;
J = 1;
REPEAT
  N = N + 1;
  IF (LV <> C) OR (HV <> 0) THEN
    I = I + 1;
    MASDIR[I,J] = LV;
    MASDIR[I,J] = HV;
    K = 0;
    WHILE (SORTED_ATR_LIST[J,K] = RV) AND (J = NR)
      K = K + 1;
    TOTAL_ATR_OF_INTVAL[I] = K;
    REAUSER_RESPONSE;
    IF USER RESPONSE = 'N' THEN
      TOTAL_ATR_OF_INTVAL[I] = 0;
      MASDIR[I] = 0;
      MASDIR[I] = 0;
    UNTIL (LV = 0) AND (HV = 0)
  END USER_CHOOSE_INTERVALS;
PROCEDURE PROGRAM_EQUALLY_DISTRIBUTE_INTERVALS;

PRINT("ENTER THE TOTAL NUMBER OF INTERVALS YOU WANT > ");
READ(N1);

NRI <- NR DIV N1;
IF NR MOD N1 <> 0 THEN NRI <- NRI + 1;
POS_OF_LAST_ATR_LST_INTERVAL <- 0;
FOR I <- 1 TO N1-
  TOTAL_INTERVALS[I] <- NRI;
  MASDIR[I,1] <- SORTED_ATR_LIST[POS_OF_LAST_ATR_LST_INTERVAL+1,23];
  MASDIR[I,2] <- SORTED_ATR_LIST[POS_OF_LAST_ATR_LST_INTERVAL+NRI+23];
  J <- 2;
  FOR K <- POS_OF_LAST_ATR_LST_INTERVAL+1 TO POS_OF_LAST_ATR_LST_INTERVAL+NRI
    J <- J + 1;
    MASDIR[I,J] <- SORTED_ATR_LIST[K,13];
    POS_OF_LAST_ATR_LST_INTERVAL <- POS_OF_LAST_ATR_LST_INTERVAL + NRI;
  ENDFORK;
  MASDIR[I,1] <- SORTED_ATR_LIST[POS_OF_LAST_ATR_LST_INTERVAL+1,23];
  MASDIR[I,2] <- SORTED_ATR_LIST[N/R,23];
  J <- 2;
  FOR K <- POS_OF_LAST_ATR_LST_INTERVAL+1 TO NR
    J <- J + 1;
    MASDIR[I,J] <- SORTED_ATR_LIST[K,13];
    TOTAL_ATR_CF_INTERVAL[I] <- J - 2;
ENDFOR
END PROGRAM_EQUALLY_DISTRIBUTE_INTERVALS;

PROCEDURE COMBINE_OR_REDISINSERT_TWO_INTERVALS;

// THIS PROCEDURE CHECKS PREVIOUS 2 INTERVALS TO SEE IF THEY CAN COMBINE OR RECONVISE IN ORDER TO HAVE UNIFORM INTERVALS //

// CHECK IF THERE EXISTS TWO INTERVALS OR IF TWO INTERVALS SHOULD COMBINE OR RECONVISE //

IF (CUR_INTERVAL > 2) AND (CUR_INTERVAL - 1 < DELETED_INTERVAL) THEN
  FIRST <- CUR_INTERVAL - 2;
  SECOND <- CUR_INTERVAL - 1;
  SUM <- TOTAL_ATR_CF_INTERVAL[FIRST] + TOTAL_ATR_CF_INTERVAL[SECOND];

// CHECK IF THESE TWO INTERVALS CAN COMBINE SO THAT BOTH INTERVAL HAVE UNIFORM FREQUENCIES //

IF (SUM <= MAX_NRI) AND (SUM > 2) THEN
  MASDIR[FIRST,2] <- MASDIR[SECOND,23];
  MASDIR[SECOND,13] <- 0;
  MASDIR[SECOND,23] <- 0;
  EXPAND <- 2;
  FOR K <- TOTAL_ATR_CF_INTERVAL[FIRST]+3 TO SUM+2
    EXPAND <- EXPAND + 1;
    MASDIR[FIRST,K] <- MASDIR[SECOND,EXPAND];
    TOTAL_ATR_CF_INTERVAL[SECOND] <- 0;
    TOTAL_ATR_CF_INTERVAL[FIRST] <- SUM;
    CUR_INTERVAL <- CUR_INTERVAL - 12;
ELSE
  // CHECK IF THESE TWO INTERVALS CAN RECONVISE SO BOTH INTERVAL CAN HAVE UNIFORM FREQUENCIES //

IF TOTAL_ATR_OF_INTERVAL(SECOND) <= SUM DIV 2
THEN
  REPEAT
    NO_OF_EQUAL_ATRVALS <= 1;
    EXPAND <= TOTAL_ATR_OF_INTERVAL(FIRST) + 2;
    ATRVAL <= ORIGIN_ATR_LIST[MASSID][FIRST][EXPAND+2];
    NE <= FALSE;
    EXPAND <= EXPAND + 2;
  WHILE (EXPAND > 2) AND (NE = FALSE)
  IF ORIGIN_ATR_LIST[MASSID][FIRST][EXPAND+2] = ATRVAL
  THEN
    NO_OF_EQUAL_ATRVALS <= NO_OF_EQUAL_ATRVALS + 1;
    EXPAND <= EXPAND + 2;
  ELSE NE <= TRUE;
  OLD_DIFFER <= TOTAL_ATR_OF_INTERVAL(FIRST) - TOTAL_ATR_OF_INTERVAL(SECOND);
  FIRST_INTERVAL_NEW_TOTAL_ATR <= TOTAL_ATR_OF_INTERVAL(FIRST) - NO_OF_EQUAL_ATRVALS;
  SECOND_INTERVAL_NEW_TOTAL_ATR <= TOTAL_ATR_OF_INTERVAL(SECOND) + NO_OF_EQUAL_ATRVALS;
  NEW_DIFFER <= AABS(FIRST_INTERVAL_NEW_TOTAL_ATR - SECOND_INTERVAL_NEW_TOTAL_ATR);
  IF OLD_DIFFER - NEW_DIFFER > 0
  THEN
    MASSID[SECOND][3] <= ORIGIN_ATR_LIST[MASSID][FIRST][EXPAND+2];
    MASSID[SECOND][13] <= ORIGIN_ATR_LIST[MASSID][FIRST][EXPAND+13];
    TOTAL_ATR_OF_INTERVAL(FIRST) <= FIRST_INTERVAL_NEW_TOTAL_ATR;
    TOTAL_ATR_OF_INTERVAL(SECOND) <= SECOND_INTERVAL_NEW_TOTAL_ATR;
    EXPAND <= 2;
    FOR N <= NO_OF_EQUAL_ATRVALS+3 TO SECOND_INTERVAL_NEW_TOTAL_ATR+2
    EXPAND <= EXPAND + 2;
    MASSID[SECOND][13] <= MASSID[SECOND][EXPAND];
    EXPAND <= FIRST_INTERVAL_NEW_TOTAL_ATR + 2;
    FOR N <= 3 TO NO_OF_EQUAL_ATRVALS+2
    EXPAND <= EXPAND + 1;
    MASSID[SECOND][13] <= MASSID[FIRST][EXPAND];
  FOR N <= FIRST_INTERVAL_NEW_TOTAL_ATR+3 TO FIRST_INTERVAL_NEW_TOTAL_ATR+NO_OF_EQUAL_ATRVALS+2
  MASSID[SECOND][13] <= 0;
UNTIL OLD_DIFFER - NEW_DIFFER < 0;
END COMBINE_OR_REDUPLICATE_TWO_INTERVALS;

//**********************************************************************************************

PROCEDURE FIT_ATTRIBUTES_INTO_CURRENT_INTERVALS;
// THIS PROCEDURE IS TO FIT "PROPER" NUMBER OF ATTRIBUTE VALUES INTO THE CURRENT INTERVAL OF MASTER DIRECTOTY //
POS_OF_LIST_ATR_CUR_INTERVAL <= POS_OF_LIST_ATR_LIST_INTERVAL + 12;
POS_OF_LIST_ATR_CUR_INTERVAL <= POS_OF_LIST_ATR_LIST_INTERVAL + EXPAND;
IF POS_OF_LIST_ATR_CUR_INTERVAL > NR
THEN POS_OF_LIST_ATR_CUR_INTERVAL <= NR;
MASSID[CURRENT][13] <= SORTED_ATR_LIST[POS_OF_LIST_ATR_CUR_INTERVAL+2];
MASSID[CURRENT][23] <= SORTED_ATR_LIST[POS_OF_LIST_ATR_CUR_INTERVAL+2];
N <= 2;
FOR L <- POS_OF_LST_ATR_CUR_INTERVAL TO POS_OF_LST_ATR_CUR_INTERVAL
M <- M + 1;
MASDIRCUR_INTERVAL <- SORTED_ATR_LIST[L];
TOTAL_ATR_CF_INTERVALCUR_INTERVAL2 <- M - 2;
CUR INTERVAL <- CUR INTERVAL + 1;
END FIT_ATTRIBUTES INTO CURRENT INTERVALS;

 kostenlose Reproduktionsverbotung. Weitere Reproduktion ohne Genehmigung ist verboten.

procedure SPLIT_ATTRIBUTES INTO TWO INTERVALS;

  // THIS PROCEDURE IS TO CHECK HOW MANY ATTRIBUTE VALUES CAN PUT IN CURRENT INTERVAL,
  LEAVE REST EQUAL VALUES AT BEGINNING OF NEXT INTERVAL */

DONE <- FALSE;
EXPAND <- EXPAND - 1;
TOTAL_ATR_CF_INTERVALCUR_INTERVAL2 <- TOTAL_ATR_CF_INTERVALCUR_INTERVAL2 - 1;
NO_OF_EQUAL_ATRVALS <- 1;

WHILE (SORTED_ATR_LIST[POS_OF_LST_ATR_LST_INTERVAL+EXPAND*1,23] = SORTED_ATR_LIST[POS_OF_LST_ATR_LST_INTERVAL+EXPAND+1,23])
AND (DONE = FALSE)
NO_OF_EQUAL_ATRVALS <- NO_OF_EQUAL_ATRVALS + 1;
TOTAL_ATR_CF_INTERVALCUR_INTERVAL2 <- TOTAL_ATR_CF_INTERVALCUR_INTERVAL2 - 1;
EXPAND <- EXPAND - 1;

IF EXPAND = 0
THEN DONE <- TRUE;  // CURRENT INTERVAL HAS TOO MANY ATTRIBUTE VALUES
IF USER DOESN'T WANT PUT IN DIRECTORY IT WILL BE EXHAUSED //

IF DONE = FALSE
THEN CALL FIT_ATTRIBUTES INTO CURRENT INTERVAL;
CALL COMBINE OR REDISTRIBUTE TWO INTERVALS;

IF NO_OF_EQUAL_ATRVALS > MAX NRI THEN
PRINT("THE ATTRIBUTE VALUE NRI'S OCCURS \"NO_OF_EQUAL_ATRVALS\" TIMES");
PRINT("YOU WANT THIS VALUE INCLUDED IN THE DIRECTORY (ENTER Y OR N)?");
READUSER_RESPONSES;
IF USER_RESPONSES = "Y"
THEN
EXPAND <- NO_OF_EQUAL_ATRVALS;
CALL FIT_ATTRIBUTES INTO CURRENT INTERVAL;
ELSE
POS_OF_LST_ATR_LST_INTERVAL <- NO_OF_EQUAL_ATRVALS + POS_OF_LST_ATR_LST_INTERVAL;
DELETE INTERVAL <- CUR INTERVAL;
END SPLIT_ATTRIBUTES INTO CURRENT INTERVALS;

procedure PROGRAM MAKES_UNIFORM INTERVALS;

  // THIS PROCEDURE IS TO DESIGN TO MAKE THE INTERVAL UNIFORM //

  // INTERACT WITH USER PROCESS, EITHER USER ENTER AVERAGE, MINIMUM OR MAXIMUM NUMBER OF RECORDS PER INTERVAL
  OR PROGRAM GENERATE THESE VALUES //

PRINT("ENTER THE AVERAGE NUMEROF RECORDS PER INTERVAL? ");
READ(NRI);
IF NRI < C
THEN Ni <- NR DIV NRI
ELSE
year <- round(sort(year));
year <- n;

print('enter the minimum number of records per interval ?');
read(min_nri);
if min_nri = 0 then
  min_nri <- nri div 2;
  print('the program generates minimum number of records per interval is \"min_nri\');
  print('enter the maximum number of records per interval ?');
  read(max_nri);
  if (max_nri = 0) or (max_nri < 2 * min_nri) then
    if max_nri = 0
      max_nri <- 2 * n;
    else max_nri <- 2 * min_nri;
    print('the program generates maximum number of records per interval is \"max_nri\');
  // initialize the desired number of attribute values per interval //
  for i = 1 to n;
    total_atr_of_interval[i] <- nri;
  // initialize the position of last attribute value of previous interval to zero to start with //
    pos_of_lst_atr_lst_intval <- 0;
    cur_intval <- 1;
  // the process to combine the attribute values into intervals with uniform frequencies //
    while pos_of_lst_atr_lst_intval < nr
      expand <- nri;
      // check if the last attribute value of current interval is equal to the first attribute value of next interval //
      if (sorted_atr_list[pos_of_lst_atr_lst_intval+expand] = sorted_atr_list[pos_of_lst_atr_lst_intval+expand+1] 
        or (pos_of_lst_atr_lst_intval + expand = nr)
        then call fitattributes_into_current_interval
      else
        // count how many equal attribute values at end of current interval together with begin of next interval //
        while sorted_atr_list[pos_of_lst_atr_lst_intval+expand] = sorted_atr_list[pos_of_lst_atr_lst_intval+expand+1] 
          expand <- expand + 1;
      end while loop
      m1 <- sorted_atr_list[pos_of_lst_atr_lst_intval+expand] 
      // check if the desired number of attribute values plus these equal values can fit in the current interval //
      if total_atr_of_interval[cur_intval] = max_nri
        then call split_attributes_into_current_interval
      else call split_attributes_into_two_intervals
      end while loop
      // check if the last interval needs to be combine or redistribute with the previous interval //
      call combine_or_redistribute_two_intervals
end program makes_uniform_intervals
PROGRAM CREATE-THE-HYBRID-DIRECTORY

//read the sorted file which contains the record no. & corresponding attribute value pairs
//then create an inverted b-plus tree for each specific attribute value until the end of sorted file.

read 1st record from the sorted file
MOVE NEW-ATRVAL TO CURRENT-ATRVAL
PERFORM CREATE-EACH-INVITED-TREE UNTIL END-OF-SORTED-FILE

//complete an inverted b-plus tree for the last attribute value.
//firstly check if there are more records which have not been written to the leaf, combine them to last leaf
//then create the internal nodes to complete this last inverted tree.

PERFORM BUILD-INTER-NODES-OF-CUR-ITREE

//read the key information file which contains each key attribute value
//and pointer to its corresponding inverted tree, then create the master b-plus tree.

PERFORM CREATE-MASTER-TREE-PROCESS

END OF PROGRAM;

PROCEDURE CREATE-EACH-INVITED-TREE

//check if a new record you read has the same attribute value as the current attribute value,
//if so, store it and create the leaf node for the current inverted tree until a new record with
//a different attribute value occur, based on these leaves, create all the internal nodes
//to complete this inverted tree, then store the new record for a first leaf of next inverted tree to use.

IF NEW-ATRVAL NOT EQUAL TO CURRENT-ATRVAL
    THEN
        check if more records have not been written to the leaf yet, combine to the last leaf of current inverted tree
        PERFORM BUILD-INTER-NODES-OF-CUR-ITREE
        MOVE NEW-ATRVAL TO CURRENT-ATRVAL
    END IF

PERFORM STORE-NEW-REC-FOR-INVITED-LEAF

//read the next record from the sorted leaf

END OF PROCEDURE;

PROCEDURE STORE-NEW-REC-FOR-INVITED-LEAF

//store one more record ready for moving to a leaf node until
//the number of the records you stored is equal to minimum number of entries for a node
//write to a leaf node for current inverted tree.

ADD 1 TO I-NUM-OF-RECS
MOVE NEW-RECNO TO I-REC-NUM(I-NUM-OF-RECS)
IF I-NUM-OF-RECS = MIN-ENTRIES
    THEN write a leaf node for current inverted tree
END IF

END OF PROCEDURE;
PROCEDURE BUILD-INTER-NOSES-OF-CLUT-ITREE;
//use all of the leaf nodes of the current inverted tree to create the internal nodes, level by level,
//until there is only one node left in a specific level, this node becomes the root of this inverted tree.
//maintain all the information to key information file which includes the key attribute values
//number of levels = number of nodes on each level and pointer to the root of current inverted tree.

PERFORM CREATE-A-INVITED-INTERNAL-LEVEL VARYING LEVEL FROM 1 BY 1 UNTIL NUM-OF-NODES-IN-CURRENT-LEVEL = 1
   write a record to the key information file
END OF PROCEDURES;

PROCEDURE CREATE-A-INVITED-INTERNAL-LEVEL;
//use the number of nodes in previous level to compute the number of nodes in current level,
//then create each node in current level by reading minimum number of children nodes every time from previous level.

DIVIDE NUM-OF-NODES-IN-PREVIOUS-LEVEL BY MIN-CHILDREN GIVING NUM-OF-NODES-IN-CURRENT-LEVEL,
REMAINDER NODES-LEFT-IN-PREVIOUS-LEVEL

//create all the internal nodes except the last one for the current level of an inverted tree.
IF NUM-OF-NODES-IN-CURRENT-LEVEL > 0
THEN
   MOVE MIN-ENTRIES TO M
   N = NUM-OF-NODES-IN-CURRENT-LEVEL - 1
   PERFORM CREATE-A-INVITED-INTERNAL-NODE N TIMES
END IF

//the rest of the statements in the procedure are the process of the last internal node for current level, check if
//there are any remaining nodes from the previous level besides the nodes have been divided for last node in current level.
//combine them with the ones already divided to last node in current level to create this last node in current level.
//if no node resides in current level yet, create the only node for current level using all the nodes of previous level.
//this node will be the root of this inverted tree.
IF NODES-LEFT-IN-PREVIOUS-LEVEL > 0
THEN
   IF NUM-OF-NODES-IN-CURRENT-LEVEL > 0
      MOVE NODES-LEFT-IN-PREVIOUS-LEVEL TO M
   ELSE
      MOVE 1 TO NUM-OF-NODES-IN-CURRENT-LEVEL
   END IF
   N = NODES-LEFT-IN-PREVIOUS-LEVEL - 1
END IF
PERFORM CREATE-A-INVITED-INTERNAL-NODE
END OF PROCEDURES;

PROCEDURE CREATE-A-INVITED-INTERNAL-NODE;
//feed the minimum number of children nodes from previous level each time
//to create one node for the current level of the inverted tree.
ADD 1 TO PROCESSING-NODE-IN-PREV-LEVEL
MOVE PROCESSING-NODE-IN-PREV-LEVEL TO I-PY-TO-CHILD
PERFORM FILL-DATA-TO-A-INV-INTERNAL-NODE VARYING I FROM 1 BY 1 UNTIL I > M
   write an internal node for current level of an inverted tree
END OF PROCEDURES;
PROCEDURE FILL-DATA-TO-A-INV-INTERN-NODE;
//move the left most entry value from the corresponding leaf to the proper entry on the current node.
//then move the proceeding node no. in the previous level to the right child position on the current node.

ADD 1 TO PROCEEDING-NODE-IN-PREV-LEVEL, I-NUM-OF-ENTRIES
//search down to the leaf level following the proper pointers.
//read an appropriate leaf node of the inverted tree
//then move the first entry value on the corresponding leaf to the proper entry position on current node.
MOVE I-REC-VAL(I) TO I-REC-NUM(I)
//check if the previous level is the leaf level, if so, move the corresponding leaf no., else,
//move the proceeding node no. in previous level to proper pointer to right child position of the current node.
IF LEVEL = 1
    MOVE FILE-KEY TO I-PT-TO-RCHILDO(I)
ELSE
    MOVE PROCEEDING-NODE-IN-PREV-LEVEL TO I-PT-TO-RCHILDO(I)
ENDIF

END OF PROCEDURE;

PROCEDURE CREATE-MASTER-TREE-PROCESS;
//read the key-info-file which contains each specific key value & pointer to its corresponding inverted tree.
//use them to create all the leaf nodes for the master tree.
//then build the internal levels, level by level, until only one node resides in a specific level.
//this node is the root of master tree.

MOVE 1 TO NUM-OF-LEVELS
PERFORM CREATE-MASTER-LEAF-LEVEL
PERFORM CREATE-MASTER-INTERNAL-LEVEL VARYING LEVEL FROM 1 BY 1 UNTIL NUM-OF-NODES-IN-CURRENT-LEVEL = 1
MOVE FILE-KEY TO PT-TO-ROOT
write the master information record for the b-plus tree file

END OF PROCEDURE;

PROCEDURE CREATE-MASTER-LEAF-LEVEL;
//use the total number of key attribute values to compute the number of leaf nodes for the master tree.
//then create each leaf node by reading minimum number of key values each time from key information file.

DIVIDE TOTAL-NUM-OF-KEYVAL BY MIN-ENTRIES GIVING NUM-OF-NODES-IN-CURRENT-LEVEL
REMAINDER KEYS-LEFT-IN-KEY-INFO-FILE
//create all the leaf nodes except the last one for the master tree.
IF NUM-OF-NODES-IN-CURRENT-LEVEL > 1
    THEN
        MOVE MIN-ENTRIES TO N
        N = NUM-OF-NODES-IN-CURRENT-LEVEL - 1
        PERFORM CREATE-MASTER-LEAF-NODE N TIMES
    END IF
/* the rest of the statements in the procedure are the process of the last leaf node. */
/* if more key values are left in the key information file besides the ones have been divided for the last leaf, then */
/* combine them with the ones already divided for last leaf to create this last leaf. */
/* if no leaf resides in current level yet, create the only leaf node using all the key values from key information file. */
/* this only leaf node will be the root of this master tree. */

IF KEYS-LEFT-IN-KEY-INFO-FILE > 0 THEN
  IF NUM-OF-NODES-IN-CURRENT-LEVEL > 0 THEN
    ADD KEYS-LEFT-IN-KEY-INFO-FILE TO N
  ELSE MOVE 1 TO NUM-OF-NODES-IN-CURRENT-LEVEL
END IF
END IF
PERFORM CREATE-A-MASTER-LEAF-NODE

END OF PROCEDURE;

PROCEDURE CREATE-A-MASTER-LEAF-NODE;
/* read the minimum number of key values from key information file each time */
/* to create one leaf node of the master tree. */
PERFORM FILL-DATA-TO-A-MASTER-LEAF VARYING I FROM 1 BY 1 UNTIL I > N
write a leaf node for the master tree

END OF PROCEDURE;

PROCEDURE FILL-DATA-TO-A-MASTER-LEAF;
ADC 1 TO NUM-OF-KEYS
read a record from the key-info-file
MOVE KEY-INFO-REC TO KEY-INFO(I)

END OF PROCEDURE;

PROCEDURE CREATE-A-MASTER-INTERNAL-NODE;
/* use the number of nodes in previous level to compute the number of nodes in current level */
/* then create each node in this level by reading the minimum number of child nodes in previous level each time. */
DIVIDE NUM-OF-NODES-IN-CURRENT-LEVEL BY MIN-CHILDREN GIVING NUM-OF-NODES-IN-CURRENT-LEVEL,
REMAINDER NODES-LEFT-IN-PREVIOUS-LEVEL

/* create all the internal nodes except the last one for the current level of the master tree. */
IF NUM-OF-NODES-IN-CURRENT-LEVEL > 0 THEN
  MOVE MIN-ENTRIES TO N
  N = NUM-OF-NODES-IN-CURRENT-LEVEL - 1
  PERFORM CREATE-A-MASTER-INTERNAL-NODE N TIMES
END IF
// the rest of the statements in this procedure are the process of the last internal node for current level.
// first, if more nodes left in previous level besides the one already divided for last node in current level, then
// combine them with the ones already divided to last node in current level to create this last node in current level.
// if no node resides in current level yet, create the only node for current level using all the nodes in previous level
// this only node will be the root of this master tree.
IF NODES-LEFT-IN-PREVIOUS-LEVEL > 0
THEN
  IF NUM-OF-NODES-IN-CURRENT-LEVEL > 0
  THEN
    ADD NODES-LEFT-IN-PREVIOUS-LEVEL TO M
  ELSE
    MOVE 1 TO NUM-OF-NODES-IN-CURRENT-LEVEL
    M = NODES-LEFT-IN-PREVIOUS-LEVEL - 1
  END IF
END IF
PERFORM CREATE-A-MASTER-INTERNAL-NODE
END OF PROCEDURE;

PROCEDURE CREATE-A-MASTER-INTERNAL-NODE;
// read the minimum number of children nodes from previous level each time
// to create one node on the current level of the master tree.
ADD 1 TO PROCCEEDING-NODE-IN-PREV-LEVEL
MOVE PROCCEEDING-NODE-IN-PREV-LEVEL TO PT-TC-CHILD
PERFORM FILL-DATA-TO-A-M-INTERNAL-NODE VARYING I FROM 1 BY 1 UNTIL I > M.
write an internal node for the master tree
END OF PROCEDURE;

PROCEDURE FILL-DATA-TO-A-M-INTERNAL-NODE;
// move the corresponding leaf to the proper entry on the current node, then
// move the proceeding node no. in the previous level to the right child position on the current node.
ADD 1 TO PROCCEEDING-NODE-IN-PREV-LEVEL, NUM-OF-ENTRIES
// search down to the leaf level by following the proper pointers.
read an appropriate leaf node of the master tree
// then move the first key value on the corresponding leaf to proper entry position on current node.
MOVE KEY=VAL(I) TO ZR=VAL(I)
// if the previous level is a leaf level, move the corresponding leaf no. else,
// move the proceeding node no. in previous level to current pointer in the right child position of the current node
IF LEVEL = 1
  THEN
    MOVE FILE=KEY TO PT-TO-CHILD(I)
  ELSE
    MOVE PROCEEDING-NODE-IN-PREV-LEVEL TO PT-TC-CHILD(I)
  END IF
END OF PROCEDURE;
PROGRAM INSERT-NEW-RECORD-TO-HYBRID-DIRECTORY;

//first search the master tree from the root level by level until we reach the proper leaf of master tree
//if new key value has not existed in this master leaf, insert it to proper entry position of current leaf
//then insert the record no. which associates with this key value to its corresponding inverted tree

read a new record no. to be inserted to the b+tree file
MOVE PT-TO-ROOT TO FILE-KEY
PERFORM SEARCH-DOWNTO-A-MASTER-LEAF VARYING I FROM 1 BY 1 UNTIL I = NUM-OF-LEVELS

//check if current leaf node contains this new key value,
//if current leaf doesn't contain this key, insert into the proper entry position on current leaf
MOVE 0 TO CONE, FOUND
PERFORM CHECK-IF-LEAF-HAS-NEW-KEYVAL VARYING J FROM 1 BY 1 UNTIL J = NUM-OF-KEYS OR DONE = 1 OR FOUND = 1
IF FOUND = 0
PERFORM INSERT-NEW-KEYVAL-TO-CUR-LEAF

//finally insert record no. which is associated with this new key value to its corresponding inverted tree
PERFORM INSERT-TO-INVTEC-TREE-PROCESS
END OF MAIN PROGRAM;

PROCEDURE SEARCH-DOWNTO-A-MASTER-LEAF;

//recursively search the proper node in current level, store this node no.,
//first check and then store the proper entry position in the node, finally,
//move the proper pointer to child node to file-key in order to read the appropriate child node next time

read an internal node record from the master tree contained in the b+tree file
MOVE 0 TO CONE
ADD 1 TO LEVEL-NO
MOVE FILE-KEY TO NICE_SIZE(LEVEL-NO)
PERFORM CHECK-POS-CN-CUR-MASTER-NODE VARYING J FROM 1 BY 1 UNTIL J = NUM-OF-ENTRIES OR DONE = 1
IF DONE = 0
MOVE PT-TO-CHILD(NUM-OF-ENTRIES) TO_FILE-KEY
MOVE J TO POSITION-IN-A-NODE(LEVEL-NO)
END OF PROCEDURE;

PROCEDURE CHECK-POS-CN-CUR-MASTER-NODE;
IF NEW-ATVAL < ATVAL(J)
MOVE J TO POSITION-IN-A-NODE(LEVEL-NO)
MOVE 1 TO DONE
IF J = 1
MOVE PT-TO-CHILD TO_FILE-KEY
ELSE
MOVE PT-TO-CHILD(J - 1) TO_FILE-KEY
END OF PROCEDURE;

PROCEDURE CHECK-IF-LEAF-HAS-NEW-KEYVAL;
IF NEW-ATVAL = KEY-VAL(J)
MOVE 1 TO FOUND
MOVE J TO ENTRY-POS-IN-A-MASTER-LEAF

END OF PROCEDURE;

END OF PROGRAM;
ELSE
  IF NEW-ATRVAL < KEY-VAL(j)
    MOVE 1 TO DONE
END OF PROCEDURE;

PROCEDURE INSERT-NEW-KEYVAL-TO-CLF-LEAF;
//shift a certain key entry value to the right in order to empty the current entry position
//later we can store new key value in the current entry position of the master leaf node
\[ M = NUM-OF-KEYS - J + 1 \]
PERFORM SHIFT-KEYS-IN-CUR-LEAF \[ M \] TIMES
MOVE 1 TO NUM-OF-KEYS
MOVE NEW-ATRVAL TO KEY-VAL(j)
rewrite the current master leaf node
IF NUM-OF-KEYS > MAX-ENTRIES
PERFORM SPLIT-A-MASTER-LEAF-PROCESS
ELSE
MOVE FILE-KEY TO NEW-KEY-LOCATES-LEAFNO
MOVE J TO ENTRY-POS-IN-A-MASTER-LEAF
END OF PROCEDURE;

PROCEDURE SPLIT-A-MASTER-LEAF-PROCESS;
//split current leaf node into two nodes, promote the image of the middle key value to next level
\[ N = \text{NUM-OF-KEYS} / 2 \]
MOVE KEY-VAL(N) TO PROMOTE-ATRVAL
MOVE AVAL-RECNO TO NEW-NODEEND
MOVE FILE-KEY TO CURRENT-NODEEND
MOVE N TO \[ M \]
//first, we use the 2nd half of the key values from current leaf to create a new leaf
PERFORM WRITE-KEYS-FOR-SPLIT-2ND-LEAF VARYING L FROM 1 BY 1 UNTIL \[ L > \text{NUM-OF-KEYS} \]
MOVE NEW-NODEEND TO FILE-KEY
PERFORM CHECK-IF-NEW-KEY-IN-CURNT-LEAF VARYING L FROM 1 BY 1 UNTIL \[ L > \text{NUM-OF-KEYS} \]
//then we rewrite the current leaf node by removing the 2nd half of the key values from the current leaf
MOVE CURRENT-NODEEND TO KEY-KEY
PERFORM WRITE-KEYS-FOR-SPLIT-1ST-LEAF VARYING L FROM \[ N \] BY 1 UNTIL \[ L > \text{NUM-OF-KEYS} \]
PERFORM CHECK-IF-NEW-KEY-IN-CURNT-LEAF VARYING L FROM 1 BY 1 UNTIL \[ L > \text{NUM-OF-KEYS} \]
//finally, we promote the image of the middle key value from the split leaf to internal levels
MOVE 0 TO DONE
PERFORM PROMPT-ATR-TO-M-INTERN-LEVELS UNTIL \[ \text{DONE} = 1 \] OR \[ \text{LEVEL-NO} = 0 \]
IF DONE = C AND \[ \text{LEVEL-NO} = 0 \]
PERFORM CREATE-NEW-MASTER-ROOT
END OF PROCEDURE;

PROCEDURE PROMPT-ATR-TO-M-INTERN-LEVELS;
//put the image of the middle key from the split node on previous level
//to the proper entry position of its parent node on current level
MOVE NODENLEVEL-AD TO FILE-KEY
MOVE POSITION-IN-A-NODELEVEL-AD TO J
N = NUM-OF-ENTRIES - J + 1

//shift a certain entry values to the right in order to empty the current entry position
//later, we can store image of the middle attribute from the split node (previous level) into current entry position
PERFORM SHIFT-ATRN-IN-CLR=NODE # TIMES
ADD 1 TO NUM-OF-ENTRIES
MOVE PROMOTE-ATRVAL TO ATR-VAL(J)
MOVE NEW-NODENC TO PT-TO-CHILD(J)
retry the current master internal node
IF NUM-OF-ENTRIES > MAX-ENTRIES
PERFORM SPLIT-A-MASTER-INTERNAL-NODE
ELSE
MOVE 1 TO DONE
END OF PROCEDURE.

PROCEDURE SPLIT-A-MASTER-INTERNAL-NODE;
//split current internal node into two nodes, promote the image of the middle attribute value to next level
N = NUM-OF-ENTRIES / 2
MOVE ATR-VAL(N) TO PROMOTE-ATRVAL

//first, we use the 2nd half of the attribute values from current node to create a new node
MOVE NEW-NODENC TO FILE-KEY
PERFORM WRITE-ATRNS-FOR-SPLIT-2ND-NODE VARYING L FROM 1 BY 1 UNTIL N = NUM-OF-ENTRIES

//then we restate the current node by removing the 2nd half of the attribute values from the current node
MOVE CURRENT-NODENC TO FILE-KEY
PERFORM WRITE-ATRNS-FOR-SPLIT-1ST-NODE VARYING L FROM N BY 1 UNTIL L > NUM-OF-ENTRIES
END OF PROCEDURE.

PROCEDURE CREATE-NEW-MASTER-ROOT;
MOVE 1 TO NUM-OF-ENTRIES
MOVE CURRENT-NODENC TO PT-TO-LCHILD
MOVE PROMOTE-ATRVAL TO ATR-VAL(1)
MOVE NEW-NODENC TO PT-TO-RECHILD(1)
MOVE AVAILABLE-CHILD TO FILE-KEY
retry the current master internal node

//store the new root & update the information for the master tree
MOVE FILE-KEY TO PT-TO-ROOT
ADD 1 TO NUM-OF-LEVELS
MOVE 1 TO NODES-PER-LEVEL(NUM-OF-LEVELS)
ADD 1 TO AVAILABLE-CHILD
END OF PROCEDURE.
PROCEDURE FIND-PCS-IN-INVERTED-LEAF;
//read a proper node in current level, store this node now,
//first check then store the proper entry position in the node, finally,
//move the proper pointer to child node to file-key in order to read the appropriate child node next time
//read a internal node record of inverted tree from the b-plus tree file
MOVE 0 TO CONE
PERFORM CHECK-POS-CN-CUR-INVERTED-NODE VARYING J FROM 1 BY 1 UNTIL J > I-REC-VAL(J) OR DONE = 1
IF DONE = 0
   MOVE I-PT-TO-RCHILD(J) TO FILE-KEY
   MOVE J TO POSITION-IN-A-NODE(LEVEL-NO)
END OF PROCEDURE;

PROCEDURE CHECK-POS-CN-CUR-INVERTED-NODE;
IF NEW-RECNO < I-REC-VAL(J)
   MOVE J TO POSITION-IN-A-NODE(LEVEL-NO)
   MOVE 1 TO DONE
IF J = 1
   MOVE I-PT-TO-RCHILD TO FILE-KEY
ELSE
   MOVE I-PT-TO-RCHILD(J - 1) TO FILE-KEY
END OF PROCEDURE;

PROCEDURE FIND-POS-IN-INVERTED-LEAF;
IF NEW-RECNO < I-REC-NUM(J)
   MOVE 1 TO DONE
END OF PROCEDURE;

PROCEDURE INSERT-TO-INVERTED-TREE-PROCESS;
//check if a new key has existed in a master leaf (if not just insert it in),
//insert the record no which associates with this new key to its corresponding inverted tree
//else, create an inverted tree using the record no which associates with this new key value
IF FOUND = 1
   PERFORM INSERT-NEW-REC-TO-AN-INV-TREE
ELSE
   PERFORM CREATE-NEW-INVERTED-TREE
END OF PROCEDURE;

PROCEDURE INSERT-NEW-REC-TO-AN-INV-TREE;
MOVE PT-TO-INVERTED-TREE(ENTRY-POS-IN-A-MASTER-LEAF) TO FILE-KEY
PERFORM SEARCH-DOWNTO-AN-INVERTED-LEAF VARYING I FROM 1 BY 1,
UNTIL I = INVTEC-TREE-LEVELS(ENTRY-POS-IN-A-MASTER-LEAF)
MOVE 0 TO CONE
PERFORM FLAG-POS-IN-INVERTED-LEAF VARYING J FROM 1 BY 1 UNTIL J > I-REC-VAL(J) OR DONE = 1
PERFORM INSERT-NEW-RECNO-TO-CUR-LEAF
END OF PROCEDURE;

PROCEDURE SEARCH-DOWNTO-AN-INVERTED-LEAF;
//read a proper node in current level, store this node now,
//first check then store the proper entry position in the node, finally,
//move the proper pointer to child node to file-key in order to read the appropriate child node next time
//read a internal node record of inverted tree from the b-plus tree file
MOVE 0 TO CONE
ADD 1 TO LEVEL-NO
MOVE FILE-KEY TO NO-END(LEVEL-NO)
PERFORM CHECK-POS-CN-CUR-INVERTED-NODE VARYING J FROM 1 BY 1 UNTIL J > I-REC-VAL(J) OR DONE = 1
IF DONE = 0
   MOVE I-PT-TO-RCHILD(I-REC-VAL(J)) TO FILE-KEY
   MOVE J TO POSITION-IN-A-NODE(LEVEL-NO)
END OF PROCEDURE;
INSERT NEW-RECORD TO CUR-LEAF
M = I-NUM-CF-RECS - J + 1
PERFORM SHIFT-RECS-IN-CUR-LEAF M TIMES
ADD 1 TO I-NUM-OF-RECS
MOVE NEW-RECORD TO I-REC-NUM(J)
REWRITE the current inverted leaf node

IF I-NUM-OF-RECS > MAX-ENTRIES
PERFORM SPLIT-AN-INVITED-LEAF-PROCESS
END OF PROCEDURE

PROCEDURE SPLIT-AN-INVITED-LEAF-PROCESS:
//split current leaf node into two nodes, promote the image of the middle record value to next level
N = I-NUM-CF-RECS / 2
MOVE I-REC-NUM(J) TO TEMP-RECORD
MOVE AVAIL-RECORD TO NEW-RECORD
MOVE FILE-KEY TO CURRENT-INDEX
MOVE N TO M

//first, we use the 2nd half of the record numbers from current leaf to create a new leaf
PERFORM WRITE-RECS-FOR-SPLIT-2ND-LEAF VARYING L FROM 1 BY 1 UNTIL M > I-NUM-OF-RECS

//then we rewrite the current leaf node by removing the 2nd half of the record numbers from current leaf
MOVE CURRENT-INDEX TO FILE-KEY
PERFORM WRITE-RECS-FOR-SPLIT-1ST-LEAF VARYING L FROM N BY 1 UNTIL L > I-NUM-OF-RECS

//finally, we promote the image of the middle record value from the split leaf to internal levels
MOVE 0 TO DONE
PERFORM PROMPT-TO-INVITED-INTER-LEVELS UNTIL DONE = 1 OR LEVEL-NO = 0
IF DONE = 0 AND LEVEL-NO = 0
PERFORM CREATE-NEW-INVITED-RECORD
END OF PROCEDURE

PROCEDURE PROMPT-TO-INVITED-INTER-LEVELS:
//put the image of the middle record value from the split node on previous level
//to the proper entry position of its parent node on current level
MOVE NOINDEX(LEVEL-ADJ) TO FILE-KEY
MOVE POSITION-IN-A-NODE(LEVEL-NO) TO J
M = I-NUM-OF-ENTRIES - J + 1
PERFORM SHIFT-RECEIVED-IN-CUR-NODE M TIMES
ADD 1 TO I-NUM-OF-ENTRIES
MOVE TEMP-RECORD TO I-REC-VAL(J)
MOVE NEW-INDEX TO I-PT-TO-CHILD(J)
REWRITE the current inverted internal node

IF I-NUM-OF-ENTRIES > MAX-ENTRIES
PERFORM SPLIT-AN-INVITED-INTERNAL-NODE
ELSE
MOVE 1 TO DONE
END OF PROCEDURE
PROCEDURE SPLIT-AN-INVITED-INTERNAL-NODES:
//split current internal node into two nodes, promote the image of the middle record value to next level
N = I-NUM-OF-ENTRIES / 2
MOVE I-REC-VAL(N) TO TEMP-RECN

//first, use the 2nd half of the record values from current node to create a new node
PERFORM WRITE-RECS-TO-SPLIT-2ND-NODE VARYING L FROM 1 BY 1 UNTIL L = I-NUM-OF-ENTRIES

//then we restructure the current node by removing the 2nd half of the record values from current node
MOVE CURRENT-NODE-END TO FILE-KEY
PERFORM WRITE-RECS-TO-SPLIT-1ST-NODE VARYING L FROM N BY 1 UNTIL L > I-NUM-OF-ENTRIES
END OF PROCEDURE

PROCEDURE CREATE-NEW-INVITED-ROOT:
MOVE 1 TO I-NUM-OF-ENTRIES
MOVE CLEAREN-NODE-END TO I-PT-TO-CHILD
MOVE TEMP-RECN TO I-REC-VAL(1)
MOVE NEW-NODE-END TO I-PT-TO-CHILD(1)
MOVE AVAIL-RECN TO FILE-KEY
rewrite the current invted internal node

MOVE AVAIL-RECN TO PT-TO-INVITE TREE(ENTRY-POS-IN-A-MASTER-LEAF)
ADD 1 TO AVAIL-RECN
ADD 1 TO INVITED-TREE-LEVELS(ENTRY-POS-IN-A-MASTER-LEAF)
MOVE INVICTED-TREE-LEVELS(ENTRY-POS-IN-A-MASTER-LEAF) TO LEVEL-NUM
MOVE 1 TO INVITED-LEVEL-NODES(ENTRY-POS-IN-A-MASTER-LEAF, LEVEL-NUM)
END OF PROCEDURE

CREATE-NEW-INVITED-TREE
MOVE AVAIL-RECN TO NEW-NODE-END
ADD 1 TO AVAIL-RECN
MOVE 0 TO INVITED-LEAF-NODE
MOVE 1 TO I-NUM-OF-RECS
MOVE NEW-RECN TO I-REC-NUM(1)
MOVE NEW-NODE-END TO FILE-KEY
MOVE FILE-KEY TO PT-TO-INVITED-TREE(ENTRY-POS-IN-A-MASTER-LEAF)
write node=rec from invited=leaf node
MOVE 1 TO INVITED-TREE-LEVELS(ENTRY-POS-IN-A-MASTER-LEAF)
MOVE 1 TO INVITED-LEVEL-NODES(ENTRY-POS-IN-A-MASTER-LEAF, 1)
END OF PROCEDURE
PROGRAM DELETE-RECORD-FROM-HYBRID-DIRECTORY;

//first search the master tree from the root level by level
//until reaches the proper leaf of master tree find the proper key entry
//then search the inverted tree which corresponds to this key
//to find a specific record ra and delete it from this inverted tree

read boto-tree-file into master-info-rec
PERFORM SEARCH-DOWNTO-MASTER-LEAF VARYING I FROM 1 BY 1 UNTIL I = NUM-OF-LEVELS
MOVE K TO FILE-KEY WRITE-ACCEED
read boto-tree-file into master-leaf-node
MOVE 0 TO FOUND
PERFORM FIND-POS-IN-MASTER-LEAF VARYING J FROM 1 BY 1 UNTIL J > NUM-OF-KEYS OR FOUND = 1
IF FOUND = 1
PERFORM DELETE-FROM-INV-TREE-PROCESS

//check if all of the nodes in current inverted tree are being deleted,
//then we continue to delete a key entry of a master leaf which the current inverted tree corresponds to
IF CONTINUE = 1
PERFORM DELETE-FROM-MASTER-LEAF
PERFORM RESTORE-MASTER-TREE-INFO
ELSE
PERFORM RESTORE-INVITED-TREE-INFO
END OF MAIN PROGRAM

PROCEDURE DELETE-FROM-INV-Tree-Process

//search down to the leaf level following the proper node on each internal level
//finds a specific leaf node where the deleted rec no is located and remove it from current inverted tree

MOVE PT-TO-INVITED-TREE(LOC) TO K
PERFORM SEARCH-DOWNTO-AN-INVITED-LEAF VARYING I FROM 1 BY 1 UNTIL I = INVITED-Tree-LEVELS(LOC)
MOVE K TO FILE-KEY
read boto-tree-file into invited-leaf-node
MOVE 0 TO FOUND
PERFORM FIND-POS-IN-INVITED-LEAF VARYING J FROM 1 BY 1 UNTIL J > I-NUM-OF-RECS OR FOUND = 1
IF FOUND = 1
PERFORM DELETE-FROM-INVITED-Tree
ELSE
DISPLAY 'SOMETHING's WRONG, RECORD NO.>>' ^TTY-Rec 'is NOT ACTIVATED IN THE FILE!!'
END OF PROCEDURE

PROCEDURE DELETE-FROM-INVITED-Tree

//delete a specific record no from a leaf of the current inverted tree,
//check if the entries of the current leaf has less than the minimum number of entries
//if it does, we need either borrow a entry from the nearest sibling node or
//combine the current leaf with a nearest entry of closest sibling node
//and their parent node to form a new node then check its parent node,
//following the same process until a specific node on one level has more than minimum number of entries

COMPUTE I-NUM-OF-RECS = I-NUM-OF-RECS - 1
write node-rec from invited-leaf-node
MOVE 0 TO FLAG
IF I-NUM-OF-RECS = 0 AND INVITED-Tree-LEVELS(LOC) = 1
MOVE 1 TO CONTINUE
IF I-NUM-OF-RECS < MIN-ENTRIES AND INVITED-Tree-LEVELS(LOC) > 1
PERFORM PROCESS-CUR-INVITED-LEAF
IF FLAG = 1
IF I-NUM-OF-ENTRIES < MIN-ENTRIES
MOVE C TO DONE, FLAG
PERFORM RESTORE-INVITE-INTERNAL UNTIL DONE = 1 OR KOUNT = 1
MOVE INVITED-LEVEL-LEVELS(LOC) TO L
IF INVITED-LEVEL-LEVELS(LOC, L) = 1 AND I-NUM-OF-ENTRIES = 0
MOVE READ-ADJACENT TO PT-TO-INVITED-LEVEL(LOC)
MOVE 0 TO INVITED-LEVEL-LEVELS(LOC, L)
COMPUTE INVITED-LEVEL-LEVELS(LOC) = INVITED-LEVEL-LEVELS(LOC) - 1
END OF PROCEDURE

PROCEDURE PROCESS-CUR-INVITED-LEAF
// look at the nearest right or left sibling node to see if it has more than minimum number of entries,
// then do a borrow transformation otherwise, we need combine the current node with its parent node
// and the nearest sibling node to rewrite the current node
MOVE J TO DONE
PERFORM TRY-BORROW-I-LEAF-SIBLING
IF DONE = C
MOVE 1 TO FLAG
MOVE 1 TO L
COMPUTE INVITED-LEVEL-LEVELS(LOC, L) = INVITED-LEVEL-LEVELS(LOC, L) - 1
PERFORM COMBINE-I-LEAF-SIBLING
END OF PROCEDURE

PROCEDURE TRY-BORROW-I-LEAF-SIBLING
// check the nearest right sibling node if there is a one to see if it has more than
// the minimum number of entries, if it does, borrow it; else,
// go to the nearest left sibling node if there is a one to check
MOVE FILE-KEY TO ADDS-KEY
MOVE I-NODENO(KOUNT) TO FILE-KEY
MOVE I-POS(KOUNT) TO P
READ BTRAS-FILE INTO INVITED-INTERNAL-NODE
IF P = 0
PERFORM CHECK-I-LEAF-SIBLING
ELSE
PERFORM CHECK-I-LEAF-SIBLING
IF DONE = 0 AND P NOT EQUAL TO I-NUM-OF-ENTRIES
PERFORM CHECK-I-LEAF-SIBLING
END OF PROCEDURE

PROCEDURE CHECK-I-LEAF-SIBLING
MOVE NODE-KEY TO FILE-KEY
READ BTRAS-FILE INTO INVITED-LEAF-NODE
MOVE I-LINK-TO-SIBLING-LEAF TO FILE-KEY
READ BTRAS-FILE INTO INVITED-LEAF-NODE
IF I-NUM-OF-RECS > MIN-ENTRIES
PERFORM LEND-I-LEAF-SIBLING
MOVE 1 TO DONE
END OF PROCEDURE
PROCEDURE LENO-I-LEAF-LSIBLING
//borrow the last entry from the nearest left sibling node to rewrite the current node
MOVE I-REC-NUM(1) TO TEMP-REC(1)
MOVE 0 TO I-REC-NUM(1)
MOVE I-REC-NUM(I-REC-ADJ) TO I-REC-ADJ
WRITE node-rec from invted-leaf-node
READ node-rec from invted-leaf-node
WRITE node-rec from invted-leaf-node
READ node-rec from invted-leaf-node
PERFORM SHIFT-REC-TO-RIGHT I-REC-NUM(1) TIMES
ADD 1 TO I-REC-NUM(1)
WRITE node-rec from invted-leaf-node
READ node-rec from invted-leaf-node
WRITE node-rec from invted-leaf-node
READ node-rec from invted-leaf-node
PERFORM SHIFT-REC-TO-RIGHT I-REC-NUM(1) TIMES
ADD 1 TO I-REC-NUM(1)
WRITE node-rec from invted-leaf-node
READ node-rec from invted-leaf-node
END OF PROCEDURE

PROCEDURE CHECK-I-LEAF-LSIBLING
IF P = 1
MOVE I-PT-TO-CHILD TO FILE-KEY
ELSE
MOVE I-PT-TO-CHILD(P-1) TO FILE-KEY
READ node-rec from invted-leaf-node
END OF PROCEDURE

PROCEDURE LENO-I-LEAF-LSIBLING
//borrow the last entry from the nearest left sibling node to rewrite the current node
MOVE I-REC-NUM(I-REC-ADJ) TO TEMP-REC(1)
MOVE 0 TO I-REC-NUM(1)
MOVE I-REC-NUM(I-REC-ADJ) TO I-REC-ADJ
WRITE node-rec from invted-leaf-node
READ node-rec from invted-leaf-node
READ node-rec from invted-leaf-node
READ node-rec from invted-leaf-node
READ node-rec from invted-leaf-node
PERFORM SHIFT-REC-TO-RIGHT I-REC-NUM(1) TIMES
ADD 1 TO I-REC-NUM(1)
WRITE node-rec from invted-leaf-node
READ node-rec from invted-leaf-node
READ node-rec from invted-leaf-node
READ node-rec from invted-leaf-node
READ node-rec from invted-leaf-node
READ node-rec from invted-leaf-node
END OF PROCEDURE

PROCEDURE COMBINE-I-LEAF-LSIBLING
//combine the current node with the nearest sibling node and the proper entry
//on their parent node to form a new node on the current level
IF P = 0
   MOVE I-PERT-RC>ILD(1) TO FILE-KEY
ELSE
   MOVE NODE-KEY TO FILE-KEY
   READ btree-file into invtd-leaf-node
   PERFORM STCK-I-LEAF-SIBLING VARYING N FROM 1 BY 1 UNTIL N > I-NUM-OF-RECS
   MOVE I-NUM-OF-RECS TO M
   MOVE I-LINK-TO-RIGHT-LEAF TO I-LINK-TO-RIGHT-LEAF
   IF P = 0 OR P = 1
   MOVE I-PERT-RCCHILD TO FILE-KEY
   ELSE
   MOVE I-PERT-RCCHILD(P - 1) TO FILE-KEY
   MOVE FILE-KEY TO READ-NODE-READ
   READ btree-file into invtd-leaf-node
   PERFORM RESTORE-CURRENT-I-LEAF VARYING N FROM 1 BY 1 UNTIL N > M
   MOVE T-I-LINK-TO-RIGHT-LEAF TO I-LINK-TO-RIGHT-LEAF
   WRITE NODE-REC FROM invtd-leaf-node
   COMPUTE I-NUM-OF-ENTRIES = I-NUM-OF-ENTRIES - 1
   MOVE I-NODENO(KOUNT) TO FILE-KEY
   WRITE NODE-REC FROM invtd-internal-node
END OF PROCEDURE

PROCEDURE RESTORE-INVTD-INTERNAL
   COMPUTE KOUNT = KOLAT - 1
   MOVE 0 TO DONE
   PERFORM TRY-BORROW-I-INTERNAL-SIBLING
   IF DONE = 0
      MOVE 1 TO FLg
      ADD 1 TO L
      COMPUTE INVTD-LEVEL-NODES(LOG, L) = INVTD-LEVEL-NODES(LOG, L) - 1
   PERFORM COMBINE-I-INTERNAL-SIBLING
END OF PROCEDURE

PROCEDURE TRY-BORROW-I-INTERNAL-SIBLING
   IF P = 0
      PERFORM CHECK-I-INTERNAL-RSIBLING
   ELSE
      PERFORM CHECK-I-INTERNAL-RSIBLING
      IF DONE = 0 AND P NOT EQUAL X
         PERFORM CHECK-I-INTERNAL-RSIBLING
END OF PROCEDURE

PROCEDURE CHECK-I-INTERNAL-RSIBLING
   READ bteer-file into invtd-internal-node
   IF I-NUM-OF-ENTRIES > MIN-ENTRIES
      PERFORM END-I-INTERNAL-RSIBLING
   MOVE 1 TO DONE
END OF PROCEDURE

PROCEDURE DELETE-FROM-MASTER-TREE
   MOVE M-KOUNT TO KOLAT
   MOVE LCC TO 1
   MOVE 0 TO KEY-INFO(J)
   COMPUTE M = NUM-OF-KEYS - J
   MOVE J TO h
   PERFORM SHIFT-KEY-TO-LEFT M TIMES
write node-rec from master-leaf-node
MOVE 0 TO FLAG
IF NUM-OF-KEYS < MIN-ENTRIES AND NUM-OF-LEVELS > 1
PERFORM PROCESS-CLR-MASTER-LEAF
IF FLAG = 1
IF NUM-OF-ENTRIES < MIN-ENTRIES
MOVE C TO DONE, FLAG,
PERFORM RESTORE-MASTER-INTERNAL UNTIL DONE = 1 OR COUNT = 1
IF NODES-PER-LEVEL(NUM-OF-LEVELS) = 1 AND NUM-OF-ENTRIES = 0
MOVE READ-ADDDC TC PT-TO-ROOT
MOVE 0 TO NODES-PER-LEVEL(NUM-OF-LEVELS)
COMPUTE NUM-OF-LEVELS = NUM-OF-LEVELS - 1
END OF PROCEDURE

PROCEDURE PROCESS-CUR-MASTER-LEAF
// look at the nearest right or left sibling node to see if it has more than minimum number of entries
// then do a borrow transformation otherwise we need combine the current node with its parent node
// and the nearest sibling node to rewrite the current node
MOVE 0 TO DONE
PERFORM TRY-BORROW-P-LEAF-SIBLING
IF DONE = 0
MOVE 1 TO FLAG
MOVE 1 TO L
COMPUTE NODES-PER-LEVEL(L) = NODES-PER-LEVEL(L) - 1
PERFORM COMBINE-P-LEAF-SIBLING
END OF PROCEDURE

PROCEDURE TRY-BORROW-P-LEAF-SIBLING
// check the nearest right or left sibling node if there is a one to see if it has more than
// the minimum number of entries, if it does, borrow it; else
// go to the nearest left sibling node if there is a one to check
MOVE FILE-KEY TO NODE-KEY
MOVE NODES(COUNT) TO FILE-KEY
MOVE POS(COUNT) TO P
Read btreenode-file into master-internal-node
IF P = 0
PERFORM CHECK-M-LEAF-RSIBLING
ELSE
PERFORM CHECK-M-LEAF-RSIBLING
IF DONE = 0 AND P NOT EQUAL TO NUM-OF-ENTRIES
PERFORM CHECK-P-LEAF-RSIBLING
END OF PROCEDURE

PROCEDURE CHECK-M-LEAF-RSIBLING
MOVE NODE-KEY TO FILE-KEY
Read btreenode-file into master-leaf-node
MOVE LINK-TO-LEFT-LEAF TO FILE-KEY
Read btreenode-file into master-leaf-node
IF NUM-OF-KEYS > MIN-ENTRIES
PERFORM LINK-M-LEAF-RSIBLING
MOVE 1 TO DONE
END OF PROCEDURE
PROCEDURE LEAD-M-LEAF-RSIBLING
// borrow the first entry from the nearest right sibling node to rewrite the current node
MOVE KEY-INFO(1) TO TEMP-KEY-INFO(1)
COMPUTE NUM-OF-KEYS = NUM-OF-KEYS - 1
MOVE 1 TO A
PERFORM SHIFT-KEY-TO-LEFT NUM-OF-KEYS TIMES
MOVE 0 TO KEY-INFO(A)
write node-rec from master-leaf-node
MOVE NOGENC(COUNT) TO FILE-KEY
MOVE KEY-VAL(1) TO ATR-VAL(P - 1)
write node-rec from master-internal-node
MOVE NODE-KEY TO FILE-KEY
read subtype-file into master-leaf-node
ADC 1 TO NUM-OF-KEYS
MOVE TEMP-KEY-INFO(1) TO KEY-INFO(NUM-OF-KEYS)
write node-rec from master-leaf-node
END OF PROCEDURE

PROCEDURE CHECK-M-LEAF-LSIBLING
IF P = 1
MOVE PT-TO-LCHILD TO FILE-KEY
ELSE
MOVE PT-TO-RCHILD(P - 1) TO FILE-KEY
read subtype-file into master-leaf-node
IF NUM-OF-KEYS > MAX-ENTRIES
PERFORM LEAD-M-LEAF-LSIBLING
MOVE 1 TO DONE
END OF PROCEDURE

PROCEDURE LEAD-M-LEAF-LSIBLING
// borrow the last entry from the nearest left sibling node to rewrite the current node
MOVE KEY-INFO(NUM-OF-KEYS) TO TEMP-KEY-INFO(1)
MOVE 0 TO KEY-INFO(NUM-OF-KEYS)
COMPUTE NUM-OF-KEYS = NUM-OF-KEYS - 1
write node-rec from master-leaf-node
MOVE NODE-KEY TO FILE-KEY
read subtype-file into master-leaf-node
MOVE NUM-OF-KEYS TO N
PERFORM SHIFT-KEY-TO-RIGHT NUM-OF-KEYS TIMES
ADC 1 TO NUM-OF-KEYS
MOVE TEMP-KEY-INFO(1) TO KEY-INFO(1)
write node-rec from master-leaf-node
MOVE NOGENC(COUNT) TO FILE-KEY
MOVE T-KEY-VAL(1) TO ATR-VAL(P)
write node-rec from master-internal-node
END OF PROCEDURE

PROCEDURE COMBINE-M-LEAF-SIBLING
// combine the current node with the nearest sibling node and the proper entry
// on their parent node to form a new node on the current level
IF P = 0
MOVE PT-TO-RCHILD(1) TO FILE-KEY
ELSE
MOVE NODE-KEY TO FILE-KEY
PROCEDURE RESTORE-MASTER-INTERNAL

COMPUTE MINTY = MINTY - 1
MOVE D TO CEA
PERFORM TRY-BORROW-W-INTERNAL-SIBLING
IF DONE = C
MOVE 1 TO FLAG
ADD 1 TO L
COMPUTE NODES-LEVEL(L) = NODES-LEVEL(L) - 1
PERFORM CONSIST-W-INTERNAL-SIBLING
END OF PROCEDURE

PROCEDURE TRY-BORROW-W-INTERNAL-SIBLING

read btree-file into master-internal-node
MOVE NUM-OF-ENTRIES TO X
IF P = D
PERFORM CHECK-W-INTERNAL-SIBLING
ELSE
PERFORM CHECK-W-INTERNAL-LSIBLING
IF DONE = D AND P NOT EQUAL X
PERFORM CHECK-W-INTERNAL-RSIBLING
END OF PROCEDURE

PROCEDURE CHECK-W-INTERNAL-RSIBLING

read btree-file into master-internal-node
IF NUM-OF-ENTRIES > MIN-ENTRIES
PERFORM LEAN-W-INTERNAL-RSIBLING
MOVE 1 TO DONE
END OF PROCEDURE
File name: INVTED.FIL (for range attribute)

<table>
<thead>
<tr>
<th>File name: INVTED.FIL (for range attribute)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1122 6 3900023</td>
</tr>
<tr>
<td>12 10000 11000 4 11000 14 10000 15 10000 16 10000 17 11000 19 10000 47 10000 56 10000 75 10000102 11000103 10000118 10000</td>
</tr>
<tr>
<td>11 23000 23000 2 23000 7 23000 10 23000 29 23000 48 23000 55 23000 66 23000 72 23000 92 23000100 23000124 23000</td>
</tr>
<tr>
<td>10 31000 34000 5 31000 8 31000 11 31000 40 34000 45 31000 73 31000 73 31000105 31000112 34000120 31000</td>
</tr>
<tr>
<td>13 35000 37000 12 35000 23 35000 36 35000 60 35000 65 37000 69 37000 78 37000 80 36000 98 35000101 35000113 37000131 37000132 35000</td>
</tr>
<tr>
<td>11 40000 43000 9 40000 18 43000 33 43000 46 43000 49 43000 51 43000 64 43000 74 43000111 43000114 43000133 43000</td>
</tr>
<tr>
<td>12 50000 56000 3 50000 25 56000 34 50000 35 50000 59 50000 82 50000 84 50000 95 50000 96 50000104 50000115 50000129 50000</td>
</tr>
<tr>
<td>15 56000 72000 32 72000 37 56000 42 58000 44 72000 59 56000 61 58000 62 58000 86 58000 89 72000 94 58000109 56000110 560001</td>
</tr>
<tr>
<td>25 72000134 58000</td>
</tr>
<tr>
<td>19 74000 74000 24 74000 28 74000 38 74000 39 74000 41 74000 43 74000 52 74000 71 74000 76 74000 77 74000 79 74000 81 74000 83 74000</td>
</tr>
<tr>
<td>87 74000 81 74000 91 74000107 74000126 74000127 74000</td>
</tr>
<tr>
<td>7 77000 77000 6 77000 20 77000 20 77000 63 77000 93 77000128 77000130 77000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File name: INVTED.FIL (for range attribute)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1122 6 3900023</td>
</tr>
<tr>
<td>12 10000 11000 4 11000 14 10000 15 10000 16 10000 17 11000 19 10000 47 10000 56 10000 75 10000102 11000103 10000118 10000</td>
</tr>
<tr>
<td>11 23000 23000 2 23000 7 23000 10 23000 29 23000 48 23000 55 23000 66 23000 72 23000 92 23000100 23000124 23000</td>
</tr>
<tr>
<td>10 31000 34000 5 31000 8 31000 11 31000 40 34000 45 31000 73 31000 73 31000105 31000112 34000120 31000</td>
</tr>
<tr>
<td>13 35000 37000 12 35000 23 35000 36 35000 60 35000 65 37000 69 37000 78 37000 80 36000 98 35000101 35000113 37000131 37000132 35000</td>
</tr>
<tr>
<td>11 40000 43000 9 40000 18 43000 33 43000 46 43000 49 43000 51 43000 64 43000 74 43000111 43000114 43000133 43000</td>
</tr>
<tr>
<td>12 50000 56000 3 50000 25 56000 34 50000 35 50000 59 50000 82 50000 84 50000 95 50000 96 50000104 50000115 50000129 50000</td>
</tr>
<tr>
<td>15 56000 72000 32 72000 37 56000 42 58000 44 72000 59 56000 61 58000 62 58000 86 58000 89 72000 94 58000109 56000110 560001</td>
</tr>
<tr>
<td>25 72000134 58000</td>
</tr>
<tr>
<td>19 74000 74000 24 74000 28 74000 38 74000 39 74000 41 74000 43 74000 52 74000 71 74000 76 74000 77 74000 79 74000 81 74000 83 74000</td>
</tr>
<tr>
<td>87 74000 81 74000 91 74000107 74000126 74000127 74000</td>
</tr>
<tr>
<td>7 77000 77000 6 77000 20 77000 20 77000 63 77000 93 77000128 77000130 77000</td>
</tr>
</tbody>
</table>
**A UNIFORM SYSTEM FOR DATA BASE DIRECTORY --- PROCESS OF DISCRETE ATTRIBUTE**

**ENTER ONE OF THE FOLLOWING FUNCTION TYPES :**

**ENTER C FOR CREATION**

**ENTER I FOR INSERTION**

**ENTER S FOR SEARCH**

**ENTER B FOR EXIT THE SYSTEM**

**ENTER TYPE HERE >>> C**

**DO YOU WANT TO CREATE A SORTED FILE WHICH CONTAINS REC NO. & ATTRIBUTE VALUE PAIRS (Y OR N) ? Y**

**DO YOU WANT TO SEE THE CURRENT DIRECTORY DISPLAYED ON TTY (Y OR N) ? Y**

**THE HYBRID B-PLUS TREE & INVERTED FILES**

**LEVEL >>1**

**NODE # >> 82**

[Tree structure details]
LEVEL >>2

NODE #>>80
# ENTRY>>2
******
LCHILD #74
******
ATR >>ECON
RCHILD #75
******
ATR >>ENGL
RCHILD #76
******

***

LEVEL >>3

NODE #>>74
# KEYS >>2
******
KEY >>EAS
PT TREE#10
******
KEY >>CS
PT TREE#23
******

NODE #>>75
# KEYS >>2
******
KEY >>ECON
PT TREE#26
******

NODE #>>76
# KEYS >>2
******
KEY >>ENGL
PT TREE#33
******

NODE #>>77
# KEYS >>2
******
KEY >>IE
PT TREE#37
******

NODE #>>78
# KEYS >>2
******
KEY >>MATH
PT TREE#58
******

NODE #>>79
# KEYS >>2
******
KEY >>MGMT
PT TREE#66
******

AN INVERTED TREE FOR KEY >> EAS
>>> AN INVERTED TREE FOR KEY >> CS <<<

LEVEL >>1

NODE #>>23
# INDEX>>2
LCHILD #20
INDEX #39
RCHILD #21
INDEX #74
RCHILD #22

LEVEL >>2

NODE #>>20
# INDEX>>2
LCHILD #11
INDEX #23
RCHILD #12
INDEX #30
RCHILD #13
AN INVERTED TREE FOR KEY >> ENGL

LEVEL >>1

NODE #>>30
# INDEX >>2
LCHILD #30
INDEX >>73
RCHILD #31
INDEX >>98
RCHILD #32

LEVEL >>2

NODE #>>31
# RECS >>2
REC # >>24
REC # >>42

NODE #>>32
# RECS >>3
REC # >>98
REC # >>108
REC # >>109

>>> AN INVERTED TREE FOR KEY >> GEOG <<<<
>>> AN INVERTED TREE FOR KEY >> MATH <<<<

LEVEL >>1
*******

NODE #358
*******
INDEX #2
LCHILD #55
INDEX #27
RCHILD #55
INDEX #32
RCHILD #57
*******

LEVEL >>2
*******

NODE #355
*******
INDEX #2
LCHILD #44
INDEX #21
RCHILD #45
INDEX #25
RCHILD #46
*******

LEVEL >>3

NODE #356
*******
INDEX #2
LCHILD #47
INDEX #24
RCHILD #48
INDEX #26
RCHILD #49
*******

NODE #357
*******
INDEX #2
LCHILD #50
INDEX #27
RCHILD #51
INDEX #28
RCHILD #52
INDEX #29
RCHILD #53
INDEX #30
RCHILD #54
*******
LEVEL >>2
*********
NODE >>>60
*********
# RECS >>2
*********
REC # >>3
*********
REC # >>2
*********
REC # >>1
*********
REC # >>1
*********
LEVEL >>>1
*********
NODE >>>73
*********
# INDEX >>1
*********
CHILD #71
*********
INDEX >>2
*********
CHILD #72
*********
*********

>>> AN INVERTED TREE FOR KEY >> SOC <<<
THE RECORDS WHICH HAS THIS ATTRIBUTE VALUE ARE AS THE FOLLOWING:

DEPT ID NUMBER PERSON NAME SALARY REC NO
**ECON** 3637356## RUTH KAY 74/630 24
**ECON** 3605429## CHARLES ANY 50/630 34
**ECON** 3445653## BOBBY VICTORIA 10/630 47
**ECON** 3112825## PAYNE RICHARD 50/630 94
**ECON** 3846666## STAFFER NANCY 39/630 90

DO YOU WANT TO SEE THE CURRENT DIRECTORY DISPLAYED ON TTY (Y OR N)? N

ENTER ONE OF THE FOLLOWING FUNCTION TYPES:

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
ENTER C FOR CREATION
ENTER D FOR DELETION
ENTER I FOR INSERTION
ENTER S FOR SEARCH
ENTER E FOR EXIT THE SYSTEM

ENTER TYPE HERE >>> I

ENTER ONE OF THE FOLLOWING OPTIONS FOR INSERTION:

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
OPTION T REPRESENTS ENTERING DATA FROM TTY
OPTION F REPRESENTS ENTERING DATA FROM EXISTING FILE

ENTER OPTION HERE >>> T

ENTER THE FOLLOWING ATTRIBUTE VALUES FOR A NEW RECORD:

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
ENTER THE DEPT. NAME >> ECON
ENTER EMPLOYEE IC # >> 33345566
ENTER EMPLOYEE NAME >> BETTY PAT
ENTER EMPLOYEE SALARY >> 30000

DO YOU WANT TO INSERT A NEW RECORD AGAIN (ENTER Y OR N)? Y

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
ENTER THE FOLLOWING ATTRIBUTE VALUES FOR A NEW RECORD:

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
ENTER THE DEPT. NAME >> ECON
ENTER EMPLOYEE IC # >> 34456666
ENTER EMPLOYEE NAME >> MUANG KAY
ENTER EMPLOYEE SALARY >> 34000

DO YOU WANT TO INSERT A NEW RECORD AGAIN (ENTER Y OR N)? Y

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
ENTER THE FOLLOWING ATTRIBUTE VALUES FOR A NEW RECORD:

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
ENTER THE DEPT. NAME >> PHY
ENTER EMPLOYEE IC # >> 32299828
ENTER EMPLOYEE NAME >> JULIA FRANK
ENTER EMPLOYEE SALARY >> 75000

DO YOU WANT TO INSERT A NEW RECORD AGAIN (ENTER Y OR N)? Y

ENTER THE FOLLOWING ATTRIBUTE VALUES FOR A NEW RECORD:

ENTER THE DEPT. NAME >> LING
ENTER EMPLOYEE IC # >> 377597654
ENTER EMPLOYEE NAME >> RAY KARL
ENTER EMPLOYEE SALARY >> 55000

DO YOU WANT TO INSERT A NEW RECORD AGAIN (ENTER Y OR N)? N

DO YOU WANT TO SEE THE CURRENT DIRECTORY DISPLAYED ON TTY (Y OR N)? N

ENTER ONE OF THE FOLLOWING FUNCTION TYPES :

ENTER C FOR CREATION
ENTER D FOR DELETION
ENTER I FOR INSERTION
ENTER S FOR SEARCH
ENTER E FOR EXIT THE SYSTEM

ENTER CODE HERE >>> S

ENTER ATTRIBUTE VALUE NEED TO SEARCH >> LING

THE RECORDS WHICH HAS THIS ATTRIBUTE VALUE ARE AS THE FOLLOWING :

<table>
<thead>
<tr>
<th>DEPT IC NUMBER</th>
<th>PERSON NAME</th>
<th>SALARY</th>
<th>DEP NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>LING 355275566</td>
<td>CHEN KARL</td>
<td>25000</td>
<td>2</td>
</tr>
<tr>
<td>LING 377508313</td>
<td>BELT LARRY</td>
<td>10500</td>
<td>16</td>
</tr>
<tr>
<td>LING 31154321</td>
<td>RABB LYNN</td>
<td>3500</td>
<td>23</td>
</tr>
<tr>
<td>LING 36645332</td>
<td>JORO DLYL</td>
<td>74500</td>
<td>41</td>
</tr>
<tr>
<td>LING 34424221</td>
<td>SOOTH SEPHEN</td>
<td>43000</td>
<td>49</td>
</tr>
<tr>
<td>LING 311222347</td>
<td>MACK JERRY</td>
<td>35000</td>
<td>60</td>
</tr>
<tr>
<td>LING 385349452</td>
<td>MOORE JENNY</td>
<td>37000</td>
<td>69</td>
</tr>
<tr>
<td>LING 311288316</td>
<td>FABER RICK</td>
<td>39000</td>
<td>90</td>
</tr>
<tr>
<td>LING 37754213</td>
<td>BELT THOMAS</td>
<td>11500</td>
<td>102</td>
</tr>
<tr>
<td>LING 377548219</td>
<td>CARRY KORMA</td>
<td>74500</td>
<td>107</td>
</tr>
<tr>
<td>LING 377597654</td>
<td>RAY KARL</td>
<td>55000</td>
<td>118</td>
</tr>
</tbody>
</table>

DO YOU WANT TO SEE THE CURRENT DIRECTORY DISPLAYED ON TTY (Y OR N)? N
ENTER ONE OF THE FOLLOWING FUNCTION TYPES:

ENTER C FOR CREATION
ENTER D FOR DELETION
ENTER I FOR INSERTION
ENTER S FOR SEARCH
ENTER E FOR EXIT THE SYSTEM

ENTER TYPE HERE >> S

ENTER ATTRIBUTE VALUE NEED TO SEARCH >> ECON

THE RECORDS WHICH HAS THIS ATTRIBUTE VALUE ARE AS THE FOLLOWING:

DEPT ID  PERSON NAME     SALARY  REC NO
---------  ---------------  ------  ----
      ECON  RUTH, KAY     74,600  24
      ECON  CHARLES, AMY  50,000  34
      ECON  BOOY, VICTORIA 10,000  47
      ECON  PHIL, RICHARD  50,000  64
      ECON  SMART, NANCY   35,000  90
      ECON  BETTY, PAT     35,000  113
      ECON  MURR, KAY      35,000  114

DO YOU WANT TO SEE THE CURRENT DIRECTORY DISPLAYED ON TTY (Y OR N) ? N

ENTER ONE OF THE FOLLOWING FUNCTION TYPES:

ENTER C FOR CREATION
ENTER D FOR DELETION
ENTER I FOR INSERTION
ENTER S FOR SEARCH
ENTER E FOR EXIT THE SYSTEM

ENTER TYPE HERE >> D

ENTER ONE OF THE FOLLOWING OPTIONS FOR DELETION:

ENTER A RECORD NO. NEED TO BE DELETE >> 112

DO YOU HAVE MORE KEY VALUES NEED TO BE DELETED (ENTER Y OR N) ? Y

ENTER A RECORD NO. NEED TO BE DELETE >> 114

DO YOU HAVE MORE KEY VALUES NEED TO BE DELETED (ENTER Y OR N) ? Y
DO YOU WANT TO SEE THE CURRENT DIRECTORY DISPLAYED ON TTY (Y OR N)? N

ENTER ONE OF THE FOLLOWING FUNCTION TYPES:

ENTER C FOR CREATION
ENTER D FOR DELETION
ENTER I FOR INSERTION
ENTER S FOR SEARCH
ENTER E FOR EXIT THE SYSTEM

ENTER TYPE HERE >>> S

ENTER ATTRIBUTE VALUE NEED TO SEARCH >> CS

THE RECORDS WHICH HAS THIS ATTRIBUTE VALUE ARE AS THE FOLLOWING:

<table>
<thead>
<tr>
<th>DEPT</th>
<th>ID NUMBER</th>
<th>PERSON NAME</th>
<th>SALARY</th>
<th>REC NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>377649933</td>
<td>COL, TYNUS</td>
<td>11,300</td>
<td>4</td>
</tr>
<tr>
<td>CS</td>
<td>377957356</td>
<td>KIRBY, WILLIAM</td>
<td>35,000</td>
<td>12</td>
</tr>
<tr>
<td>CS</td>
<td>388779927</td>
<td>BAY, CHARLES</td>
<td>10,200</td>
<td>15</td>
</tr>
<tr>
<td>CS</td>
<td>393936556</td>
<td>ROFF, KEN</td>
<td>11,000</td>
<td>17</td>
</tr>
<tr>
<td>CS</td>
<td>355477474</td>
<td>DAVIS, KEVIN</td>
<td>23,000</td>
<td>20</td>
</tr>
<tr>
<td>CS</td>
<td>391327226</td>
<td>KARRA, MICHAEL</td>
<td>58,000</td>
<td>37</td>
</tr>
<tr>
<td>CS</td>
<td>391226632</td>
<td>KAY, KAYA</td>
<td>74,000</td>
<td>39</td>
</tr>
<tr>
<td>CS</td>
<td>335949440</td>
<td>LAM, LISA</td>
<td>39,000</td>
<td>53</td>
</tr>
<tr>
<td>CS</td>
<td>377410551</td>
<td>DOPP, JOHN</td>
<td>23,000</td>
<td>55</td>
</tr>
<tr>
<td>CS</td>
<td>352507927</td>
<td>FRITZ, ALICE</td>
<td>10,000</td>
<td>56</td>
</tr>
<tr>
<td>CS</td>
<td>366588939</td>
<td>WALLEN, DAVE</td>
<td>35,000</td>
<td>60</td>
</tr>
<tr>
<td>CS</td>
<td>322925373</td>
<td>LIN, CARLIE</td>
<td>40,000</td>
<td>74</td>
</tr>
<tr>
<td>CS</td>
<td>344124433</td>
<td>ABASS, LISA</td>
<td>74,000</td>
<td>81</td>
</tr>
<tr>
<td>CS</td>
<td>356104666</td>
<td>KAPPER, KENIN</td>
<td>74,000</td>
<td>87</td>
</tr>
<tr>
<td>CS</td>
<td>344224244</td>
<td>WARD, LILY</td>
<td>74,000</td>
<td>88</td>
</tr>
<tr>
<td>CS</td>
<td>366434000</td>
<td>WAGS, KAT</td>
<td>23,000</td>
<td>92</td>
</tr>
<tr>
<td>CS</td>
<td>399263386</td>
<td>WEBER, ROBIN</td>
<td>77,000</td>
<td>93</td>
</tr>
<tr>
<td>CS</td>
<td>322223373</td>
<td>HALL, JEFFERY</td>
<td>58,000</td>
<td>94</td>
</tr>
</tbody>
</table>

DO YOU WANT TO SEE THE CURRENT DIRECTORY DISPLAYED ON TTY (Y OR N)? N

ENTER ONE OF THE FOLLOWING FUNCTION TYPES:

ENTER C FOR CREATION
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
<table>
<thead>
<tr>
<th>45</th>
<th>31000</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>31000</td>
</tr>
<tr>
<td>73</td>
<td>31000</td>
</tr>
<tr>
<td>105</td>
<td>31000</td>
</tr>
<tr>
<td>120</td>
<td>31000</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt; Attribute Value 31000 Occurs 8 Times</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>40</th>
<th>34000</th>
</tr>
</thead>
<tbody>
<tr>
<td>112</td>
<td>34000</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt; Attribute Value 34000 Occurs 2 Times</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>12</th>
<th>35000</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>35000</td>
</tr>
<tr>
<td>36</td>
<td>35000</td>
</tr>
<tr>
<td>60</td>
<td>35000</td>
</tr>
<tr>
<td>90</td>
<td>35000</td>
</tr>
<tr>
<td>101</td>
<td>35000</td>
</tr>
<tr>
<td>112</td>
<td>35000</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt; Attribute Value 35000 Occurs 7 Times</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>73</th>
<th>36000</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>36000</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt; Attribute Value 36000 Occurs 2 Times</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>65</th>
<th>37000</th>
</tr>
</thead>
<tbody>
<tr>
<td>69</td>
<td>37000</td>
</tr>
<tr>
<td>103</td>
<td>37000</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt; Attribute Value 37000 Occurs 4 Times</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>13</th>
<th>39000</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>39000</td>
</tr>
<tr>
<td>22</td>
<td>39000</td>
</tr>
<tr>
<td>26</td>
<td>39000</td>
</tr>
<tr>
<td>31</td>
<td>39000</td>
</tr>
<tr>
<td>90</td>
<td>39000</td>
</tr>
<tr>
<td>103</td>
<td>39000</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt; Attribute Value 39000 Occurs 23 Times</td>
<td></td>
</tr>
</tbody>
</table>
>>> Attribute Value 40000 Occurs 5 Times
33 43CO0
46 43CO0
69 43CO0
94 43CO0
111 43CO0
114 43CO0

>>> Attribute Value 43000 Occurs 6 Times
3 50CC0
25 50CC0
34 50CC0
59 50CC0
82 50CC0
95 50CC0
96 50CC0
104 50CC0
115 50CC0
129 50CC0

>>> Attribute Value 50000 Occurs 12 Times
37 56CC0
58 56CC0
103 56CC0
110 56CC0

>>> Attribute Value 56000 Occurs 4 Times
42 58CC0
61 58CC0
88 58CC0
102 58CC0
134 58CC0

>>> Attribute Value 58000 Occurs 7 Times
32 72CC0
44 72CC0
89 72CC0
125 72CC0

>>> Attribute Value 72000 Occurs 4 Times
24 74CC0
28 74CC0
36 74CC0
39 74CC0

93

THE ATTRIBUTE VALUE 74000 OCCURS 19 TIMES

25 74000
36 74000
63 74000
93 74000
128 74000
150 74000

THE ATTRIBUTE VALUE 77000 OCCURS 7 TIMES

TOTAL NUMBER OF RECORDS IN THIS FILE FILE: 133
LOWEST ATTRIBUTE VALUE IN THIS FILE FILE: 30000
HIGHEST ATTRIBUTE VALUE IN THIS FILE FILE: 77000

INVERTED FILE --- INTERVAL SELECTION PROCESS:

ENTER 1 --- YOU SPECIFY THE RANGE FOR EACH INTERVAL
ENTER 2 --- YOU SPECIFY THE TOTAL NUMBER OF EQUAL INTERVAL RANGES
ENTER 3 --- YOU SPECIFY THE DESIRED NUMBER OF RECORDS PER INTERVAL
ENTER 4 --- PROGRAM SELECT INTERVALS
ENTER 5 --- EXIT THE PROGRAM

ENTER OPTION HERE >>> 4

PROGRAM MAKES UNIFORM INTERVALS PROCESS:

DO YOU WANT TO STORE THIS DIRECTORY IN A FILE INVERT.FIL (IF YOU HAVE STORED A DIRECTORY IN, ENTER N)

ENTER Y OR N HERE >>> Y

ENTER THE AVERAGE NUMBER OF RECORDS PER INTERVAL TENTER G, PROGRAM WILL USE THE SQUARE ROOT OF TOTAL NUMBER OF RECORDS AS ITS VALUE

ENTER VALUE HERE >>> G

THE PROGRAM GENERATES AVERAGE NUMBER OF RECORDS PER INTERVAL IS 11

ENTER THE MINIMUM NUMBER OF RECORDS PER INTERVAL TENTER G, PROGRAM WILL USE (0.5 X AVERAGE INTERVAL SIZE) THAT IS >> 6 AS ITS VALUE

ENTER VALUE HERE >>> 0

THE PROGRAM GENERATES MINIMUM NUMBER OF RECORDS PER INTERVAL IS 6

ENTER THE MAXIMUM NUMBER OF RECORDS PER INTERVAL TENTER G, PROGRAM WILL USE (2 X AVERAGE INTERVAL SIZE) THAT IS >> 22 AS ITS VALUE

ENTER VALUE NO LESS THAN (2 X MINIMUM INTERVAL SIZE); ELSE PROGRAM WILL USE (2 X MINIMUM INTERVAL SIZE) THAT IS >> 12 AS ITS VALUE

ENTER VALUE HERE >>> 0

THE PROGRAM GENERATES MAXIMUM NUMBER OF RECORDS PER INTERVAL IS 22

THE ATTRIBUTE VALUE 39330 OCCURS 23 TIMES
IT WILL NOT BE STORED IN THE DIRECTORY!!

<table>
<thead>
<tr>
<th>INTERVAL VALUES</th>
<th># OF RECORDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOW - HIGH</td>
<td>POINTERS TO FILE</td>
</tr>
<tr>
<td>10000 - 11000</td>
<td>14 15 16 19 47 56 75 103 118 4 17 102</td>
</tr>
<tr>
<td>21000 - 22000</td>
<td>2 7 10 29 48 55 66 72 92 100 124</td>
</tr>
<tr>
<td>31000 - 34000</td>
<td>5 8 11 45 70 73 105 120 40 112</td>
</tr>
<tr>
<td>35000 - 37000</td>
<td>12 23 36 80 98 101 132 78 80 65 69 113 131</td>
</tr>
<tr>
<td>40000 - 43000</td>
<td>9 18 51 74 133 33 46 49 64 111 114</td>
</tr>
<tr>
<td>50000 - 53000</td>
<td>3 25 34 35 59 82 84 95 96 104 115 129</td>
</tr>
<tr>
<td>54000 - 72000</td>
<td>37 58 108 110 42 61 62 86 94 109 134 32 44 89 125</td>
</tr>
<tr>
<td>74000 - 74000</td>
<td>24 28 38 39 41 43 52 71 76 77 79 81 83 87 88 91 107 126 1</td>
</tr>
<tr>
<td>77000 - 77000</td>
<td>6 20 30 63 93 129 130</td>
</tr>
</tbody>
</table>

INVERTED FILE --- INTERVAL SELECTION PROCESS:

ENTER 1 --- YOU SPECIFY THE RANGE FOR EACH INTERVAL
ENTER 2 --- YOU SPECIFY THE TOTAL NUMBER OF EQUAL INTERVAL RANGES
ENTER 3 --- YOU SPECIFY THE DESIRED NUMBER OF RECORDS PER INTERVAL
ENTER 4 --- PROGRAM SELECT INTERVALS
ENTER 5 --- EXIT THE PROGRAM

ENTER OPTION HERE >>> 5

EXIT
**A UNIFORM SYSTEM FOR DATA Base DIRECTORY --- PROCESS OF RANGE ATTRIBUTE**

**ENTER ONE OF THE FOLLOWING FUNCTION TYPES :**

ENTER C FOR CREATION
ENTER D FOR DELETION
ENTER I FOR INSERTION
ENTER S FOR SEARCH
ENTER E FOR EXIT THE SYSTEM

**ELEMENT CURRENT DIRECTORY DISPLAYED ON TTY (Y OR N) ? Y**

**THE HYBRID B-PLUS TREE & INVERTED FILES**

>>> MASTER TREE OF INVERTED FILES <<<<

**LEVEL >>1**

**NODE #>>78**

**ENTRY>>3**

LCHILD #74

ATP>31,C00

RCHILD #75

ATP>40,C00

RCHILD #76

ATP>50,C00

RCHILD #?7

*********
AN INVERTED TREE FOR RANGE ATRVAL 23,000 TO 23,000
AN INVERTED TREE FOR RANGE INTERVAL 31,000 TO 34,000

LEVEL >>1

NODE >>17
# RECS >>2
REC # >>5
ATR>>31,000

LEVEL >>2

NODE >>17
# RECS >>2
REC # >>11
ATR>>31,000

NODE >>18
# RECS >>2
REC # >>40
ATR>>31,000

NODE >>19
# RECS >>7C
ATR>>31,000
LEVEL >>1

**AN INVERTED TREE FOR RANGE INTERVAL 35,000 TO 37,000**

LEVEL >>2

**NODE #31**

# INDEX #31

LCHILD #29

INDEX #78

RCHILD #30

**NODE #29**

# INDEX #29

LCHILD #23

INDEX #36

RCHILD #24

**INDEX #65**

RCHILD #25

**NODE #30**

# INDEX #30

LCHILD #26

INDEX #98

RCHILD #27

**INDEX #113**

RCHILD #28

**INDEX #115**
LEVEL >>3

<table>
<thead>
<tr>
<th>NODE #</th>
<th>RECS</th>
<th>REC #</th>
<th>ATR</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>2</td>
<td>12</td>
<td>35,000</td>
</tr>
<tr>
<td>24</td>
<td>2</td>
<td>36</td>
<td>35,000</td>
</tr>
<tr>
<td>25</td>
<td>2</td>
<td>65</td>
<td>36,000</td>
</tr>
<tr>
<td>26</td>
<td>2</td>
<td>78</td>
<td>36,000</td>
</tr>
<tr>
<td>27</td>
<td>2</td>
<td>98</td>
<td>35,000</td>
</tr>
<tr>
<td>28</td>
<td>2</td>
<td>113</td>
<td>37,000</td>
</tr>
</tbody>
</table>

LEVEL >>1

<table>
<thead>
<tr>
<th>NODE #</th>
<th>INDEX</th>
<th>LCHILD #</th>
<th>RCHILD #</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>32</td>
<td>33</td>
<td>34</td>
</tr>
</tbody>
</table>

<<< AN INVERTED TREE FOR RANGE ATRVAL 40,000 TO 43,000 >>>
LEVEL >>2

<table>
<thead>
<tr>
<th>NODE #&gt;&gt;32</th>
<th>NODE #&gt;&gt;33</th>
<th>NODE #&gt;&gt;34</th>
<th>NODE #&gt;&gt;35</th>
<th>NODE #&gt;&gt;36</th>
</tr>
</thead>
<tbody>
<tr>
<td># REC'S &gt;&gt;2</td>
<td># REC'S &gt;&gt;2</td>
<td># REC'S &gt;&gt;2</td>
<td># REC'S &gt;&gt;2</td>
<td># REC'S &gt;&gt;2</td>
</tr>
<tr>
<td>REC # &gt;&gt;9</td>
<td>REC # &gt;&gt;13</td>
<td>REC # &gt;&gt;49</td>
<td>REC # &gt;&gt;64</td>
<td>REC # &gt;&gt;111</td>
</tr>
<tr>
<td>ATR&gt;&gt;40,000</td>
<td>ATR&gt;&gt;43,000</td>
<td>ATR&gt;&gt;43,000</td>
<td>ATR&gt;&gt;43,000</td>
<td>ATR&gt;&gt;43,000</td>
</tr>
</tbody>
</table>

>>> AN INVERTED TREE FOR RANGE ATRVAL 50,000 TO 50,000 <<<

LEVEL >>1

<table>
<thead>
<tr>
<th>NODE #&gt;&gt;46</th>
</tr>
</thead>
<tbody>
<tr>
<td># INDEXX&gt;&gt;1</td>
</tr>
<tr>
<td>LCHIL #44</td>
</tr>
<tr>
<td>RCHIL #45</td>
</tr>
</tbody>
</table>

>>>
AN INVERTED TREE FOR RANGE INTERVAL 56,000 TO 72,000

LEVEL >>1

NODE #>>56
=========
# INDEX>>1
========
LCHILD #>>54
========
INDEX >>62
RCHILD #>>55
========

LEVEL >>2

NODE #>>54
=========
# INDEX>>2
=========
LCHILD #>>47
=========
INDEX >>62
RCHILD #>>48
=========
INDEX >>58
RCHILD #>>49
=========

NODE #>>55
=========
# INDEX>>3
=========
LCHILD #>>50
=========
INDEX >>87
RCHILD #>>51
=========
INDEX >>108
RCHILD #>>52
=========
INDEX >>110
RCHILD #>>53
AN INVERTED TREE FOR RANGE ATVAL 77,000 TO 77,000

LEVEL >>1

NODE >>73

# INDEX >>2
# RECS >>2
LCHILD #70
INDEX >>30

RCHILD #71
INDEX >>93

LEVEL >>2

NODE >>70

# RECS >>5
ATR >>77,000

---

NODE >>71

# RECS >>2
---

REC # >>30
ATR >>77,000
---

REC # >>20
ATR >>77,000
---

---

NODE >>72

# RECS >>3
---

REC # >>93
ATR >>77,000
---

REC # >>128
ATR >>77,000
---

REC # >>130
ATR >>77,000
---
ENTER ONE OF THE FOLLOWING FUNCTION TYPES :
+++++++++++++++++++++++++++++++++++ 
ENTER C FOR CREATION
ENTER D FOR DELETION
ENTER I FOR INSERTION
ENTER S FOR SEARCH
ENTER E FOR EXIT THE SYSTEM
ENTER TYPE HERE >>>

ENTER ATTRIBUTE VALUE NEED TO SEARCH >>> 23000

THE RECORDS WHICH HAS THIS ATTRIBUTE VALUE ARE AS THE FOLLOWING : 

<table>
<thead>
<tr>
<th>DEPT</th>
<th>ID NUMBER</th>
<th>PERSON NAME</th>
<th>SALARY</th>
<th>REC NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>LING</td>
<td>355271562</td>
<td>CHE, KARL</td>
<td>23000</td>
<td>2</td>
</tr>
<tr>
<td>GEDG</td>
<td>322591003</td>
<td>LYON, NANCY</td>
<td>23000</td>
<td>7</td>
</tr>
<tr>
<td>SDC</td>
<td>333691365</td>
<td>SEAT, STEVE</td>
<td>23000</td>
<td>10</td>
</tr>
<tr>
<td>CS</td>
<td>356474746</td>
<td>RAGES, BETTY</td>
<td>23000</td>
<td>29</td>
</tr>
<tr>
<td>MGMT</td>
<td>327377776</td>
<td>SEIZ, DAVID</td>
<td>23000</td>
<td>48</td>
</tr>
<tr>
<td>CS</td>
<td>377455541</td>
<td>DOPP, JOHN</td>
<td>23000</td>
<td>55</td>
</tr>
<tr>
<td>MATH</td>
<td>3637474765</td>
<td>KEICH, SUNNY</td>
<td>23000</td>
<td>66</td>
</tr>
<tr>
<td>MGMT</td>
<td>3442772727</td>
<td>REN, TIK</td>
<td>23000</td>
<td>72</td>
</tr>
<tr>
<td>CS</td>
<td>3865434400</td>
<td>NASH, KAY</td>
<td>23000</td>
<td>92</td>
</tr>
<tr>
<td>REAS</td>
<td>3444555551</td>
<td>JACC, SAM</td>
<td>23000</td>
<td>100</td>
</tr>
<tr>
<td>MATH</td>
<td>383474672</td>
<td>LAM, DE</td>
<td>23000</td>
<td>124</td>
</tr>
</tbody>
</table>

DO YOU WANT TO SEE THE CURRENT DIRECTORY DISPLAYED ON TTY (Y OR N) ? N

ENTER ONE OF THE FOLLOWING FUNCTION TYPES :
+++++++++++++++++++++++++++++++++++ 
ENTER C FOR CREATION
ENTER D FOR DELETION
ENTER I FOR INSERTION
ENTER S FOR SEARCH
ENTER E FOR EXIT THE SYSTEM
ENTER TYPE HERE >>> I

ENTER ONE OF THE FOLLOWING OPTIONS FOR INSERTION :
+++++++++++++++++++++++++++++++++++ 
OPTION T REPRESENTS ENTERING DATA FROM TTY
OPTION F REPRESENTS ENTERING DATA FROM EXISTING FILE
ENTER OPTION HERE >>> T
ENTER THE FOLLOWING ATTRIBUTE VALUES FOR A NEW RECORD:

ENTR THE DEPT. NAME >> CS
ENTER EMPLOYEE ID >> 57765039
ENTER EMPLOYEE NAME >> CLARK, KARL
ENTER EMPLOYEE SALARY >> 2500

DO YOU WANT TO INSERT A NEW RECORD AGAIN (ENTER Y OR N)? Y

ENTER THE FOLLOWING ATTRIBUTE VALUES FOR A NEW RECORD:

ENTER THE DEPT. NAME >> CT
ENTER EMPLOYEE ID >> 52002576
ENTER EMPLOYEE NAME >> WILLIAM, AMY
ENTER EMPLOYEE SALARY >> 7100

DO YOU WANT TO INSERT A NEW RECORD AGAIN (ENTER Y OR N)? Y

ENTER THE FOLLOWING ATTRIBUTE VALUES FOR A NEW RECORD:

ENTER THE DEPT. NAME >> IE
ENTER EMPLOYEE ID >> 344562345
ENTER EMPLOYEE NAME >> RITA, PETER
ENTER EMPLOYEE SALARY >> 56000

DO YOU WANT TO INSERT A NEW RECORD AGAIN (ENTER Y OR N)? N

DO YOU WANT TO SEE THE CURRENT DIRECTORY DISPLAYED ON TTY (Y OR N)? N

ENTER ONE OF THE FOLLOWING FUNCTION TYPES:

ENTER C FOR CREATION
ENTER S FOR DELETION
ENTER I FOR INSERTION
ENTER S FOR SEARCH
ENTER E FOR EXIT THE SYSTEM

ENTER TYPE HERE >>> S

ENTER ATTRIBUTE VALUE NEED TO SEARCH >> 56000

THE RECORDS WHICH HAS THIS ATTRIBUTE VALUE ARE AS THE FOLLOWING:

<table>
<thead>
<tr>
<th>DEPT</th>
<th>ID NUMBER</th>
<th>PERSON NAME</th>
<th>SALARY</th>
<th>REC NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>311252726</td>
<td>KARL, MICHAEL</td>
<td>56000</td>
<td>37</td>
</tr>
<tr>
<td>BEAS</td>
<td>355521515</td>
<td>AUGUST, CHARLES</td>
<td>56000</td>
<td>58</td>
</tr>
<tr>
<td>ENGL</td>
<td>344567235</td>
<td>NICHOL, RICH</td>
<td>56000</td>
<td>108</td>
</tr>
<tr>
<td>MATH</td>
<td>344567897</td>
<td>COOK, WENY</td>
<td>56000</td>
<td>110</td>
</tr>
<tr>
<td>IE</td>
<td>344562345</td>
<td>RITA, PETER</td>
<td>56000</td>
<td>137</td>
</tr>
</tbody>
</table>
DO YOU WANT TO SEE THE CURRENT DIRECTORY DISPLAYED ON TTY (Y OR N) ? N

ENTER ONE OF THE FOLLOWING FUNCTION TYPES :
***********************************************
ENTER C FOR CREATION
ENTER D FOR DELETION
ENTER I FOR INSERTION
ENTER S FOR SEARCH
ENTER E FOR EXIT THE SYSTEM

ENTER TYPE HERE >> S

ENTER ATTRIBUTE VALUE NEED TO SEARCH >> 23500

THE RECORDS WHICH HAS THIS ATTRIBUTE VALUE ARE AS THE FOLLOWING :

DEPT  ID NUMBER  PERSON NAME  SALARY  REC NO
CS  117689009  CHANG, KARL  23550  135

DO YOU WANT TO SEE THE CURRENT DIRECTORY DISPLAYED ON TTY (Y OR N) ? N

ENTER ONE OF THE FOLLOWING FUNCTION TYPES :
***********************************************
ENTER C FOR CREATION
ENTER D FOR DELETION
ENTER I FOR INSERTION
ENTER S FOR SEARCH
ENTER E FOR EXIT THE SYSTEM

ENTER TYPE HERE >> C

ENTER ONE OF THE FOLLOWING OPTIONS FOR DELETION :
***********************************************
OPTION Y STANDS FOR ENTERING DELETED DATA FROM TTY
OPTION F STANDS FOR ENTERING DELETED DATA FROM EXISTING FILE

ENTER OPTION HERE >> T

ENTER A RECORD NO. NEED TO BE DELETED >> 137

DO YOU HAVE MORE KEY VALUES NEED TO BE DELETED (ENTER Y OR N) ? Y

ENTER A RECORD NO. NEED TO BE DELETED >> 17

DO YOU HAVE MORE KEY VALUES NEED TO BE DELETED (ENTER Y OR N) ? Y
### Enter a Record No. Need to be Deleted >> 64

Do you have more key values need to be deleted (Enter Y or N)? N

### Do you want to see the current directory displayed on TTY (Y or N)? N

Enter one of the following function types:

- Enter C for Creation
- Enter D for Deletion
- Enter I for Insertion
- Enter S for Search
- Enter E for Exit the System

Enter Type Here >> S

Enter attribute value need to search >> 43000

The records which have this attribute value are as the following:

<table>
<thead>
<tr>
<th>DEPT</th>
<th>ID NUMBER</th>
<th>PERSON NAME</th>
<th>SALARY</th>
<th>REC NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATH</td>
<td>355475775</td>
<td>WHITE, TOM</td>
<td>43,000</td>
<td>33</td>
</tr>
<tr>
<td>MB</td>
<td>31122424</td>
<td>GORDON, ANDREW</td>
<td>43,000</td>
<td>46</td>
</tr>
<tr>
<td>LING</td>
<td>34426226</td>
<td>BOOTH, SEPHEN</td>
<td>43,000</td>
<td>49</td>
</tr>
<tr>
<td>MATH</td>
<td>346635677</td>
<td>RIZA, BETTY</td>
<td>43,000</td>
<td>111</td>
</tr>
<tr>
<td>MATH</td>
<td>377455111</td>
<td>ROBINSON, SCOTT</td>
<td>43,000</td>
<td>114</td>
</tr>
</tbody>
</table>

Do you want to see the current directory displayed on TTY (Y or N)? N

Enter one of the following function types:

- Enter C for Creation
- Enter D for Deletion
- Enter I for Insertion
- Enter S for Search
- Enter E for Exit the System

Enter Type Here >> S

Enter attribute value need to search >> 56000

The records which have this attribute value are as the following:

<table>
<thead>
<tr>
<th>DEPT</th>
<th>ID NUMBER</th>
<th>PERSON NAME</th>
<th>SALARY</th>
<th>REC NO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEPT ID NUMBER</td>
<td>PERSON NAME</td>
<td>SALARY</td>
<td>REC NO</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>--------------</td>
<td>--------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td>377649733</td>
<td>11,000</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>LING</td>
<td>377646213</td>
<td>11,000</td>
<td>102</td>
<td></td>
</tr>
</tbody>
</table>

Do you want to see the current directory displayed on TTY (Y or N)? N

Enter one of the following function types:

ENTER C FOR CREATION
ENTER D FOR DELETION
ENTER I FOR INSERTION
ENTER S FOR SEARCH
ENTER E FOR EXIT THE SYSTEM

Enter type here >> S

Enter attribute value need to search >> 11000

The records which has this attribute value are as the following:

<table>
<thead>
<tr>
<th>DEPT ID NUMBER</th>
<th>PERSON NAME</th>
<th>SALARY</th>
<th>REC NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>377649733</td>
<td>11,000</td>
<td>4</td>
</tr>
<tr>
<td>LING</td>
<td>377646213</td>
<td>11,000</td>
<td>102</td>
</tr>
</tbody>
</table>

Do you want to see the current directory displayed on TTY (Y or N)? N

Enter one of the following function types:

ENTER C FOR CREATION
ENTER D FOR DELETION
ENTER I FOR INSERTION
ENTER S FOR SEARCH
ENTER E FOR EXIT THE SYSTEM

Enter type here >> E

Exit

. X/F

[Logon Other users logged-in under [56200,54226], Jobs: 25,65]
BIBLIOGRAPHY


